Rok akademicki 2015/2016

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Paweł Kaczyński

Możliwości programowania klocka LEGO EV3.

Opiekun pracy
dr inż. Henryk Dobrowolski

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Kierunek:	Informatyka
Specjalność:	Inżynieria Systemów Informatycznych
Data rozpoczęcia studiów:	2012.10.01

.....
Podpis studenta

EGZAMIN DYPLOMOWY
Złożył egzamin dyplomowy w dniu 2016r
z wynikiem .....
Ogólny wynik studiów: .....
Dodatkowe wnioski i uwagi Komisji: .....
.....
.....

Streszczenie

Celem niniejszej pracy jest przetestowanie możliwości programowania zestawu LEGO EV3 i jego zastosowań w sterowaniu robotem mobilnym. Zaprojektowana aplikacja jest napisana w języku C++ z wykorzystaniem biblioteki ev3dev. Jej główną zaimplementowaną funkcjonalnością jest automat sterujący zachowaniami robota oraz reagujący na zdarzenia generowane przez bodźce z otoczenia. W pracy opisano również system komunikacji między poszczególnymi agentami za pomocą sieci bezprzewodowej oraz przedstawiono działanie jednostki nadzorującej cały system.

Słowa kluczowe: LEGO Mindstorms EV3, robot mobilny, zachowanie złożone, automat skończony, agent, C++.

Possibilities of programming the LEGO EV3 set.

The goal of this thesis is to test the possibilities of programming the LEGO EV3 brick and its use in controlling a mobile robot. The designed application is written using C++ language and ev3dev library. Its main implemented feature is the state machine that supervises robot's behaviour and reacts to different events. This work also covers details about wireless communication between agents and describes the central unit which takes care of the whole system.

Keywords: LEGO Mindstorms EV3, mobile robot, complex behaviour, state machine, agent, C++.

Spis treści

Rozdział 1

Wstęp

Celem niniejszej pracy było zaprojektowanie i implementacja aplikacji kontrolującej zachowanie robotów mobilnych. Jest ona odpowiedzialna za nadzór nad zachowaniami robotów (agentów), sterowanie sensorami i efektorami oraz komunikację z wyższymi warstwami architektury, weryfikującymi poprawność całego systemu. Zachowania robota są opisane za pomocą automatów skończonych. Warstwa komunikacyjna dostarcza interfejs przesyłania specjalnych komunikatów za pomocą protokołu UDP.

Aplikacja jest zaprojektowana do uruchamiania na systemach Unixowych, w szczególności na klocku centralnym LEGO, a także na komputerze osobistym w roli nadzorcy. Warstwa połączeniowa jest punktem wspólnym architektury ARM z inną, używając do komunikacji interfejsu gniazd sieciowych.

1.1 Motywacja

Motywacją do podjęcia powyższego tematu było przetestowanie możliwości technicznych oraz programowania robota zbudowanego z klocków LEGO Mindstorms EV3. Aplikacja działająca na klocku była napisana z użyciem biblioteki ev3dev [4] w języku C++.

Możliwość dostępu do sterującego klockiem centralnym systemu Linux zdejmuje ograniczenie używania prostych środowisk graficznych i pozwala osiągnąć dużo więcej małym kosztem. Należało zatem sprawdzić, co najnowsza wersja LEGO Mindstorms ma do zaoferowania, w szczególności:

- Wydajność napisanych aplikacji z użyciem ww. bilbioteki
- Skuteczność komunikacji z wykorzystaniem bezprzewodowej sieci Wi-Fi
- Dokładność dostarczanych odczytów z sensorów oraz szybkość i niezawodność reakcji na nietypowe zdarzenia.

1.2 Założenia

Zostały przyjęte następujące założenia:

- Każdy agent jest zdolny do wykonywania pewnych konkretnych zachowań niezależnie od pozostałych agentów
- Zachowania te są reprezentowane za pomocą automatu skończonego
- Dany robot może, ale nie musi być zdolny do wykonania konkretnej czynności. Jest to zależne od podłączonych do niego sensorów i efektorów
- Każdy robot samodzielnie generuje sposób wykonania danej akcji na podstawie dostępnych urządzeń
- Zachowania mogą być dynamicznie tworzone z użyciem specjalnej składni
- Urządzeniem nadzorującym może być inny robot, ale pożądana jest też możliwość kontroli z poziomu zwykłego komputera
- Agenci komunikują się zarówno z jednostką centralną (nadzorująca), jak i między sobą za pomocą sieci bezprzewodowej
- Do poprawnej komunikacji wymagana jest jedna, wspólna dla wszystkich jednostek, istniejąca sieć
- System powinien dostosować się do braków w łączności, umożliwiając komunikację za pośrednictwem innych agentów
- Agent może poruszać się tylko po płaskiej powierzchni.

1.3 Zawartość rozdziałów

1. **Wstęp -** zawiera cel pracy razem z założeniami i ograniczeniami

2. **Opis systemu -** opisuje dokładnie aspekty projektowe i technologiczne oraz zawiera wytyczne dotyczące wykorzystanych narzędzi

3. **Budowa aplikacji -** szczegółowo omawia wykorzystane w projekcie klasy, ich wzajemne zależności oraz podział na moduły

4. **Zachowania -** zawiera opis budowy oraz działania zachowań złożonych robota wraz z przykładami

5. **Komunikacja -** opisuje szczegóły implementacji warstwy połączeniowej, bazującej na interfejsie gniazd sieciowych i protokole UDP, a także komunikacji wewnętrznej bazującej na zdarzeniach

6. **Testowanie aplikacji -** zawiera przebieg przykładowych testów i ich rezultaty

7. **Podsumowanie -** zawiera wnioski końcowe z przebiegu projektowania aplikacji oraz przeprowadzonych testów.

Rozdział 2

Opis systemu

Przy projektowaniu systemu, należało uwzględnić cechy charakterystyczne wielu różnych dziedzin. Aplikacja łączy w sobie działanie niskopoziomowych, sprzętowych operacji na sensorach i efektorach, zaprogramowaną logikę wyższego poziomu w języku C++ oraz komunikację sieciową w protokole UDP. Architektura ARM oraz docelowa platforma systemowa dodatkowo wpływają na podejmowane decyzje projektowe oraz implementacyjne. Szczegółowy opis użytych technologii znajduje się poniżej.

2.1 Wymagania

- Użyte oprogramowanie działa na klocku centralnym LEGO Mindstorms EV3
- Wybrana do projektu biblioteka umożliwia pracę na urządzeniach podłączanych do klocka centralnego przy użyciu języka wyższego poziomu
- Aplikacja umożliwia konstrukcję zachowań złożonych w postaci automatu skończonego
- Zachowania robota są parametryzowane
- Robot powinien reagować na generowane zdarzenia w jak najkrótszym czasie
- Robot poprawnie reaguje na odłączenie, bądź zaburzenie pracy któregokolwiek z podłączonych urządzeń
- Oprogramowanie monitoruje poziom baterii i reaguje na zmiany jego poziomu
- Oprogramowanie umożliwia komunikację z innymi robotami za pomocą sieci bezprzewodowej
- Komunikacja jest szybka i odporna na błędy oraz gubienie pakietów

- Kod aplikacji jest czytelny oraz utrzymuje niezmienną konwencję nazewnictwa
- Kod aplikacji, w szczególności pliki nagłówkowe, jest dobrze udokumentowany.

2.2 Użyta technologia

2.2.1 Dostępne środowiska

Istnieje wiele dostępnych środowisk na licencji otwartego oprogramowania, dedykowanych LEGO Mindstorms EV3, z których każde ma trochę inne zastosowanie. Poniżej opisane są najbardziej popularne z nich.

leJOS

Projekt leJOS [5] jest oprogramowaniem zastępującym domyślnie zainstalowany system na klocku centralnym. Bazuje on na maszynie wirtualnej języka Java i jest kompatybilny z platformami Linux, Windows oraz Mac OS X. Programowanie aplikacji oraz transfer plików wykonywalnych do robota odbywa się za pomocą wtyczki do środowiska Eclipse lub za pomocą linii poleceń.

Środowisko udostępnia wiele możliwości języka Java, w szczególności obsługę synchronizacji i wyjątków, wątki oraz większość klas z pakietów java.lang, java.util oraz java.io. Dostarcza ono także ujednolicony interfejs do obsługi sensorów. Na wyświetlaczu klocka centralnego udostępniony jest nowy interfejs graficzny z wieloma dodatkowymi funkcjonalnościami. Ponadto, leJOS posiada wiele zdefiniowanych modułów obsługujących połączenia z wieloma robotami, algorytmy lokalizacyjne czy dostęp do konsoli klocka LEGO. Co ważniejsze, są one uruchamiane na zwykłym komputerze, a nadzór odbywa się zdalnie.

Mimo oferowanego bogactwa możliwości, środowisko leJOS nie byłoby dobrym wyborem. Zastąpienie oryginalnego oprogramowania klocka, dodatkowe wykorzystanie zasobów urządzenia przez maszynę wirtualną, czy brak niskopoziomowego wsparcia to główne powody dla których ta biblioteka została odrzucona.

MonoBrick

Innym wieloplatformowym środowiskiem, które współpracuje z zestawem Mindstorms EV3, jest biblioteka .NET o nazwie MonoBrick [3]. Wspiera ona języki takie jak C#, F# czy IronPython. Instalacja oprogramowania nie zastępuje domyślnego systemu na klocku, ale uruchamiana jest oddzielnie z poziomu karty SD. Oferowane możliwości są prawie tak duże, jak w przypadku leJOS, aczkolwiek projektowi brakowało solidnej dokumentacji, a także wiele odnośników na stronie prowadziło do nieistniejących witryn.

Powoływanie się na niepewną bibliotekę, która nie umożliwia natywnej kompilacji kodu, byłoby dużym błędem. Projekt prężnie się rozwija, ale w trakcie wyboru docelowego oprogramowania nie był wystarczająco stabilny i funkcjonalny.

ev3dev

Najbardziej dogodnym rozwiązaniem okazał się projekt ev3dev. Jest to dopasowana do potrzeb klocka LEGO dystrybucja Linuxa (Debian Jessie), która jest wgrywana na kartę SD i uruchamiana obok istniejącego systemu. Zawsze istnieje możliwość przywrócenia domyślnego stanu klocka przez wyjęcie karty z systemem. Platforma stworzona w ramach ev3dev zawiera wiele sterowników, nie tylko do akcesoriów zestawu EV3, ale także poprzednich dystrybucji LEGO Mindstorms oraz komponentów wytwarzanych przez osoby trzecie. Możliwe jest programowanie klocka w języku C/C++, ale ev3dev obsługuje też wiele innych języków, takich jak Python czy Lua. To wszystko daje dużą swobodę samego programowania, jak i sposobu tworzenia programu i komunikacji z urządzeniem. Kompilacja aplikacji może odbywać się bezpośrednio na urządzeniu lub na komputerze z wbudowanym kompilatorem na procesory typu ARM. Komunikacja z klockiem centralnym realizowalna jest na trzy sposoby: za pomocą Wi-Fi, Bluetooth lub przy użyciu kabla USB.

Zalety, takie jak praca w natywnym, obiektowym języku na natywnej platformie, niskopoziomowość, pełna kontrola nad sprzętem oraz brak narzutów maszyn wirtualnych zadecydowały o przyjęciu ev3dev jako biblioteki wykorzystanej w tym projekcie.

2.2.2 Sprzęt

Parametry klocka centralnego:

System: Debian Jessie

Wyświetlacz: 178x128 pikseli, monochromatyczny

Procesor: TI Sitara AM1808 300MHz

Pamięć: 64MB RAM, 16MB Flash, slot na kartę microSDHC

Sieć: WiFi, Bluetooth, USB.

2.2.3 Konfiguracja

Każda jednostka z wgraną biblioteką ev3dev posiada użytkownika `robot` z hasłem `maker`. Konfiguracja nowego systemu odbyła się w kilku krokach:

1. Na kartę microSDHC wgrany został specjalne przygotowany obraz systemu, pobrany ze strony głównej projektu ev3dev

2. Pierwsze uruchomienie robota z wgranym obrazem na karcie było poprzedzone długim przygotowaniem wszystkich plików

3. Przy użyciu połączenia SSH przez kabel USB, system został skonfigurowany i pobrane zostały wszystkie wymagane pakiety

4. Dalsza komunikacja odbywała się bezprzewodowo z użyciem urządzenia NETGEAR WNA1100, podłączonego do portu USB klocka centralnego

5. Aplikacja była kompilowana na laptopie z systemem Ubuntu 15.10 i synchronizowana zdalnie z robotem.

2.2.4 Biblioteka

W ramach projektu ev3dev dostępne są dwa pliki źródłowe napisane w języku C++. Dostarczają one wymagany interfejs do sterowania klockiem centralnym i podłączonymi do niego urządzeniami.

Wersja użytej biblioteki: 0.9.3-pre, rev 2.

Wprowadzone zmiany

Początkowo, w wersji 0.9.2-pre, biblioteka nie była kompatybilna ze wszystkimi urządzeniami dostarczonymi przez LEGO. Wymagane było dopisanie rozpoznawalnych nazw sterowników dla sensorów oraz zaimplementowanie własnej obsługi diod LED przedniego panelu, w szczególności funkcji migania. Ponadto dopisane zostało specjalne pole klasy `ev3dev::motor` pamiętające typ użytego efektora.

W finalnej fazie projektowania, biblioteka została zaktualizowana do najnowszej wersji 0.9.3-pre, co zdjęło konieczność posiadania specjalnych nazw sterowników, a także znacząco przyspieszyło obsługę interfejsu klocka oraz transmisję danych.

2.2.5 Możliwości

Użyte środowisko Linux orax język programowania C++ dostarczają wiele możliwości programistycznych, a w szczególności:

- Użycie biblioteki `stl` i zgodność ze standardem C++11
- Wątki
- Polimorfizm
- Komunikację przez protokół UDP z wykorzystaniem gniazd.

2.2.6 Narzędzia

Projekt aplikacji był rozwijany z wykorzystaniem narzędzie NetBeans. Mimo iż sama aplikacja może być skompilowana z poziomu konsoli i narzędzia Makefile, NetBeans dostarcza także wygodne narzędzia debugujące oraz przyspiesza pisanie kodu.

Konfiguracje aplikacji

Zostały zdefiniowane dwie domyślne konfiguracje:

- D_ARM: konfiguracja przeznaczona na urządzenia z procesorami typu ARM. Domyślnie przeznaczona do uruchamiania na robocie mobilnym

  Kompilator: arm-linux-gnueabi-g++

  Flagi kompilacji: -D_GLIBCXX_USE_NANOSLEEP -pthread -static-libstdc++ -std=c++11 -DAGENT

- D_DESKTOP: konfiguracja kompilowana z myślą o tradycyjnych komputerach osobistych. Domyślnie przeznaczona do uruchomienia w trybie nadzorcy systemu, komunikującego się zdalnie z robotami.

  Kompilator: g++

  Flagi kompilacji: -D_GLIBCXX_USE_NANOSLEEP -pthread -static-libstdc++ -std=c++11

Obie konfiguracje posiadają dodatkowo wersję z przedrostkiem `R_`, które oznaczają wersję Release zamiast wersji Debug.

Inne użyte narzędzia to przede wszystkim aplikacja Doxygen służąca generowaniu dokumentacji kodu programu, system kontroli wersji Git do zarządzania całym projektem oraz oprogramowanie LaTeX, za pomocą którego wygenerowany został ten dokument.

Środowisko przygotowane przez ev3dev umożliwia zdalne debugowanie aplikacji. Szczegółowe informacje zawarte są w dodatku A.3.

2.2.7 Konstrukcja robota

W celu przetestowania zaimplementowanych funkcjonalności, wyposażono robota w następujące elementy:

- Dwa duże motory do poruszania się po płaskiej powierzchni
- Ultradźwiękowy sensor odległości ustawiony w kierunku poruszania się

- Przedni zderzak oraz sensor dotyku do wykrywania zderzeń
- Sensor koloru do wykrywania zmian w odcieniu powierzchni.

Rozdział 3

Budowa aplikacji

3.1 Moduły i klasy

W celu uzyskania przejrzystości aplikacji, wydzielone zostały moduły[1], które opisują pewien fragment funkcjonalności programu. Moduły niższych warstw mogą być wykorzystywane przez moduły warstw wyższych, lub być tylko zestawem dodatkowych narzędzi. Kolejne punkty opisują w czym dany moduł się specjalizuje i jakie klasy wchodzą w jego skład. Szczegółowe dane na temat klas oraz ich metod i pól znajdują się w dokumentacji kodu w katalogu `doc`.

3.1.1 Komendy

Klasy komend są tak naprawdę nakładką na istniejące mechanizmy biblioteki ev3dev, operujące bezpośrednio na sprzęcie. Oprócz właściwej komendy, będącej poleceniem dla efektora bądź sensora, dana klasa zawiera referencje do obiektu, na którym ma zostać wykonana oraz jej parametry, o ile takowe posiada. Nazewnictwo klas dokładnie odwzorowuje nazwy komend przekazywanej urządzeniom. W obrębie konkretnych komend, definiowane są także stałe opisujące charakter przekazywanych argumentów oraz ich limity.
Komendy zostały podzielone na dwie podgrupy:

Komendy motorów: Klasa bazowa - `CommandMotor`. Zawierają referencje do klasy `Motor` oraz opcjonalnie przechowują także przekazywane parametry.
Np przykład: `CommandMotorStop`, `CommandMotorRunForever`.

Komendy sensorów: Klasa bazowa - `CommandSensor`. Zawierają referencje do klasy `Sensor`. Definiują obsługiwane tryby danego sensora. Komendy te nie służą do pobierania wartości, lecz tylko do zmiany ustawień sensora. Po-

[1] W kontekście tej pracy *moduł* oznacza pewną grupę skojarzonych ze sobą klas.

bieranie wartości używane jest przy pomocy specjalnej klasy `Devices`. Przykładowa komenda: `CommandSensorSetMode`.

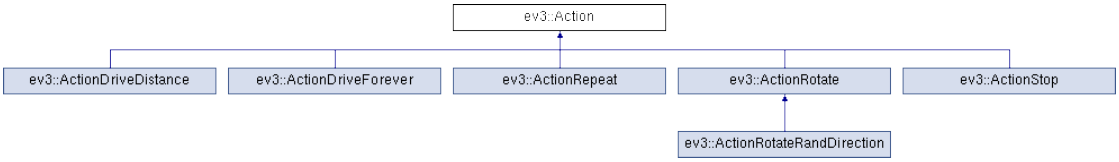
Klasą bazową dla wszystkich komend jest klasa `Command`.

3.1.2 Akcje

Akcje są kolejnym stopniem abstrakcji definiowania zachowań robota. Klasy akcji przechowują przede wszystkim sekwencje komend, które mają zostać wykonane. Ponadto, z powodu natychmiastowego charakteru wykonywania wszystkich zgromadzonych komend, akcja może mieć zdefiniowany warunek jej zakończenia. Przyjmuje ona postać funkcji anonimowej, w której następuje zwrócenie wartości logicznej na podstawie dowolnie sprecyzowanych instrukcji. Pozwala to wyższej warstwie sterującej sprawdzić, czy kolejna akcja może zostać wykonana. Dodatkowo, akcje mogą deklarować dopuszczalne zdarzenia, które przerywają jej działanie lub zmieniają jej parametry.

Wszystkie dostępne klasy akcji są zdefiniowane w aplikacji i nie istnieje możliwość zwiększenia zbioru o nowe bądź dynamicznego generowania nowych, własnych klas. Ta decyzja implementacyjna jest podyktowana specyfiką konkretnych modeli robotów, różniących się budową oraz podłączonymi akcesoriami, a co za tym idzie, odmiennym sposobem implementacji tych samych czynności.

Klasy opisujące konkretne akcje, np. `ActionDriveDistance`, dziedziczą po klasie `Action`. Wspólnymi elementami każdej z nich są: typ, warunek końcowy, sekwencja komend oraz metody wykonawcze. Różnią się natomiast dodatkowymi parametrami, takimi jak prędkość czy kąt obrotu. Ponadto, istnieje możliwość wygodnego zapętlenia jednej lub wielu akcji dowolną liczbę razy za pomocą specjalnej klasy - `ActionRepeat`. Konstruktor tej klasy przyjmuje liczbę powtórzeń oraz listę akcji, które zostaną wykonane w podanej kolejności.

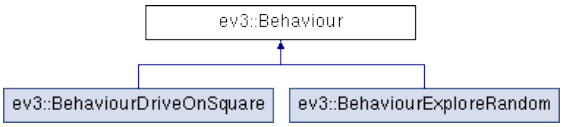


Rysunek 3.1: Diagram dziedziczenia klas akcji.

3.1.3 Zachowania

Definiowanie zachowań jest dużo trudniejsze niż akcji czy komend, gdyż bazują one na schemacie automatu skończonego. Oprócz konkretnych akcji, na jakich dane zachowanie ma się opierać, należy zdefiniować przejścia pomiędzy stanami (akcjami) w toku poprawnego wykonania oraz specjalne warunki zmiany stanu w reakcji na zaistniałe zdarzenia (np. napotkana przeszkoda lub utrata połączenia).

Podobnie jak w przypadku akcji i komend, każde zachowanie zdefiniowane jest w osobnej klasie (np. `BehaviourExplore`), które dziedziczy po wspólnej klasie `Behaviour`. Każde z nich zawiera typ, strukturę stanów automatu złożonych z akcji i przejść oraz funkcje wykonawcze. Ponadto zachowania pochodne zawierają własne parametry, np. maksymalny dystans do przejechania. Więcej szczegółów w rozdziale 4. Zachowania.



Rysunek 3.2: Diagram dziedziczenia klas zachowań.

3.1.4 Urządzenia

Biblioteka ev3dev dostarcza wygodnego interfejsu do zarządzania urządzeniami przez wygenerowanie drzew klas dla sensorów i efektorów. W ramach tej aplikacji, na każdy typ urządzenia została nałożona specjalna klasa pośrednicząca, w środku której znajduje się referencja do właściwego obiektu. Są to klasy `Motor` oraz `Sensor`, które po odpowiedniej identyfikacji zostają zmapowane do par port-urządzenie.

Całością nadzoruje klasa `Devices` napisana zgodnie ze wzorcem projektowym singleton. Ograniczone są w ten sposób nadmierne kopie obiektów i potencjalnie niejednoznaczne odwołania. Ponadto klasy urządzeń są potrzebne w wielu różnych miejscach aplikacji, a wzorzec ten umożliwia taki dostęp za pomocą statycznego wydobycia instancji.

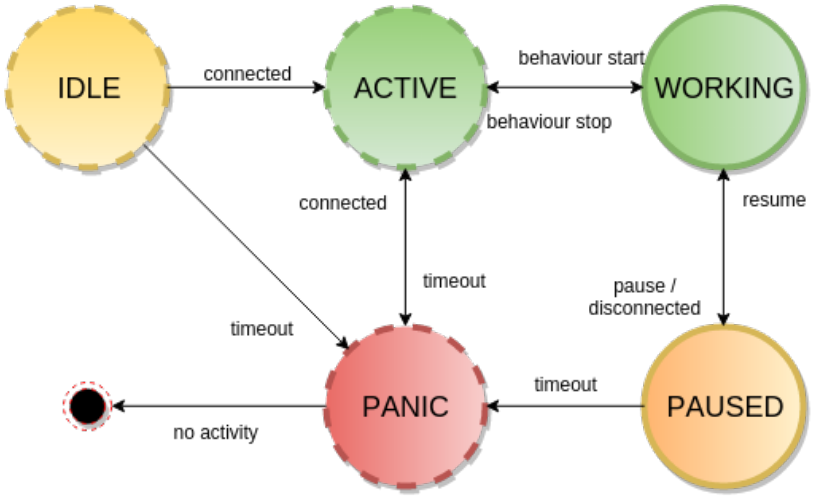
Moduł urządzeń jest także odpowiedzialny za detekcję zdarzeń. Wyższe warstwy mogą zgłaszać zdarzenia, na które klasa `Devices` ma nasłuchiwać. Jeśli dane zdarzenie wystąpi, wysyłane jest do odpowiedniej kolejki dla wyższych warstw do przetworzenia. Zgłaszane mogą być również zdarzenie niezależne, np. niski poziom baterii lub odłączenie urządzenia.

3.1.5 Robot

Jest to najbardziej rozbudowana klasa, ponieważ agreguje w sobie działanie wielu modułów. Zarządza zarówno urządzeniami podłączonymi do klocka centralnego, steruje zachowaniem robota oraz przetwarza przesłane komunikaty i zdarzenia. Podstawowa metoda klasy `run` jest uruchamiana w głównym wątku aplikacji i w pętli przetwarza wszystkie dane. Jest bezpośrednio zsynchronizowana z wątkiem komunikacyjnym za pomocą dwóch kolejek wiadomości - nadawczej i odbiorczej. Wyróżnienie dwóch niezależnych ścieżek wykonania umożliwia lepsze zarządzanie zasobami sprzętowymi oraz pozwala robotowi na samodzielne działanie, niezależnie od przesyłanych pakietów.

Robot może być fizycznie zbudowany na wiele różnych sposobów. Dlatego wymagane jest, żeby zdefiniowane były konkretne klasy implementujące szczegóły danego modelu. W związku z powyższym, klasa bazowa `Robot` jest klasą abstrakcyjną, a każdemu wariantowi konstrukcji odpowiada osobna klasa podrzędna. W celu ujednolicenia interfejsu, każdy model deklaruje listę obsługiwanych akcji oraz wymaganych do tego podłączonych urządzeń. W ten sposób można łatwo sprawdzić, czy dane zachowanie jest obsługiwane i jakich pomiarów robot może dostarczyć. Klasy konkretnych modeli, bazując na istniejących funkcjach wirtualnych, definiują swoje wersje w sposób adekwatny do domniemanego efektu końcowego wymaganych akcji.

Każda instancja robota posiada również maszynę stanów oraz zdefiniowane warunki przejść pomiędzy nimi. Każdy stan przetwarza tylko konkretne komunikaty i zdarzenia. Rezultatem ich przetworzenia może być wysłanie specjalnej wiadomości zwrotnej lub zmiana stanu na inny. To ostanie wiąże się jeszcze ze zmianą koloru oraz częstotliwości migania diod LED umieszczonych na przednim panelu klocka centralnego. Możliwe stany przedstawia diagram 3.3.



Rysunek 3.3: Diagram przejść stanów robota. Przerywana linia oznacza migające diody w kolorze stanu, ciągła stałe świecenie.

3.1.6 Komunikacja

Komunikacja pomiędzy różnymi partiami całego systemu odbywa się na dwóch poziomach:

Zdarzenia

Poszczególne moduły robota, takie jak akcje czy klasa urządzeń, mogą generować zdarzenia. Kolejka zdarzeń posiada wiele punktów wejścia, ale tylko jedno wyj-

ście - klasę `Robot`. Wszystkie klasy zdarzeń dziedziczą po klasie `Event`. Zgłaszane do kolejki obiekty mogą dotyczyć zarówno zmian wartości sensorów, niespodziewanych zmian w przebiegu zachowania lub wyjątków rzuconych przez aplikację. Więcej szczegółów w rozdziale 5. Komunikacja.

Protokół UDP

Komunikacja pomiędzy robotami odbywa się przy użyciu sieci bezprzewodowej oraz protokołu UDP. Każdy wysłany pakiet to zakodowany do postaci znakowej obiekt klasy `Message`. Każdy z nich zawiera pięć elementów: identyfikator wiadomości, nadawcy i odbiorcy, typ oraz listę opcjonalnych parametrów. Separatorem parametrów jest dwukropek. Podstawą ustalenia szczegółów wiadomości jest jej identyfikator oraz typ. Pierwszy komponent zapewnia synchronizację pakietów oraz eliminację duplikatów, a drugi steruje zmianami stanów agenta oraz jego zachowań. Więcej szczegółów w rozdziale 5. Komunikacja.

3.1.7 Nadzorca

Do poprawnej i kontrolowanej pracy całego systemu potrzebny jest jego nadzorca. Ten wyspecjalizowany moduł zarządza pozostałymi agentami, będąc punktem wspólnym komunikacji wszystkich jednostek. Pozwala mu to na rozdzielanie zachowań dla poszczególnych robotów, synchronizację komunikacji oraz zbieranie danych. Nadzorcą może być zarówno robot LEGO jak i dowolny komputer osobisy, co w tym drugim przypadku umożliwia wykorzystanie większej mocy obliczeniowej.

Za kontrolę systemu odpowiada klasa `Master`. Podobnie jak klasa `Robot`, zawiera ona dwie kolejki do wymiany wiadomości z pracującym równolegle wątkiem komunikacji. Oprócz tego, przechowuje ona informacje o wszystkich agentach w specjalnej klasie `Agent`, która odpowiada konkretnemu fizycznemu robotowi, ale jest pozbawiona wszystkich komponentów sterujących zachowaniami. Jej głównymi atrybutami są identyfikator urządzenia oraz identyfikator wiadomości. Pierwszy z nich przydzielany jest agentowi, gdy ten po raz pierwszy nawiąże połączenie z nadzorcą. Drugi z kolei wykorzystywany jest do synchronizacji przesyłanych pakietów w celu dobierania par zapytanie-odpowiedź oraz pomaga usunąć nadmiarowe duplikaty.

Największym problemem przed jakim mogą stanąć roboty (w krytycznej sytuacji - wszystkie jednocześnie), to utrata połączenia z nadzorcą. Szczegóły takiego zdarzenia opisane są w sekcji 5.3 Utrata połączenia z nadzorcą.

3.1.8 Moduły dodatkowe

Główne klasy aplikacji są wspierane przez dodatkowe, mniej znaczące moduły. Do takich należą:

Kontrola diod LED

Klasa `LedControl` powstała z konieczności, ponieważ dostarczony interfejs poprawnie obsługiwał tylko jedną z czterech dostępnych diod. Dostarczono zatem metod, które pozwalają sterować jasnością konkretnych diod, wybierać kolor oraz zlecać miganie z odpowiednim interwałem. Klasa ta jest głównie wykorzystywana do wizualnej identyfikacji stanu, w jakim obecnie znajduje się robot.

Logowanie

Zarówno w procesie tworzenia aplikacji, jak i w późniejszym jej użytkowaniu, informacje o przebiegu wykonania są niezbędne. Klasa `Logger` pozwala zdefiniować poziom obsługiwanych komunikatów, od bardzo rozwlekłego (verbose) do zawężonego tylko do błędów (error). Ponadto, można wyznaczyć domyślny strumień wyjściowych danych, taki jak standardowe wyjście bądź zapis do pliku. W przypadku zapisywania wiadomości na dysk, dane są gromadzone w paczce i zrzucane na nośnik co pewien interwał, w celu ograniczenia ilości operacji dyskowych.

Obsługa sygnałów

Aplikacja musi obsługiwać dostarczane do niej sygnały i poprawnie na nie reagować. Przy starcie programu, tworzony jest obiekt klasy `SignalHandler`, który odwołując się obiektów sterujących, wysyła żądanie zatrzymania po otrzymaniu konkretnych sygnałów.

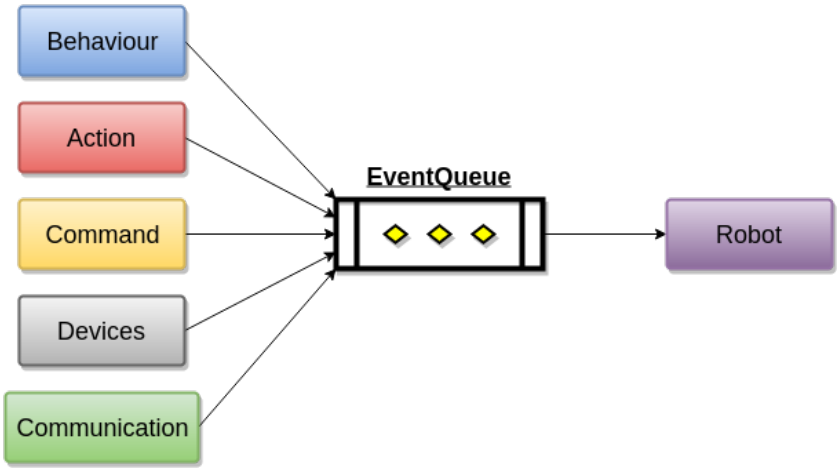
Kolejki i bufory

Zarówno komunikacja wewnętrzna jak i zewnętrzna, korzysta z specjalnych klas do przesyłania oraz przechowywania danych. W przypadku wiadomości, wątek główny i komunikacyjny korzystają z synchronizowanych obiektów klasy szablonowej `Queue`, która implementuje dobrze znaną kolejkę wraz z obsługą współbieżności. Klasa `EventQueue`, stworzona w oparciu o wzorzez singleton, implementuje kolejkę obiektów typu `Event`. Ostatnią kolekcją danych jest klasa szablonowa `CircularBuffer`, implementująca bufor cykliczny z nałożonym limitem obiektów w nim przechowywanych. Wykorzystywana głównie w wątku komunikacyjnym do eliminacji nadmiarowych pakietów.

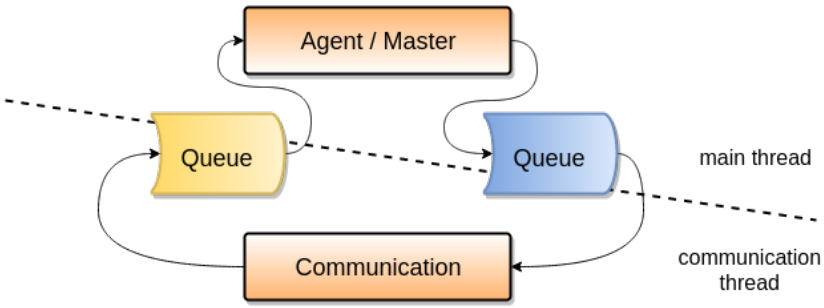
Przepływ danych został przedstawiony na rysunkach 3.4 i 3.5.

Inne

Do innych, mniej znaczących elementów należy plik `Utils`, zawierający statyczne definicje powszechnie wykorzystywanych metod, stałych i uproszczonych nazw typów danych, a także klasa `ColorUtils` wspomagająca kolorowanie wiadomości logowanych na ekran.



Rysunek 3.4: Kolejka zdarzeń. Wiele klas generujących zdarzenia odbierane przez klasę Robot.



Rysunek 3.5: Synchronizacja wiadomości pomiędzy wątkami za pomocą kolejek wiadomości.

Rozdział 4

Zachowania

Zachowania definiują najwyższy stopień abstrakcji sterowania agentem. Oprócz właściwych akcji, które wykonują, mogą mieć zdefiniowane również zdarzenia, które wymuszają niestandardowy przebieg wykonania. Ponadto, zlecają również śledzenie konkretnych wartości na podłączonych urządzeniach.

4.1 Budowa zachowań złożonych

Każde zachowanie składa się z kilku elementów:

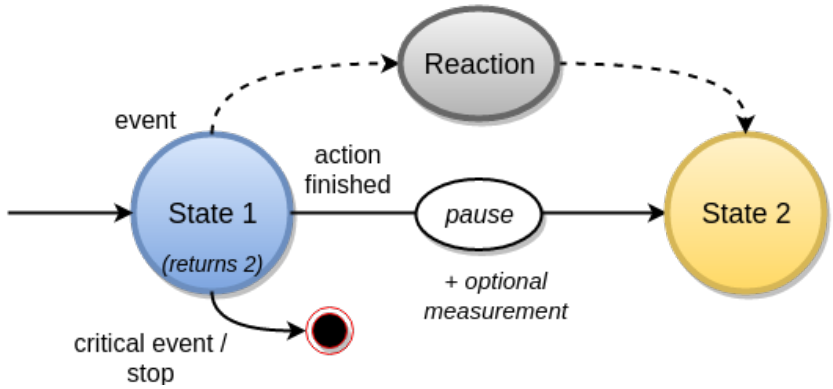
- Listy stanów, z których każdy zawiera konkretną akcję oraz zdefiniowane przejście
- Reakcji, które wystąpią po zajściu danego zdarzenia (np. cofnięcie się po uderzeniu w przeszkodę)
- Stanu stopującego działanie agenta, wykorzystywanego w razie problemów
- Opcjonalnych parametrów.

4.1.1 Tworzenie zachowań

Stworzenie zachowania może odbyć się na dwa sposoby. Pierwszym z nich jest skorzystanie z zachowania już zdefiniowanego w bibliotece. W tym przypadku zbierane są ustalone wcześniej, odpowiednio ukonkretnione akcje razem z warunkami przejść. Drugim wariantem jest stworzenie własnego zachowania. Jeśli taki typ zostanie wykryty, do agenta należy dostarczyć dodatkowe dane opisujące użyte akcje, przejścia i zdarzenia.

4.1.2 Sterowanie wykonywanymi akcjami

Klasa `Behaviour` przechowuje informację o bieżącym stanie. W każdej iteracji pętli głównej robota, następuje sprawdzenie, czy nie otrzymano instrukcji zakończenia z warsty wyższej lub czy nie wystąpiło zdarzenie krytyczne. Następnie przetwarzany jest bieżący stan. Jeżeli akcja została wykonana, sprawdzany jest jej warunek zakończenia. W przypadku pomyślnego zakończenia akcji, następuje przejście do domyślnego stanu następnego. Jeśli natomiast w trakcie wykonywania akcji zostanie przechwycone zdarzenie, jest ono porównywane z listą obsługiwanych zdarzeń danego stanu. Typowy scenariusz zakłada wybranie stanu pośredniego, który prawidłowo zareaguje na zaistniałe zdarzenie, a następnie przekaże kontrolę dalej, zapewniając normalny dalszy przebieg (rysunek 4.1). Jeśli zdarzenie nie jest obsługiwane, ewentualna zmiana stanu jest ignorowana (zdarzenia krytyczne obsługiwane są przez warstwę wyższą).



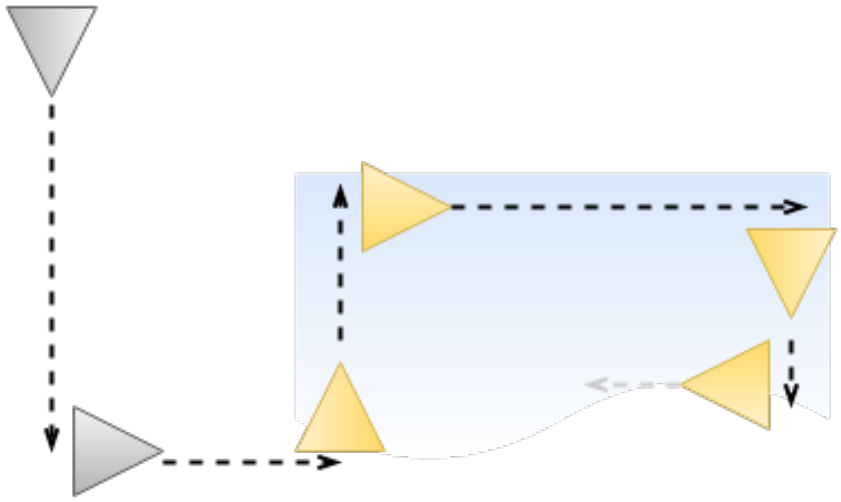
Rysunek 4.1: Diagram stanów zachowania oraz przetwarzania reakcji.

Jeśli stan zwróci wartość nieujemną, to oznacza ona identyfikator kolejnego stanu. Możliwe jest natomiast pominięcie definicji następnego stanu w przypadku akcji o nieokreślonych granicach wykonywania. Za poprawny przebieg wykonania odpowiadają wtedy zdefiniowane przejścia zachodzące po wystąpieniu zdarzenia lub zinterpretowane komunikaty z sieci.

4.1.3 Zbieranie danych

Bieżące zachowanie robota może wymuszać zbieranie danych z podłączonych sensorów. Należy wtedy zdefiniować specjalną listę zawierającą typy sensorów, których pomiar ma być przekazywany dalej. W każdym kroku przetwarzania zachowania generowane są raporty typu SENSOR_VALUE, które umieszczone w kolejce wiadomości robota mogą być przekazane innemu urządzeniu, domyślnie nadzorcy. Jest to rozszerzenie funkcjonalności zachowań o dodatkowy cel, bowiem np. znalezienie interesującego fragmentu otoczenia może skutkować zamianą logiki robota

na inną z zamiarem eksploatacji danego miejsca (rysunek 4.2). Przykładowym scenariuszem może być np. szukanie powierzchni o konkretnym kolorze, a następie poruszanie się tylko w jej obrębie.



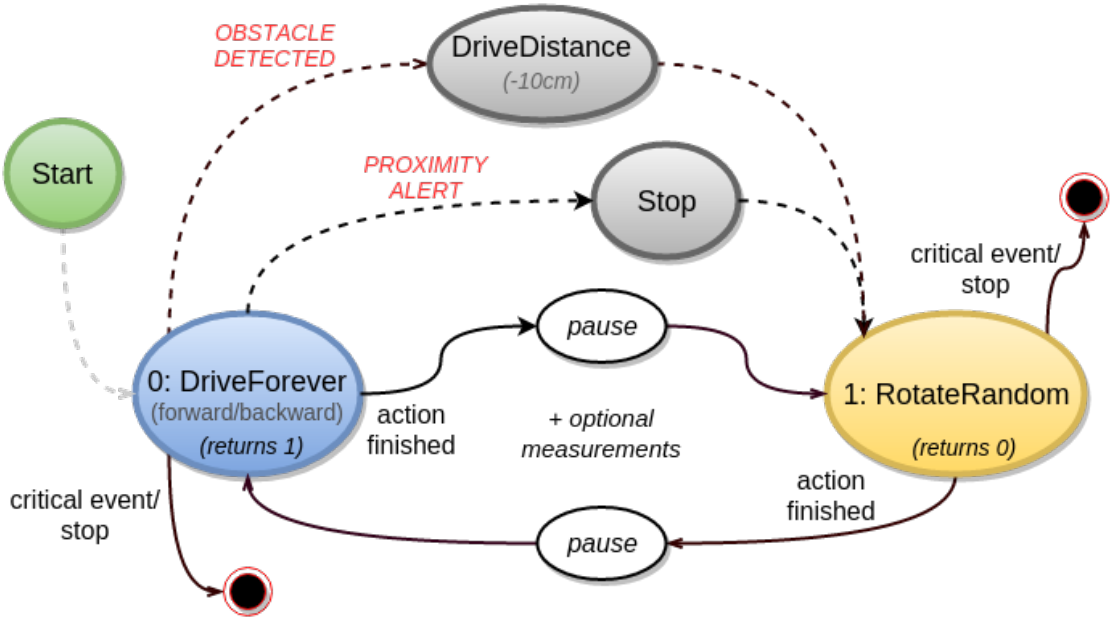
Rysunek 4.2: Uproszczony schemat przykładowego przebiegu eksploracji z późniejszą eksploatacją.

4.2 Przykładowe zachowania

Definicje zachowań mogą przyjmować różne formy. Przykładowo, zadanie zwiedzania otoczenia może być zdefiniowane jako poruszanie się z pilnowaniem lewej lub prawej krawędzi lub poruszanie się po prostej aż do napotkania przeszkody i obieranie ustalonego bądź losowego kąta obrotu. Niezależnie od zdefiniowanego rodzaju zachowania, ważny pozostaje jej cel. Jednakże jego wyznaczaniem i interpretacją zajmuje się warsta wyższa, a z poziomu zachowania można wydobyć tylko informację, czy dany cel został osiągnięty i czy nadal jest osiągalny. Poniżej przedstawione jest przykładowe zachowanie.

4.2.1 Zwiedzanie otoczenia

Na rysunku 4.3 znajduje się przykładowy algorytm zwiedzania otoczenia. Składa się z dwóch podstawowych stanów oraz dwóch stanów reakcji, zachodzących po zajściu konkretnych zdarzeń. Na uwagę zasługują specjalne stany pośrednie (oznaczone na diagramie nazwą *pause*), których obecność jest krytyczna zarówno dla dokładności akcji, jak i pomiarów. Przez ich krótki czas trwania, robot wytraca swoją prędkość, kalibrując pozycję motorów oraz dokonując adekwatnego pomiaru z wybranych sensorów. Jeśli zaszło zdarzenie krytyczne lub zachowanie musi zostać zatrzymane, następuje przejście do stanu końcowego.



Rysunek 4.3: Szczegółowy diagram zwiedzania otoczenia.

Rozdział 5

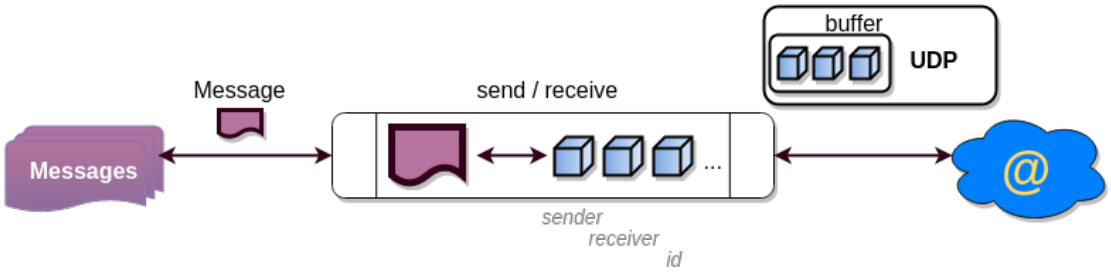
Komunikacja

5.1 Komunikacja przez sieć

Komunikacja przez sieć odbywa się przy użyciu protokołu UDP oraz z wykorzystaniem interfejsu gniazd systemu Unix. Domyślnie, ustawione są następujące parametry:

- Numer portu: 12345
- Maksymalny rozmiar paczki: 4kB
- Liczba wysyłanych kopii pakietów: 3
- Wielkość kolejki przechowującej ostatnio odebrane pakiety: 50

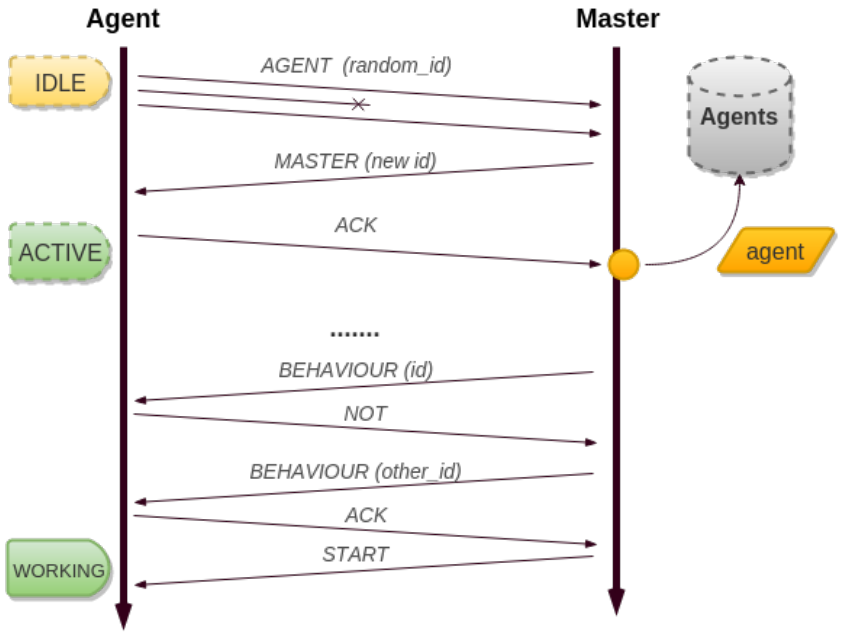
W klasie `CommUtils` zdefiniowane są metody `sendMessage, receiveMessage` oraz `receiveMessageDelay`. Przetwarzają one zdefiniowane struktury aplikacji na niskopoziomowe bufory danych wysyłane lub odbierane za pomocą systemowych funkcji (Rysunek 5.1). W trakcie ich wykonywania sprawdzane są dostępne interfejsy sieciowe oraz następuje tworzenie gniazd na zadanym porcie. Metoda `receiveMessage` jest blokująca, natomiast `receiveMessageDelay` czeka na odpowiedź tylko przez zadany w milisekundach interwał.



Rysunek 5.1: Uproszczony diagram niskopoziomowej komunikacji.

5.1.1 Synchronizacja z nadzorcą

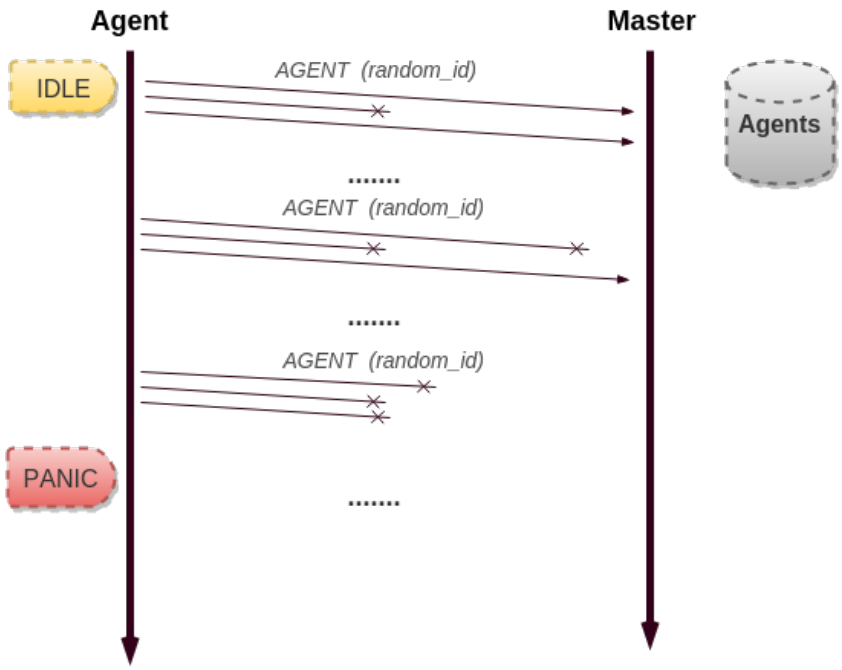
Na samym początku, tuż po uruchomieniu agentów i nadzorcy, następuje synchronizacja. Nadzorca nasłuchuje sieć, oczekując na wiadomości typu AGENT. Uruchomiony agent jest w stanie IDLE i wysyła wiadomość typu AGENT na adres rozgłoszeniowy z losowym identyfikatorem agenta, oczekując na komunikat MASTER. Czekanie nie trwa dłużej niż zadany interwał czasu (domyślnie nadzorca ma 10 sekund na odpowiedź). Nadzorca odbiera żądanie od agenta i dodaje go do swojej bazy, jednocześnie przypisując mu identyfikator oraz zapamiętując adres ip. Następnie na adres wysyłany jest komunikat MASTER z nowo przydzielonym identyfikatorem wpisanym w miejsce odbiorcy. Jeśli agent przyjmie taki pakiet, zwraca wiadomość typu ACK. Po wymianie wiadomości (rysunek 5.2), dany robot jest zsynchronizowany, ma przypisany numer identyfikujący oraz jest odnotowany w bazie. Następuje przejście do stanu ACTIVE, oznaczające danego agenta, jako aktywnego w systemie.



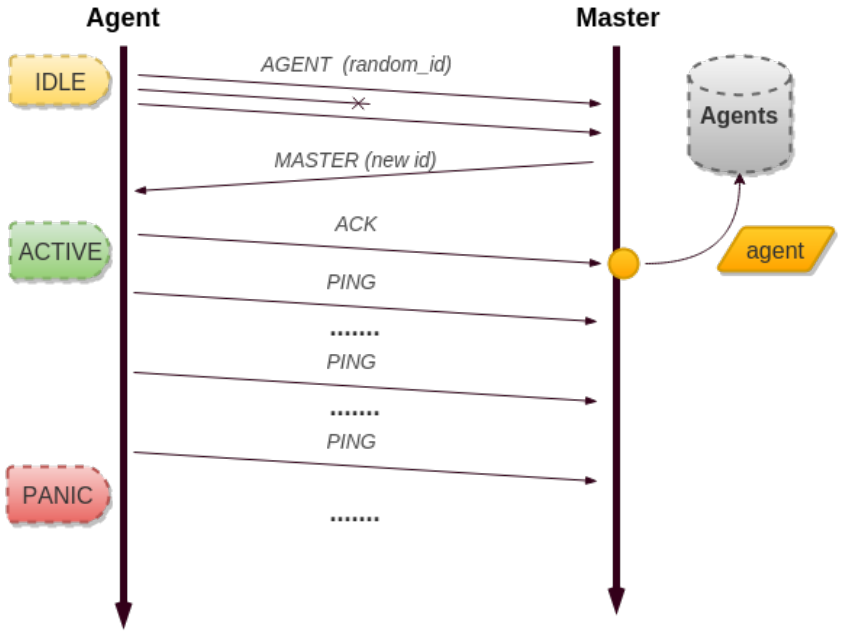
Rysunek 5.2: Diagram komunikacji dla poprawnego rozpoczęcia zachowania.

W przypadku braku aktywnego agenta, nadzorca pozostaje bezczynny. Jeśli aktywny agent nie zostanie zsynchronizowany, przechodzi on do stanu PANIC (rysunek 5.3, patrz: 5.3 Utrata połączenia z nadzorcą).

W każdym stanie oprócz stanu PANIC, z zadanym interwałem czasowym wysyłane są do nadzorcy wiadomości podtrzymujące połączenie. Agent wysyła komunikat PING, nadzorca natychmiast po jego odebraniu odsyła pakiet PONG. Jeśli oczekiwanie przekroczy zadany próg czasowy, następuje przejście do nowego stanu (rysunek 5.4).



Rysunek 5.3: Diagram komunikacji dla braku synchronizacji z nadzorcą.



Rysunek 5.4: Diagram komunikacji dla utraty połączenia po synchronizacji.

5.1.2 Oczekiwanie na aktywację

W stanie ACTIVE agent oczekuje na przydzielenie zadania. Jeśli żadna definicja zachowania nie nadchodzi, wysyłane są pakiety podtrzymujące połączenie. W przypadku braku komunikatów zwrotnych, robot przechodzi do stanu PANIC (rysunek 5.4).

Nadzorca decyduje o przydzieleniu konkretnego zachowania danemu agentowi. Wysyła do niego wiadomość typu BEHAVIOUR z parametrami zawierającymi jego typ oraz ewentualne dodatkowe argumenty. Jeśli agent zdolny jest do wykonywania danego zadania, zwraca pakiet ACK, w przeciwnym razie pakiet NOT. Komunikat ACK skutkuje odpowiedzią START, po której agent przechodzi do stanu WORKING. W przypadku komnunikatu NOT, nadzorca może zdefiniować inne zadanie lub zostawić agenta tymczasowo nieaktywnego (rysunek 5.2).

5.1.3 Przetwarzanie zachowania

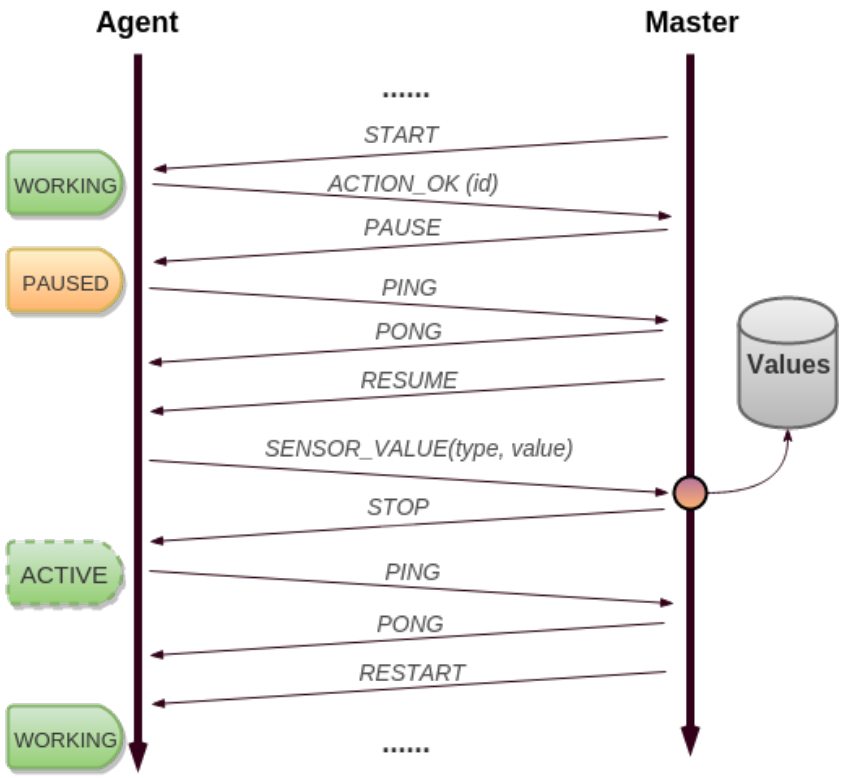
Agent może przetwarzać zlecone instrukcje tylko w stanie WORKING. Po każdej prawidłowo zakończonej akcji, wysyłana jest wiadomość typu ACTION_OK z identyfikatorem poprawnie wykonanej akcji. Pozwala to nadzorcy logować przebieg pray agentów. W przypadku zakłócenia przebiegu wykonania, robot może wysłać pakiet ACTION_INTERR, jeśli zaszło jakieś zdarzenie wymagające zmiany postępowania, a także pakiet ACTION_ERR, jeśli wystąpił błąd. W pierwszym przypadku, jeśli robot sam nie może określić następnej czynności, następuje oczekiwanie na pakiet CONTINUE. W drugim przypadku, robot bezwarunkowo zawiesza aktualną działalność i oczekuje na wiadomość RESTART, która rozpocznie przebieg bieżącego zachowania od nowa lub w przypadku wiadomości BEHAVIOUR, przebieg nowego zachowania. Niezależnie od rodzaju wiadomości wznawiającej, agent odpowiada komunikatami ACK lub NOT.

W czasie wykonywania akcji, pobierane są odczyty z sensorów. Jeśli zachowanie ma zdefiniowane konkretne sensory, których pomiary są potrzebne, wysyłane są wiadomości typu SENSOR_VALUE wraz z pomiarem (rysunek 5.5). Nadzorca interpretuje przesłane dane i w miarę możliwości zapamiętuje. Mogą one zadecydować o zmianie zachowania danego robota.

Brak odpowiedzi na wiadomości podtrzymujące lub wysłanie wiadomości typu PAUSE skutkuje przejściem do stanu PAUSED (rysunek 5.6. Otrzymanie pakietu STAND_BY powoduje przejście agenta z powrotem w stan ACTIVE.

5.1.4 Bezczynność agenta

W stanie PAUSED, robot pamięta aktualny przebieg zachowania ale nie wykonuje żadnych czynności. Dopiero po otrzymaniu komunikatu RESUME i odesłaniu odpowiedzi ACK, może powrócić do stanu WORKING i kontynuować wstrzymane zachowanie.



Rysunek 5.5: Przykładowy przebieg dla stanu WORKING.

Brak wiadomości podtrzymujących od strony nadzorcy powoduje przejście do stanu PANIC (rysunek 5.7).

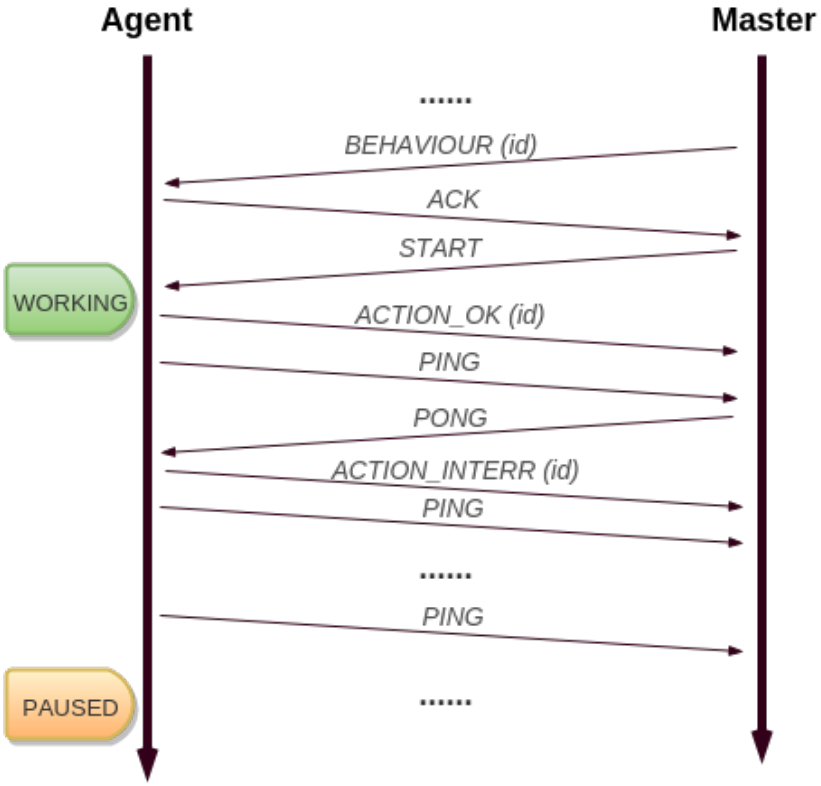
5.2 Komunikacja wewnątrz jednostki

W obrębie jednego agenta, wyróżnia się dwie podstawowe formy komunikacji: wiadomości i zdarzenia.

5.2.1 Wiadomości

Przy uruchomieniu programu, tworzony jest dodatkowy wątek komunikacyjny, który jest zsynchronizowany z wątkiem głównym robota (lub nadzorcy). Wymiana informacji odbywa się za pomocą dwóch, przeciwnie przydzielonych obiektów klasy szablonowej `Queue`, która zawiera interfejs obsługi kolejki wiadomości wraz z operacjami jednoczesnego dostępu wielu wątków. Kolejka wiadomości odbieranych przez klasę `Robot` jest kolejką wiadomości wysyłanych przez klasę `Communication` i odwrotnie w przypadku drugiego obiektu.

Po stronie komunikacji odbywa się jedynie przesyłanie i odbieranie pakietów między fizycznymi urządzeniami oraz sprawdzanie duplikatów. Interpretacją zaj-



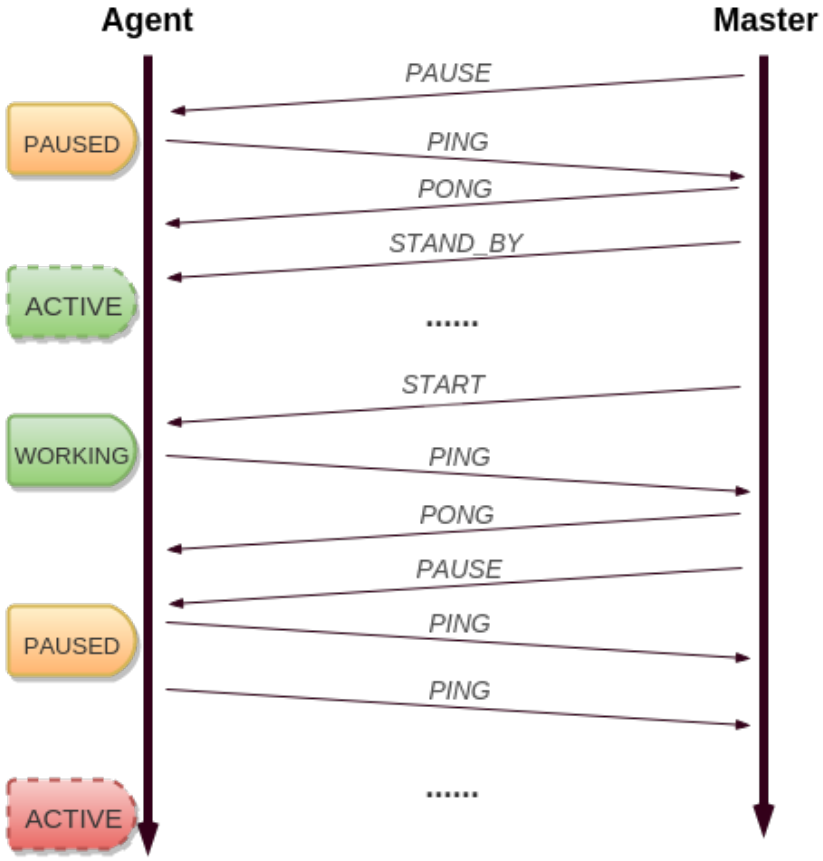
Rysunek 5.6: Brak komunikacji w stanie WORKING.

muje się wątek główny. Wszystkie możliwe do wysłania pakiety są zapisywane w klasie `Message`, w której skład wchodzą następujące elementy:

- Identyfikator nadawcy, przydzielany automatycznie
- Identyfikator odbiorcy, stały i równy 1 w przypadku, gdy odbiorcą jest nadzorca. W przeciwnym razie wybierany spośród agentów z bazy
- Identyfikator wiadomości, sukcesywnie inkrementowany
- Rodzaj wiadomości, bedący wartością typu wyliczeniowego, czyli tak naprawdę liczbą
- (opcjonalnie) Dodatkowe parametry w zależniości od typu wiadomości.

Przed wysłaniem, wiadomość będąca obiektem zamieniana jest na łańcuch znaków zawierający jej prototyp. Przykładowy prototyp może wyglądać następująco:

`14:1:254:5:foo:bar`,



Rysunek 5.7: Diagram komunikacji dla stany pauzy.

gdzie wartości oddzielone separatorem (domyślnie dwukropek) oznaczają kolejno: identyfikator agenta, identyfikator odbiorcy (nadzorcy), numer wiadomości, typ wiadomości oraz dodatkowe parametry.

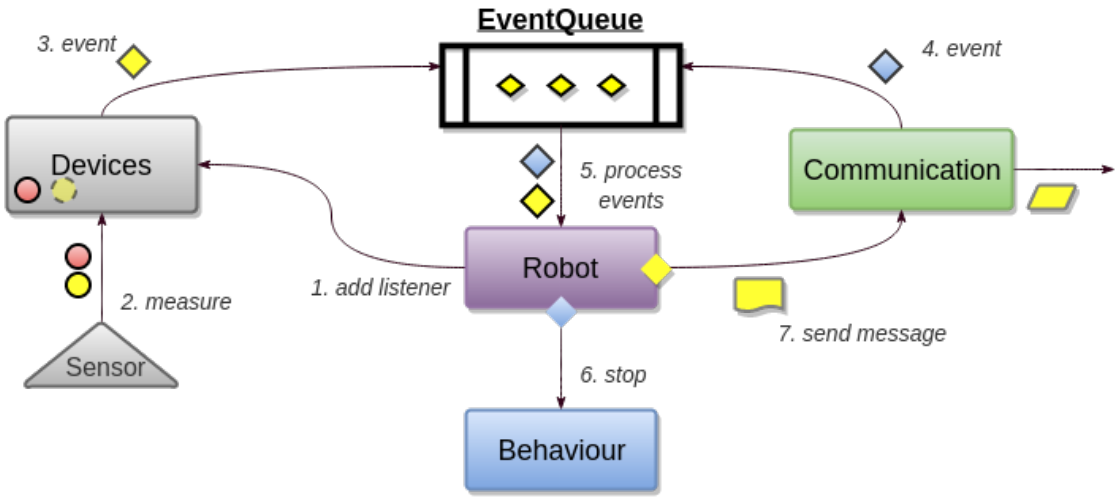
Przy odbieraniu kolejnych wiadomości, w buforze cyklicznym zapisywane są ich prototypy. Nowe wiadomości zastępują stare, jeśli bufor jest w całości zapełniony. Przed dodaniem prototypu do bufora, następuje sprawdzenie, czy nie jest on już w nim zapisany. Jeśli tak, oznacza to odebranie duplikatu i odrzucenie wiadomości. Rozwiązanie to jest konieczne w przypadku wielu agentów, gdyż kopie wysyłanych przez nich wiadomości mogą przychodzić naprzemiennie.

Odebrany i zakwalifikowany do przyjęcia prototyp zostaje zdekodowany do postaci obiektu klasy `Message` i przesłany do wątku głównego w celu interpretacji. Pakiety, które zawierają niezaktualizowane identyfikatory wiadomości lub błędne informacje o nadawcy są odrzucane. Ma to miejsce szczególnie w przypadku pakietów wysyłanych na adres rozgłoszeniowy, np. podczas synchronizacji agenta z nadzorcą.

5.2.2 Zdarzenia

W celu usprawnienia komunikacji oraz zwiększenia czytelności całej aplikacji, konkretne warstwy i moduły komunikują się z klasą `Robot` za pomocą zdarzeń opisanych klasą `Event`. Zaimplementowana kolejka (`EventQueue`) przechowująca zdarzenia jest wspólna dla całej instancji aplikacji. Generować nowe zdarzenia mogą klasy zachowań, akcji czy urządzeń, a jedynym punktem wyjścia kolejki jest główna klasa robota (rysunek 3.4), który interpretuje zdarzenie i reaguje na nie.

Rozróżniane są zdarzenia zwykłe oraz krytyczne, które przerywają wykonywanie części bądź wszystkich działań. Do takich zdarzeń należą np. odłączenie urządzenia czy niski poziom baterii robota. Pozostałe zdarzenia mogą przyjmować formę czysto informacyjną (np. poprawnie podłączone urządzenia), ostrzegającą (np. wykryto zderzenie) lub zgłaszającą wystąpienie wyjątku. Taka forma wewnętrznej komunikacji zdejmuje konieczność przekazywania zdarzeń w argumentach wielu funkcji lub pamiętania bezpośrednich referencji do obiektów generujących zdarzenia oraz pozwala łatwo koordynować cały cykl życia obiektu `Event` z poziomu jednej kolejki. Ponadto, dodanie lub usunięcie nowego obiektu generującego bądź nasłuchującego jest proste i nie ingeruje zbyt mocno w istniejącą strukturę komunikacji (rysunek 5.8).



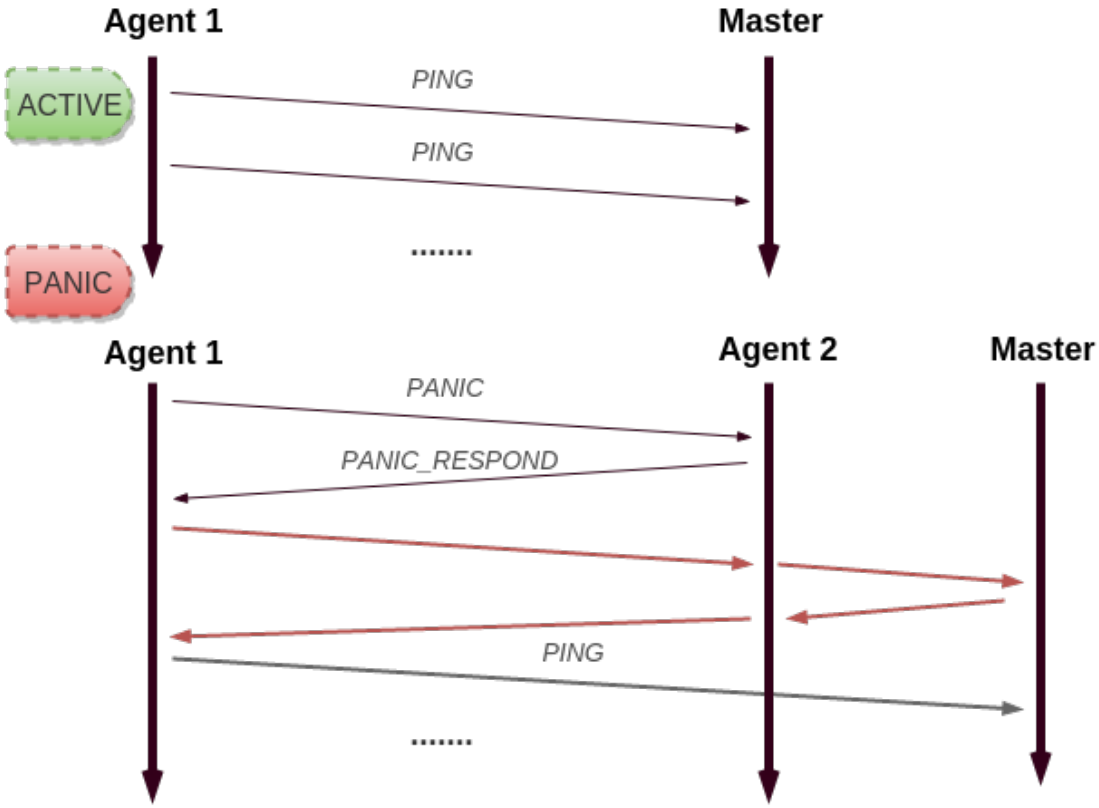
Rysunek 5.8: Przykładowy diagram komunikacji za pomocą zdarzeń.

5.3 Utrata połączenia z nadzorcą

Zdarzenie to może zajść w dwóch wariantach:

5.3.1 Jeden agent traci połączenie z nadzorcą

Robot przechodzi do stanu PANIC i przez adres rozgłoszeniowy informuje pozostałych agentów o utraconym połączeniu. Jeden z nich może odpowiedzieć pozytywnie, wysyłając potwierdzenie posiadania komunikacji z nadzorcą. Wtedy następuje synchronizacja dwóch agentów i ustalenie przekierowania wiadomości przez robota pośredniego (rysunek 5.9). Jeżeli po pewnym czasie, nikt nie odpowie pozytywnie oraz nie zacznie się głosowanie opisane w punkcie poniżej, robot kończy swoje działanie.



Rysunek 5.9: Przekierowanie wiadomości do nadzorcy przez drugiego agenta.

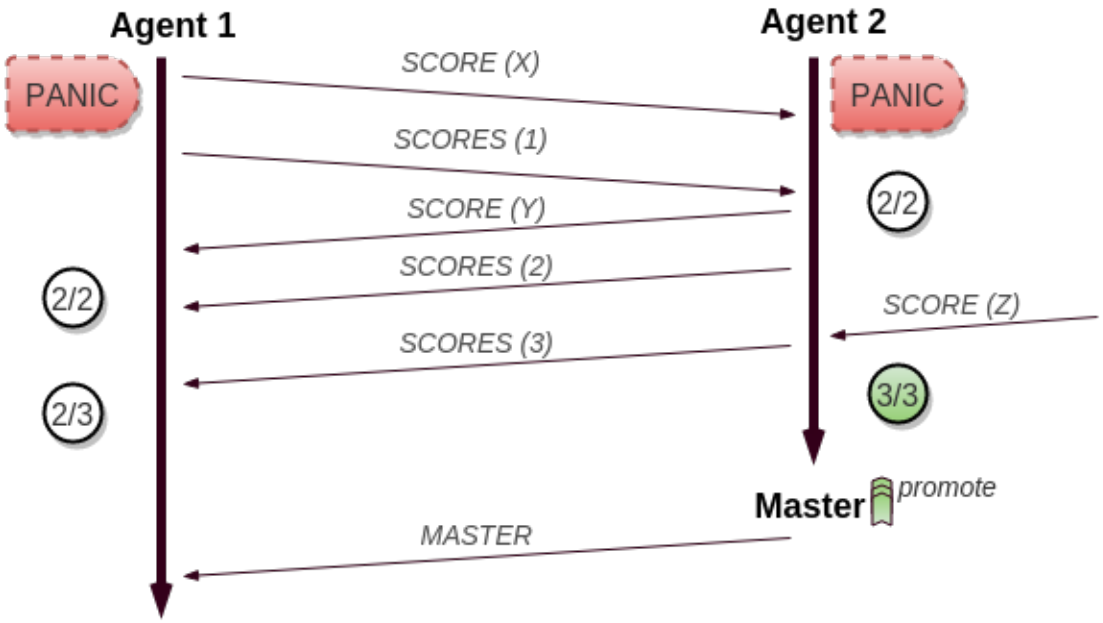
5.3.2 Wszyscy agenci tracą połączenie z nadzorcą

Wszystkie roboty utraciły połączenie i żaden nie otrzymał odpowiedzi na zapytanie o przekierowanie. W takim przypadku odbywa się głosowanie. Każdy agent wysyła na adres rozgłoszeniowy swój osobisty wynik[1]. Po pewnym czasie, kiedy

[1]Wynik agenta zależy m. in. od ilości przesłanych danych, poprawności wykonanych akcji czy czasu aktywności.

wszystkie roboty znają już swoje wyniki, ten z najniższym rezultatem mianuje siebie nadzorcą. W przypadku tych samych wyników, o wyborze decyduje czynnik losowy. Wybrany zostaje najsłabszy rezultat, ponieważ system zyska więcej, jesli aktywni pozostaną agenci najlepsi.

Problem jaki może napotkać grupa robotów, to jednoczesny wybór dwóch nadzorców. Ograniczenie użycia jednej wspólnej sieci nie eliminuje problemu gubienia pakietów, a przez to potencjalnej sytuacji, w której mamy wiele niezależnych głosowań, bądź głosowania są niepełne. W tym celu, oprócz wysłania przez danego agenta swoich danych, wysyła on dodatkowo liczbę oznaczającą ilość otrzymanych rekordów. W ten sposób agent A może dowiedzieć się od agenta B, że nie dostał informacji od innego agenta, np. C, kiedy połączenie między A i C jest niemożliwe. Ponadto, własny wynik jest ponownie rozsyłany za każdym zwiększeniem ilości informacji o pozostałych agentach (rysunek 5.10). Jednostki nie posiadające kompletu informacji nie biorą udziału w finalnym głosowaniu, ponieważ mogą podjąć złą decyzję, a ponadto utrudniona komunikacja jest dla nich dodatkowo niekorzystna.



Rysunek 5.10: Głosowanie i wybór nowego nadzorcy.

Rozdział 6

Testowanie aplikacji

6.1 Testy poprawności

6.1.1 Testowanie pojedynczych akcji

Ruch z zadaną prędkością - wykonanie akcji powoduje utrzymywanie stałej prędkości obu motorów przez czas nieograniczony

Zatrzymywanie - robot wprawiony w ruch zatrzymuje się niemal natychmiastowo. W zależności od sposobu zatrzymywania się, czas po którym prędkość jest równa zeru, a co za tym idzie, dystans hamowania, zmienia się. W przypadku trybu `coast` jest on największy i równy od 1 do 5 cm w zależności od szybkości jazdy. Zaś w przypadku trybu `hold`, koła zatrzymują się natychmiast i dystans maleje do maksymalnie 2 cm na śliskiej powierzchni

Ruch na zadaną odległość - akcja wykonywana poprawnie z dokładnością do kilku nadmiarowych centymetrów. Nadwyżka odległości zależy od ustalonego trybu zatrzymywania oraz rodzaju nawierzchni, a wynika z opóźnienia odczytów z efektorów. Na podstawie zmierzonego promienia koła, zadanej odległości oraz ilości jednostek w jednym pełnym obrocie, można wyliczyć docelową pozycję motoru i zatrzymać go, gdy ta zostanie osiągnięta

Obrót - jest wykonywany wokół osi równo oddalonej od obu kół. Podobnie jak w przypadku ruchu, sprawdzana jest zmiana położenia motoru, której docelowa wartość wyliczana jest na podstawie rozstawu kół i kąta. Gdy jeden efektor ma zadaną wartość dodatnią prędkości, drugi ma wartość przeciwną. Wykryto odchylenie o kilka stopni przy większych prędkościach. Zalecana jest zatem mała prędkość obrotu, w celu uzyskania większej precyzji.

6.1.2 Testowanie obsługi zdarzeń

Rozpoczęcie przetwarzania zachowania

Warunki: Robot implementuje wymagane przez zachowanie akcje, posiada urządzenia umożliwiające wykonanie tych akcji oraz jest w stanie ACTIVE, WORKING lub PAUSED

Przypadek 1 - Brak wymaganych urządzeń: Robot zgłasza błąd i przerywa działanie

```
[WARNING] Missing device: lego-ev3-l-motor at outB
[VERBOSE] Device connected: lego-ev3-l-motor at outC
[VERBOSE] Device connected: lego-ev3-touch at in2
[VERBOSE] Device connected: lego-ev3-us at in1
[ ERROR ] Devices incorrect!
[ INFO  ] Robot exiting...
[ INFO  ] Comm. exiting...
[ INFO  ] Exiting...
```

Przypadek 2 - Robot nie implementuje akcji: Zgłoszenie ostrzeżenia (domyślnie wysyłany jest komunikat zwrotny o niepowodzeniu przypisania zachowania)

```
[VERBOSE] Device connected: lego-ev3-l-motor at outB
[VERBOSE] Device connected: lego-ev3-l-motor at outC
[VERBOSE] Device connected: lego-ev3-touch at in2
[VERBOSE] Device connected: lego-ev3-us at in1
[ INFO  ] Devices correct.
[VERBOSE] A >> .. Sender:  496324 | Receiver:     1 |        MsgId.:  0 |
          Type:      AGENT | Params:
[VERBOSE] A >> .. Sender:       0 | Receiver:     1 |        MsgId.:  1 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  0 |
          Type:     MASTER | Params:
[ INFO  ] New ID aqcuired: 3
[ INFO  ] Changing state to ACTIVE
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  2 |
          Type:        ACK | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  2 |
          Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
[VERBOSE] Obtaining Behaviour parameters...
[ INFO  ] Loading predefined Behaviour: 1
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  3 |
          Type:        NOT | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  4 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  4 |
          Type:       PONG | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  5 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  5 |
          Type:       PONG | Params:
```

Przypadek 3 - Nieprawidłowy stan: Brak działania (domyślnie dostarczono zły komunikat, więc jest on ignorowany)

```
........
[ INFO  ] Devices correct.
```

```
3  [VERBOSE] A >> .. Sender:  496324 | Receiver:     1 |        MsgId.:  0 |
             Type:       AGENT | Params:
4  [VERBOSE] A >> .. Sender:       0 | Receiver:     1 |        MsgId.:  1 |
             Type:        PING | Params:
5  [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  1 |
             Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
6  ### IGNORED
7  [VERBOSE] A >> .. Sender:  894236 | Receiver:     1 |        MsgId.:  2 |
             Type:       AGENT | Params:
8  [VERBOSE] A >> .. Sender:  764523 | Receiver:     1 |        MsgId.:  3 |
             Type:       AGENT | Params:
```

Przypadek 4 - Warunki spełnione: Następuje przypisanie zachowania, a następnie wygenerowanie zdarzenia BEHAVIOUR_START, które przechwycone przez klasę `Robot` uruchamia je.

```
1  ........
2  [ INFO  ] Devices correct.
3  [VERBOSE] A >> .. Sender:   93538 | Receiver:     1 |        MsgId.:  0 |
             Type:       AGENT | Params:
4  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  0 |
             Type:      MASTER | Params:
5  [ INFO  ] New ID aqcuired: 2
6  [ INFO  ] Changing state to ACTIVE
7  [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  3 |
             Type:         ACK | Params:
8  [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  4 |
             Type:        PING | Params:
9  [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  3 |
             Type:  BEHAVIOUR | Params: 2
10 [VERBOSE] Obtaining Behaviour parameters...
11 [ INFO  ] Loading predefined Behaviour: 1
12 [ INFO  ] <Robot model A> Explore random
13 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  5 |
             Type:         ACK | Params:
14 [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  4 |
             Type:        PONG | Params:
15 [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  5 |
             Type:       START | Params:
16 [VERBOSE] Generated event: BEHAVIOUR_START
17 [ INFO  ] Changing state to WORKING
18 [VERBOSE] Behaviour start.
```

Ostrzeżenie o przeszkodzie

Warunki: Robot posiada sensor nacisku oraz zachowanie ma zdefiniowaną reakcję na zdarzenie OBSTACLE_DETECTED

Przypadek 1 - Brak sensora: Jeśli robot powinien go posiadać, zgłaszany jest błąd. Jeśli nie - to zdarzenie nie będzie generowane

```
1  [VERBOSE] Device connected: lego-ev3-l-motor at outB
2  [VERBOSE] Device connected: lego-ev3-l-motor at outC
3  [WARNING] Missing device: lego-ev3-touch at in2
4  [VERBOSE] Device connected: lego-ev3-us at in1
5  [ ERROR ] Devices incorrect!
6  [ INFO  ] Robot exiting...
7  [ INFO  ] Comm. exiting...
8  [ INFO  ] Exiting...
```

Przypadek 2 - Brak zdefiniowanej reakcji: Zdarzenie jest wygenerowane, ale nie podjęte są żadne działania

```
1  .......
2  [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.:  1 |
               Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
3  [VERBOSE] Obtaining Behaviour parameters...
4  [ INFO  ] Loading predefined Behaviour: 1
5  [ INFO  ] <Robot model A> Drive on square: side length (1000), turning
       right(1)
6  [VERBOSE] A >> .. Sender:       4 | Receiver:     1 |        MsgId.:  3 |
               Type:         ACK | Params:
7  [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.:  2 |
               Type:        PONG | Params:
8  [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.:  3 |
               Type:       START | Params:
9  [VERBOSE] Generated event: BEHAVIOUR_START
10 [ INFO  ] Changing state to WORKING
11 [VERBOSE] Behaviour start.
12 [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.:  0 |
               Type:    MEASURE | Params: 2 | 0 |
13 [VERBOSE] Generated event: SENSOR_WATCH
14 [VERBOSE] A >> .. Sender:       4 | Receiver:     1 |        MsgId.:  4 |
               Type: SENSOR_VAL | Params: 2 | 2550 | 1 |
15 [VERBOSE] A >> .. Sender:       4 | Receiver:     1 |        MsgId.:  5 |
               Type:        PING | Params:
16 [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.:  5 |
               Type:        PONG | Params:
17 [VERBOSE] Generated event: OBSTACLE_DETECTED
18 [WARNING] Reaction not found.
```

Przypadek 3 - Warunki spełnione: Zgłoszone zdarzenie jest przesłane do zachowania i następuje natychmiastowe zatrzymanie robota, a następnie przejście w stan pośredni oraz kontynuowanie zachowania w kolejnym domyślnym stanie.

```
1  .......
2  [VERBOSE] .. << M Sender:       1 | Receiver:     5 |        MsgId.:  1 |
               Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
3  [VERBOSE] Obtaining Behaviour parameters...
4  [ INFO  ] Loading predefined Behaviour: 1
5  [1a0414
6  [ INFO  ] <Robot model A> Drive on square: side length (1000), turning
       right(1)
7  [VERBOSE] A >> .. Sender:       5 | Receiver:     1 |        MsgId.:  3 |
               Type:         ACK | Params:
8  [VERBOSE] .. << M Sender:       1 | Receiver:     5 |        MsgId.:  2 |
               Type:        PONG | Params:
9  [VERBOSE] .. << M Sender:       1 | Receiver:     5 |        MsgId.:  3 |
               Type:       START | Params:
10 [VERBOSE] Generated event: BEHAVIOUR_START
11 [ INFO  ] Changing state to WORKING
12 [VERBOSE] Behaviour start.
13 [VERBOSE] .. << M Sender:       1 | Receiver:     5 |        MsgId.:  0 |
               Type:    MEASURE | Params: 2 | 0 |
14 [VERBOSE] Generated event: SENSOR_WATCH
15 [VERBOSE] A >> .. Sender:       5 | Receiver:     1 |        MsgId.:  4 |
               Type: SENSOR_VAL | Params: 2 | 2550 | 1 |
16 [VERBOSE] Generated event: OBSTACLE_DETECTED
17 [ DEBUG ] Reaction performed.
```

Ostrzeżenie o bliskim obiekcie

Warunki: Robot posiada sensor odległości oraz zachowanie ma zdefiniowaną reakcję na zdarzenie PROXIMITY_ALERT

Przypadek 1 - Brak sensora: Jeśli robot powinien go posiadać, zgłaszany jest błąd. Jeśli nie - to zdarzenie nie będzie generowane

```
1  [VERBOSE] Device connected: lego-ev3-l-motor at outB
2  [VERBOSE] Device connected: lego-ev3-l-motor at outC
3  [VERBOSE] Device connected: lego-ev3-touch at in2
4  [WARNING] Missing device: lego-ev3-us at in1
5  [ ERROR ] Devices incorrect!
6  [ INFO  ] Robot exiting...
7  [ INFO  ] Comm. exiting...
8  [ INFO  ] Exiting...
```

Przypadek 2 - Brak zdefiniowanej reakcji: Zdarzenie jest wygenerowane, ale nie podjęte są żadne działania

```
1  .......
2  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  2 |
             Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
3  [VERBOSE] Obtaining Behaviour parameters...
4  [ INFO  ] Loading predefined Behaviour: 1
5  [ INFO  ] <Robot model A> Explore
6  [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  3 |
             Type:         ACK | Params:
7  [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  4 |
             Type:        PING | Params:
8  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  3 |
             Type:       START | Params:
9  [VERBOSE] Generated event: BEHAVIOUR_START
10 [ INFO  ] Changing state to WORKING
11 [VERBOSE] Behaviour start.
12 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  4 |
             Type:        PONG | Params:
13 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  4 |
             Type:     MEASURE | Params: 2 | 0 |
14 [VERBOSE] Generated event: SENSOR_WATCH
15 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  5 |
             Type: SENSOR_VAL | Params: 2 | 2266 | 1 |
16 [VERBOSE] Generated event: PROXIMITY_ALERT
17 [WARNING] Reaction not found.
```

Przypadek 3 - Warunki spełnione: Zgłoszone zdarzenie jest przesłane do zachowania i następuje natychmiastowe zatrzymanie robota, a następnie przejście w stan pośredni oraz kontynuowanie zachowania w kolejnym domyślnym stanie.

```
1  .......
2  [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  2 |
             Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
3  [VERBOSE] Obtaining Behaviour parameters...
4  [ INFO  ] Loading predefined Behaviour: 1
5  [ INFO  ] <Robot model A> Drive on square: side length (1000), turning
       right(1)
```

```
6  [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  3 |
             Type:        ACK | Params:
7  [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  3 |
             Type:      START | Params:
8  [VERBOSE] Generated event: BEHAVIOUR_START
9  [ INFO  ] Changing state to WORKING
10 [VERBOSE] Behaviour start.
11 [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  0 |
             Type:    MEASURE | Params: 2 | 0 |
12 [VERBOSE] Generated event: SENSOR_WATCH
13 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  4 |
             Type: SENSOR_VAL | Params: 2 | 1799 | 1 |
14 [VERBOSE] Generated event: PROXIMITY_ALERT
15 [WARNING] Reaction not found.
16 [VERBOSE] Generated event: SENSOR_WATCH
17 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  9 |
             Type: SENSOR_VAL | Params: 2 | 72 | 1 |
18 [VERBOSE] Generated event: PROXIMITY_ALERT
19 [ DEBUG ] Reaction performed.
```

Niski poziom baterii

Przetestowany dla poziomu większego niż minimalne dopuszczalne napięcie. Po jego przekroczeniu, wygenerowane zostało zdarzenie BATTERY_LOW, wysłano komunikat o niskim naładowaniu baterii oraz nastąpiło zamknięcie aplikacji.

6.1.3 Testowanie przejść między stanami

Komunikaty niezarządzające przetwarzaniem danego stanu są ignorowane.

Stan IDLE

Początek stanu - Dioda miga na żółto

Brak odpowiedzi od nadzorcy - Po ustalonym czasie, przejście w stan PANIC

```
1  [VERBOSE] Device connected: lego-ev3-l-motor at outB
2  [VERBOSE] Device connected: lego-ev3-l-motor at outC
3  [VERBOSE] Device connected: lego-ev3-touch at in2
4  [VERBOSE] Device connected: lego-ev3-us at in1
5  [ INFO  ] Devices correct.
6  [VERBOSE] A >> .. Sender:  933804 | Receiver:     1 |        MsgId.:  0 |
             Type:      AGENT | Params:
7  [VERBOSE] A >> .. Sender:       0 | Receiver:     1 |        MsgId.:  1 |
             Type:       PING | Params:
8  [VERBOSE] A >> .. Sender:  490219 | Receiver:     1 |        MsgId.:  2 |
             Type:      AGENT | Params:
9  [VERBOSE] A >> .. Sender:       0 | Receiver:     1 |        MsgId.:  3 |
             Type:       PING | Params:
10 [VERBOSE] A >> .. Sender:  463753 | Receiver:     1 |        MsgId.:  4 |
             Type:      AGENT | Params:
11 [VERBOSE] A >> .. Sender:       0 | Receiver:     1 |        MsgId.:  5 |
             Type:       PING | Params:
12 [ INFO  ] Changing state to PANIC
```

Synchronizacja z nadzorcą - Nadanie identyfikatora agentowi i przejście w stan ACTIVE

```
[VERBOSE] Device connected: lego-ev3-l-motor at outB
[VERBOSE] Device connected: lego-ev3-l-motor at outC
[VERBOSE] Device connected: lego-ev3-touch at in2
[VERBOSE] Device connected: lego-ev3-us at in1
[ INFO  ] Devices correct.
[VERBOSE] A >> .. Sender:  799110 | Receiver:     1 |        MsgId.:  0 |
          Type:      AGENT | Params:
[VERBOSE] A >> .. Sender:       0 | Receiver:     1 |        MsgId.:  1 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  0 |
          Type:     MASTER | Params:
[ INFO  ] New ID aqcuired: 2
[ INFO  ] Changing state to ACTIVE
[VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  2 |
          Type:        ACK | Params:
```

Stan ACTIVE

Początek stanu - Dioda miga na zielono

Brak odpowiedzi od nadzorcy - Po ustalonym czasie, przejście w stan PANIC

```
........
[ INFO  ] New ID aqcuired: 3
[ INFO  ] Changing state to ACTIVE
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  2 |
          Type:        ACK | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  2 |
          Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
[VERBOSE] Obtaining Behaviour parameters...
[ INFO  ] Loading predefined Behaviour: 1
[ INFO  ] <Robot model A> Drive on square: side length (1000), turning
    right(1)
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  3 |
          Type:        ACK | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  4 |
          Type:       PING | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  5 |
          Type:       PING | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  6 |
          Type:       PING | Params:
[ INFO  ] Changing state to PANIC
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  7 |
          Type:       PING | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  8 |
          Type:       PING | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  9 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       0 | Receiver:     0 |        MsgId.:  0 |
          Type: AGENT_OVER | Params:
[ INFO  ] Robot exiting...
[ INFO  ] Comm. exiting...
[ INFO  ] Exiting...
```

Przesłanie zachowania i komunikatu START - Przypisanie zachowania, przejście w stan WORKING, rozpoczęcie przetwarzania zachowania.

```
.......
[ INFO  ] New ID aqcuired: 2
[ INFO  ] Changing state to ACTIVE
```

```
4  [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  1 |
              Type:        ACK | Params:
5  [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  2 |
              Type:       PING | Params:
6  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  1 |
              Type:  BEHAVIOUR | Params: 1 | 1000 | 1 |
7  [VERBOSE] Obtaining Behaviour parameters...
8  [ INFO  ] Loading predefined Behaviour: 1
9  [ INFO  ] <Robot model A> Drive on square: side length (1000), turning
      right(1)
10 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  3 |
              Type:        ACK | Params:
11 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  2 |
              Type:       PONG | Params:
12 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  3 |
              Type:      START | Params:
13 [VERBOSE] Generated event: BEHAVIOUR_START
14 [ INFO  ] Changing state to WORKING
15 [VERBOSE] Behaviour start.
16 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  0 |
              Type:    MEASURE | Params: 2 | 0 |
17 [VERBOSE] Generated event: SENSOR_WATCH
18 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  4 |
              Type: SENSOR_VAL | Params: 2 | 1783 | 1 |
```

Stan WORKING

Początek stanu - Dioda świeci stale na zielono

Brak odpowiedzi od nadzorcy - Po ustalonym czasie, przejście w stan PAUSED oraz zatrzymanie zachowania

```
1  .......
2  [ INFO  ] Changing state to WORKING
3  [VERBOSE] Behaviour start.
4  [VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  0 |
              Type:    MEASURE | Params: 2 | 0 |
5  ........
6  [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 17 |
              Type: SENSOR_VAL | Params: 2 | 321 | 1 |
7  [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 18 |
              Type:       PING | Params:
8  [VERBOSE] Generated event: SENSOR_WATCH
9  [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 19 |
              Type: SENSOR_VAL | Params: 2 | 372 | 1 |
10 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 20 |
              Type:       PING | Params:
11 [VERBOSE] Generated event: SENSOR_WATCH
12 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 21 |
              Type: SENSOR_VAL | Params: 2 | 908 | 1 |
13 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 22 |
              Type:       PING | Params:
14 [VERBOSE] Generated event: SENSOR_WATCH
15 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 23 |
              Type: SENSOR_VAL | Params: 2 | 370 | 1 |
16 [VERBOSE] Generated event: SENSOR_WATCH
17 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 24 |
              Type: SENSOR_VAL | Params: 2 | 2387 | 1 |
18 [VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 25 |
              Type:       PING | Params:
19 [ INFO  ] Changing state to PAUSED
```

Przesłanie komunikatu STOP - Zatrzymanie zachowania, przejście do stanu ACTIVE

```
.......
[ INFO  ] Changing state to WORKING
[VERBOSE] Behaviour start.
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  3 |
          Type:    MEASURE | Params: 2 | 0 |
[VERBOSE] Generated event: SENSOR_WATCH
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  4 |
          Type: SENSOR_VAL | Params: 2 | 1054 | 1 |
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  5 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  5 |
          Type:       PONG | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  5 |
          Type:       STOP | Params:
[VERBOSE] Generated event: BEHAVIOUR_STOP
[ DEBUG ] Behaviour stop.
[ INFO  ] Changing state to ACTIVE
```

Przesłanie komunikatu PAUSE - Zatrzymanie zachowania, przejście do stanu PAUSED

```
.......
[ INFO  ] Changing state to WORKING
[VERBOSE] Behaviour start.
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  5 |
          Type:    MEASURE | Params: 2 | 0 |
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  6 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  6 |
          Type:       PONG | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     3 |        MsgId.:  7 |
          Type:      PAUSE | Params:
[VERBOSE] Generated event: SENSOR_WATCH
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.:  7 |
          Type: SENSOR_VAL | Params: 2 | 578 | 1 |
[VERBOSE] Generated event: BEHAVIOUR_PAUSE
[ INFO  ] Changing state to PAUSED
```

Stan PAUSED

Początek stanu - Dioda świeci stale na pomarańczowo

Brak odpowiedzi od nadzorcy - Po ustalonym czasie, przejście w stan PANIC

```
........
[ INFO  ] Changing state to PAUSED
[VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 19 |
          Type:       PING | Params:
[VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 19 |
          Type:       PONG | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 20 |
          Type:       PING | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 21 |
          Type:       PING | Params:
[VERBOSE] A >> .. Sender:       3 | Receiver:     1 |        MsgId.: 22 |
          Type:       PING | Params:
[ INFO  ] Changing state to PANIC
```

Przesłanie komunikatu RESUME - Wznowienie zachowania, przejście do stanu WORKING

```
1  .......
2  [ INFO  ] Changing state to PAUSED
3  [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 17 |
                Type:       PING | Params:
4  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 17 |
                Type:       PONG | Params:
5  [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 18 |
                Type:       PING | Params:
6  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 18 |
                Type:     RESUME | Params:
7  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 18 |
                Type:       PONG | Params:
8  [VERBOSE] Generated event: BEHAVIOUR_RESUME
9  [ INFO  ] Changing state to WORKING
```

Przesłanie komunikatu STOP - Zatrzymanie zachowania, przejście do stanu ACTIVE

```
1  .......
2  [ INFO  ] Changing state to PAUSED
3  [VERBOSE] A >> .. Sender:       4 | Receiver:     1 |        MsgId.: 36 |
                Type:       PING | Params:
4  [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.: 36 |
                Type:       PONG | Params:
5  [VERBOSE] A >> .. Sender:       4 | Receiver:     1 |        MsgId.: 37 |
                Type:       PING | Params:
6  [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.: 37 |
                Type:       STOP | Params:
7  [VERBOSE] .. << M Sender:       1 | Receiver:     4 |        MsgId.: 37 |
                Type:       PONG | Params:
8  [VERBOSE] Generated event: BEHAVIOUR_STOP
9  [ INFO  ] Changing state to ACTIVE
```

6.2 Testy funkcjonalne

6.2.1 Testowanie zachowania złożonego i komunikacji

Na poniższym listingu przedstawiono jeden pełny przebieg zachowania wraz z komunikatami komunikacyjnymi i przejściami między stanami.

```
1  [VERBOSE] Device connected: lego-ev3-l-motor at outB
2  [VERBOSE] Device connected: lego-ev3-l-motor at outC
3  [VERBOSE] Device connected: lego-ev3-touch at in2
4  [VERBOSE] Device connected: lego-ev3-us at in1
5  [ INFO  ] Devices correct.
6  [VERBOSE] A >> .. Sender:  666081 | Receiver:     1 |        MsgId.:  0 |
                Type:      AGENT | Params:
7  [VERBOSE] .. << M Sender:  666081 | Receiver:     1 |        MsgId.:  0 |
                Type:      AGENT | Params:
8  [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  0 |
                Type:     MASTER | Params:
9  [ INFO  ] New ID aqcuired: 2
10 [ INFO  ] Changing state to ACTIVE
11 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  1 |
                Type:        ACK | Params:
```

```
12 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  2 |
                Type:       PING | Params:
13 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  1 |
                Type:  BEHAVIOUR | Params: 2 |
14 [VERBOSE] Obtaining Behaviour parameters...
15 [ INFO  ] Loading predefined Behaviour: 2
16 [ INFO  ] <Robot model A> Explore random
17 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  3 |
                Type:        ACK | Params:
18 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  2 |
                Type:       PONG | Params:
19 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  3 |
                Type:      START | Params:
20 [VERBOSE] Generated event: BEHAVIOUR_START
21 [ INFO  ] Changing state to WORKING
22 [VERBOSE] Behaviour start.
23 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  0 |
                Type:    MEASURE | Params: 2 | 0 |
24 [VERBOSE] Generated event: ACTION_FINISHED
25 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  4 |
                Type:  ACTION_OK | Params: 5 |
26 [VERBOSE] Generated event: SENSOR_WATCH
27 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  5 |
                Type: SENSOR_VAL | Params: 2 | 1638 | 1 |
28 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  6 |
                Type:       PING | Params:
29 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.:  6 |
                Type:       PONG | Params:
30 [VERBOSE] Generated event: PROXIMITY_ALERT
31 [ DEBUG ] Reaction performed.
32 [VERBOSE] Generated event: ACTION_FINISHED
33 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  7 |
                Type:  ACTION_OK | Params: 6 |
34 [VERBOSE] Generated event: PROXIMITY_ALERT
35 [WARNING] Reaction not found.
36 [VERBOSE] Generated event: ACTION_FINISHED
37 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  8 |
                Type:  ACTION_OK | Params: 5 |
38 [VERBOSE] Generated event: PROXIMITY_ALERT
39 [WARNING] Reaction not found.
40 [VERBOSE] Generated event: ACTION_FINISHED
41 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.:  9 |
                Type:  ACTION_OK | Params: 5 |
42 [VERBOSE] Generated event: SENSOR_WATCH
43 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 10 |
                Type: SENSOR_VAL | Params: 2 | 195 | 1 |
44 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 11 |
                Type:       PING | Params:
45 [VERBOSE] Generated event: PROXIMITY_ALERT
46 [VERBOSE] Generated event: ACTION_FINISHED
47 [WARNING] Reaction not found.
48 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 12 |
                Type:  ACTION_OK | Params: 4 |
49 [VERBOSE] Generated event: SENSOR_WATCH
50 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 13 |
                Type: SENSOR_VAL | Params: 2 | 908 | 1 |
51 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 14 |
                Type:       PING | Params:
52 [VERBOSE] Generated event: SENSOR_WATCH
53 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 15 |
                Type: SENSOR_VAL | Params: 2 | 370 | 1 |
54 [VERBOSE] Generated event: SENSOR_WATCH
55 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 16 |
                Type: SENSOR_VAL | Params: 2 | 2387 | 1 |
```

```
56 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 17 |
                Type:       PING | Params:
57 [ INFO  ] Changing state to PAUSED
58 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 18 |
                Type:       PING | Params:
59 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 18 |
                Type:       PONG | Params:
60 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 18 |
                Type:     RESUME | Params:
61 [VERBOSE] Generated event: BEHAVIOUR_RESUME
62 [ INFO  ] Changing state to WORKING
63 [VERBOSE] Generated event: SENSOR_WATCH
64 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 19 |
                Type: SENSOR_VAL | Params: 2 | 195 | 1 |
65 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 20 |
                Type:       PING | Params:
66 [VERBOSE] Generated event: PROXIMITY_ALERT
67 [VERBOSE] Generated event: ACTION_FINISHED
68 [WARNING] Reaction not found.
69 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 20 |
                Type:       PONG | Params:
70 [VERBOSE] .. << M Sender:       1 | Receiver:     2 |        MsgId.: 20 |
                Type:       STOP | Params:
71 [VERBOSE] Generated event: BEHAVIOUR_STOP
72 [ DEBUG ] Behaviour stop.
73 [ INFO  ] Changing state to ACTIVE
74 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 21 |
                Type:       PING | Params:
75 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 22 |
                Type:       PING | Params:
76 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 23 |
                Type:       PING | Params:
77 [ INFO  ] Changing state to PANIC
78 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 24 |
                Type:       PING | Params:
79 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 25 |
                Type:       PING | Params:
80 [VERBOSE] A >> .. Sender:       2 | Receiver:     1 |        MsgId.: 26 |
                Type:       PING | Params:
81 [VERBOSE] .. << M Sender:       0 | Receiver:     0 |        MsgId.: 27 |
                Type: AGENT_OVER | Params:
82 [ INFO  ] Robot exiting...
83 [ INFO  ] Comm. exiting...
84 [ INFO  ] Exiting...
```

Rozdział 7

Podsumowanie

Praca z robotem LEGO EV3 Mindstorms w środowisku ev3dev przyniosła następuje rezultaty:

- Przeprowadzony został przegląd środowisk dostępnych dla LEGO EV3 i wybrany został projekt ev3dev
- Robot z zainstalowanym nowym oprogramowaniem został poprawnie skonfigurowany, a testy motorów oraz sensorów przebiegły pomyślnie
- Podłączenie robota do sieci WiFi za pomocą dodatkowego adaptera zakończyło się sukcesem
- Napisana została aplikacja w C++, działająca na systemie Linux, w której zawarte są poniższe funkcjonalności:
  - Implementacja prostych akcji, wykonujących sekwencje komend na urządzeniach robota
  - Możliwość wykonywania parametryzowalnych, złożonych zachowań, zaprogramowanych w postaci maszyny stanów
  - Pełna kontrola nad udostępnionymi przez sensory i motory interfejsami
  - Kontrola przebiegu wykonania za pomocą zdarzeń
  - Oddzielenie poszczególnych działań robota za pomocą zdefiniowanych stanów
  - Elastyczność definiowania nowych modeli robotów oraz interpretacji akcji przez rozszerzanie funkcjonalności za pomocą klas pochodnych
  - Komunikacja między fizycznymi jednostkami z wykorzystaniem protokołu UDP za pomocą interfejsu klas zawierających implementację systemowych funkcji przesyłania danych
  - Obsługa sygnałów dostarczanych do programu

  - Możliwość wydzielenia nadzorcy systemu, kontrolującego pracę agentów
  - Funkcja logowania, opatrzonego etykietami oraz kontrolą nad zapisywanymi rodzajami komunikatów.

- Przeprowadzono testy zarówno zachowania robota, jak i komunikacji za pomocą sieci WiFi.

Dzięki pracy nad tym projektem powstała aplikacja, realizująca przede wszystkim funkcje agenta, jego zachowania oraz komunikację, a także warstwa nadzorcza, monitorująca przebieg działań i zbierająca dane od zarządzanych przez nią agentów.

7.1 Perspektywy rozwoju

Istnieje wiele aspektów projektu, których implementacja ułatwi lub rozszerzy bieżące możliwości. Do takich należy na przykład:

- Dodanie ujednoliconego interfejsu obsługi sensorów produkowanych przez osoby trzecie
- Dodanie funkcji przywracania połączenia między agentami w przypadku całkowitej utraty wykorzystywanej sieci
- Umożliwienie wykonywania konkretnych akcji bez wymaganego połączenia agenta z nadzorcą
- Parametryzacja konfiguracji agenta przy pomocy zewnętrznych plików (na przykład XML)
- Dodanie konfiguracji, umożliwiającej kompilację projektu do zewnętrznej biblioteki w celu wykorzystania jej w innej aplikacji.

Aplikacja może być wykorzystywana jako dolna warstwa systemu wieloagentowego, a komunikacja może odbywać się za pomocą ujednoliconych wiadomości w protokole UDP. Po dalszej rozbudowie części nadzorującej, może też służyć jako warstwa sterująca, także po stronie zwykłego komputera. W obecnym stanie, po stronie agenta, dwa wątki aplikacji wykorzystują w sumie około 20% zasobów procesora oraz zajmują około 10% dostępnej pamięci RAM. Mimo możliwych optymalizacji, program zużywa na tyle mało zasobów sprzętowych, że może być uruchamiany obok innego procesu bez zauważalnego spadku wydajności.

Szczegóły techniczne, dotyczące struktury plików oraz rozwoju i implementacji nowych funkcjonalności, znajdują się w dodatku A.4.

Bibliografia

Publikacje

[1] Wolfram Burgard Sebastian Thrun i Dieter Fox. *Probabilistic Robotics*. Massachusetts Institute of Technology, 2006.

Materiały niepublikowane

[2] Henryk Dobrowolski. "Layered Software Architecture for Implementation in Mobile Robots MAS Student Lab (excerpt)".

Źródła internetowe

[3] *Biblioteka MonoBrick dla LEGO EV3*. URL: `http://www.monobrick.dk/software/ev3firmware/`.

[4] *Strona domowa projektu ev3dev*. URL: `http://www.ev3dev.org/`.

[5] *Strona projektu leJOS EV3*. URL: `http://www.lejos.org/`.

[6] *Zdalne debugowanie aplikacji z wykorzystaniem GDB*. URL: `http://www.ev3dev.org/docs/tutorials/using-brickstrap-to-cross-compile/`.

Słownik pojęć

agent - System komputerowy przebywający w dynamicznym środowisku, reagujący na jego zmiany i wykonujący przydzielone mu zadania. 2

ARM - Advanced RISC Machine. 4

automat skończony - Iteracyjny model zachowania systemu oparty na tablicy dyskretnych przejść między jego stanami. 2

efektor - Organ wykonawczy systemu autonomicznego, który oddziaływuje na otoczenie. 5

Git - Rozproszony system kontroli wersji zaprojektowany przez Linusa Torvaldsa. 11

I²C - Inter-Integrated Circuit. 58

sensor - Urządzenie wychwytujące i rejestrujące sygnały z otoczenia. 4

singleton - Wzorzec projektowy, który zapewnia jedną instancję klasy przez cały czas trwania programu. 15, 18

UART - Universal Asynchronous Receiver/Transmitter. 58

UDP - User Datagram Protocol. 4

zachowanie złożone - Algorytm postępowania agenta, złożony z wielu akcji prostych. 2

Spis rysunków

Dodatek A

Załączniki

A.1 Konfiguracja systemu

W tym załączniku znajduje się instrukcja kompilacji aplikacji oraz sposobu uruchamiania.

A.1.1 Kompilacja

Uwaga: Kompilacja konfiguracji na architekturę typu ARM wymaga zainstalowanego programu `arm-linux-gnueabi-g++`. Pozostałe dwie konfiguracje wymagają standardowego `g++`.

W środowisku graficznym

Wgrane na płytę CD pliki zawierają projekt aplikacji dla środowiska NetBeans (użyta wersja: 8.0.2). Po wczytaniu projektu, na górze programu można wybrać z listy rozwijanej bieżącą konfigurację i ją skompilować (klawisz F11).

W oknie konsoli

Projekt aplikacji NetBeans tak naprawdę korzysta z zapisanych przez siebie plików Makefile. W katalogu głównym znajduje się plik ogólny Makefile, a w folderze `nbproject` pliki poszczególnych konfiguracji. Proces budowania można rozpocząć poleceniem:

`make <nazwa-konfiguracji>` lub `make all`,

a usunięcie plików tymczasowych wykonuje komenda: `make clean`

A.1.2 Synchronizacja plików

W celu szybkiego wysłania plików do wielu robotów, przygotowany został specjalny katalog `sync` wraz ze skryptem kopiującym. Edytując skrypt można podać następujące wartości parametrów:

- Adresy ip wszystkich docelowych urządzeń, na które mają być wgrane pliki wykonywalne
- Nazwę użytkownika*
- Katalog docelowy*
- Hasło*

* - w przypadku, jeśli ten parametr jest identyczny na każdym urządzeniu.

Skrypt wykonuje kopię plików binarnych z katalogu `dist` dla konfiguracji ARM i umieszcza ją w katalogu `sync`. Następnie, przy pomocy polecenia `rsync` z użyciem podanego hasła, zawartość katalogu zdalnego będzie dokładnym odwzorowaniem bieżącego katalogu.

A.1.3 Uruchomienie aplikacji

Aplikacja może być uruchomiona na dwa sposoby: z poziomu interfejsu graficznego klocka centralnego lub przez zdalne uruchomienie za pomocą połączenia ssh. Ten drugi wariant pozwala lepiej kontrolować komunikaty wyświetlane na ekranie, ale bardziej obciąża urządzenie. Z tego powodu nie powinno się uruchamiać wersji mobilnej z ustawieniami logowania na standardowe wyjście, ale jeśli jest to wymagane, należy ograniczyć rodzaj wyświetlanych wiadomości do minimum.

Przesłany plik wykonywalny można uruchomić w dwóch trybach: agent i master. Ponadto, aplikacja przyjmuje następujące opcjonalne argumenty wywołania:

- log=<tryb> - sprecyzowanie minimalnego poziomu logowania wiadomości za pomocą jednego z następujących trybów: `none error warning info verbose debug`.

Zakończenie programu można przedwcześnie wymusić przez przesłanie sygnału SIGINT za pomocą kombinacji klawiszy CTRL+C.

Przykładowe uruchomienie:

```
./ev3dev agent log=warning
```

A.2 Zawartość płyty CD

Na dołączonej do pracy płycie CD znajdują się następujące katalogi:

- **doc** - zawiera wygenerowaną w programie Doxygen dokumentację w formie plików html oraz pdf
- **include** - zawiera pliki nagłówkowe aplikacji pogrupowane w moduły
- **nbproject** - katalog projektu aplikacji w środowisku NetBeans. Zawiera także pliki Makefile do kompilacji
- **src** - zawiera pliki źródłowe aplikacji, w szczególności plik startowy
- **sync** - katalog służący synchronizacji plików. Zawiera skrypt kopiujący odpowiednie pliki wykonywalne i przesyłający je do podanych urządzeń.

A.3 Zdalne debugowanie aplikacji

Narzędzie `brickstrap`, którego twórcy biblioteki ev3dev używają głównie do przygotowywania obrazu systemu dla LEGO EV3, może posłużyć również do debugowania aplikacji na klocku. Możliwe jest także utworzenie zdalnego systemu plików do synchronizacji danych pomiędzy dwoma systemami Linux za pomocą `nfs`. Konfiguracja środowiska wirtualnego[6] składa się z następujących kroków:

1. Instalacja narzędzia `brickstrap`
2. Stworzenie wirtualnego środowiska na bazie obrazu systemu identycznego z tym zainstalowanym na robocie LEGO
3. Kompilacja programu z poziomu powłoki maszyny witualnej (z flagą `-g` w celu dołączenia symboli debugowania)
4. Instalacja `gdbserver` po stronie klocka LEGO oraz `gdb` po stronie środowiska wirtualnego
5. Uruchomienie skompilowanego pliku binarnego za pomocą `gdbserver` po stronie klocka.

Po wykonaniu wszystkich kroków, opisanych szczegółowo na portalu ev3dev[6], można uruchomić `gdb` z parametrem identycznym jak nazwa programu uruchomiona na robocie. W ten sposób stworzona została interaktywna sesja debuggera i za pomocą komend wpisywanych bezpośrednio do terminala można np. ustalać punkty przerwań czy podglądać wartości zmiennych lokalnych. Z racji tego, że program `gdb` uruchamiany jest w środowisku wirtualnym, można czasami natrafić na nieobsługiwane wywołania systemowe. Większość napotykanych błędów lub ostrzeżeń nie powinna sprawiać problemów, ale może ograniczać niektóre funkcjonalności debuggera.

A.4 Rozwój projektu

Główne pliki projektu znajdują się w katalogach `include` oraz `src`. Pierwszy z nich zawiera pliki nagłówkowe, drugi źródłowe. Rozdzielenie plików pozwala w przyszłości skompilować projekt do osobnej biblioteki i w trakcie jej używania korzystać z już wyodrębionych plików z definicjami klas.

Poniżej przestawiono kilka możliwych aspektów rozwinięcia bieżącego projektu o nowe funkcjonalności.

A.4.1 Elastyczność konfiguracji

Dużym udogodnieniem, jakie można wprowadzić do aplikacji, jest możliwość zmiany konfiguracji za pomocą zewnętrznych plików, na przykład w formacie XML. Taka modyfikacja pozwoliłaby na:

- Dynamiczną zmianę konfiguracji w trakcie działania programu
- Łatwe zwiększanie liczby dostępnych zachowań a bilbiotece
- Elastyczną specyfikację każdego nowego modelu robota
- Szybsze definiowanie sposobu wykonywania akcji przez dany model agenta.

W celu osiągnięcia powyższych założeń, należałoby napisać parser plików konfiguracyjnych, który sprawdzałby ich poprawność oraz generował gotowe obiekty odpowiadające wczytanym parametrom. Ponadto, należałoby dodać metodę sprawdzającą co pewien interwał czasu, czy zawartość wczytanych plików nie uległa zmianie. Jeśli tak, konieczne byłoby ponowne ich załadowanie.

A.4.2 Nowe modele agentów

Aplikacja nie ogranicza użytkownika do korzystania z jednego modelu robota. Wręcz przeciwnie, dowolna konstrukcja może być użyta jako agent systemu. Osiągnięcie pełni funkcjonalności nowego robota wymaga następujących modyfikacji:

- Stworzenia klasy reprezentującej nowy model robota, która dziedziczy po głównej klasie `Robot`
- Zdefiniowanie wymaganych urządzeń oraz możliwych do wykonania akcji w konstruktorach nowej klasy
- Przeciążenie odpowiednich metod, odpowiedzialnych za generowanie struktury zachowań oraz konkretnych postaci akcji. Są to metody `generateBehaviour` oraz `generateAction`.

A.4.3 Komunikacja

Sprawna komunikacja między agentami to podstawa całego systemu. Na obecnym etapie, wykorzystywana jest istniejąca sieć WiFi, z którą łączą się wszyscy agenci. Każdy z nich komunikuje się tylko z nadzorcą, który zna stan wszystkich innych agentów. Niestety to rozwiązanie ma dwie podstawowe wady:

1. Nadzorca jest najsłabszym ogniwem całej komunikacji w systemie
2. System nie jest w stanie funkcjonować w przypadku błędów lub braku wspólnej sieci.

Pierwszy problem częściowo rozwiązuje przekazanie odpowiedzialności nadzorcy jednemu z agentów, ale w przypadku dużej ich ilości zasoby agenta mogą nie wystarczyć, żeby utrzymać system w pełni sprawnym.

Oba powyższe problemy można rozwiązać tworząc własną sieć dynamicznie przez jednego z agentów lub nadzorcę. Największą trudnością jest ustalenie, który agent tworzy sieć dla pozostałych i w jakiej kolejności przebiega cały proces.
W przypadku nagłej utraty połączenia, trzeba określić sposób jego odtworzenia.
W związku z tym, każda jednostka musi wiedzieć o wszystkich pozostałych, co przeczy pierwotnemu założeniu, że wszyscy agenci są niezależni (patrz A.4.4).

A.4.4 Wyelimowanie konieczności aktywnego nadzorcy

System nie zawsze może sobie pozwolić na bycie kontrolowanym przez jedną, główną jednostkę. Ograniczać to może przede wszystkim stan połączenia. Należy więc udostępnić możliwość komunikowania się robotom bezpośrednio między sobą. Takie rozwiązanie uzależnia agentów od siebie, a przez to pozwala na niezależną od zewnętrznych jednostek egzystencję. Implementacja inteligencji zbiorowej oraz sprawnego protokołu komunikacji umożliwiłaby samodzielne wykonywanie zadań jako grupa oraz eksplorację trudno dostępnych obszarów. Wada jednego agenta osłabiała by cały system, ale nie powodowałaby jego awarii, a zdolność działania wszystkich robotów byłaby ograniczona tylko poziomem naładowania akumulatora każdego z nich.

A.5 Używanie niestandardowych sensorów

A.5.1 Typy sensorów

Klocek LEGO EV3 posiada cztery wejścia dedykowane różnorodnym sensorom, multiplekserom sensorów lub kontrolerom motorów. Wyróżnia się cztery typy komunikacji wykorzystywane przez sensory: analogowy, NXT Color Sensor, I^2C oraz UART.

Typ analogowy

Jest to najprostszy typ sensorów. Mierzona wartość zamieniana jest na napięcie z przedziału 0-5V, które jest odczytywane przez klocek EV3. Wśród typów analogowych można wydzielić kilka kategorii:

EV3/Analog - zaprojektowane specjalnie dla EV3 i nie są kompatybilne z poprzednimi generacjami LEGO. Zawierają specjalny rezystor, który umożliwia identyfikację typu użytego sensora

NXT/Analog - dedykowane LEGO NXT, ale mogą współpracować z EV3. Z reguły wymagane jest podanie rodzaju sensora, ponieważ klocek centralny nie potrafi sam go zidentyfikować

WeDo/Analog - pod kątem elektroniki są niemal identyczne z sensorami analogowymi EV3 (napięcie 5V i rezystor identyfikujący). Obecnie nie wszystkie z nich są w pełni rozpoznawalne i wymagane są własne sterowniki

RCX - nie są kompatybilne z EV3, ponieważ inaczej wykorzystują wejściowe piny.

LEGO NXT Color Sensor

Ten sensor jest potraktowany wyjątkowo, ponieważ łączy analogową z cyfrową (niestandardową) komunikacją. Dla obecnej wersji systemu nie ma jednakże napisanego sterownika obsługi, ale klocek potrafi automatycznie wykryć obecność tego urządzenia.

Typ I^2C

Sensory tego typu komunikują się z inteligentnym klockiem za pomocą protokołu I^2C. Są to cyfrowe sensory, z których można wyróżnić te projektowane zgodnie z wytycznymi LEGO oraz te używające niestandardowych czipów I^2C. Tylko sensory pierwszego typu są wykrywane automatycznie. Na stronach projektu ev3dev, pierwszy typ jest wyróżniany jako NXT/I2C, a drugi jako Other/I2C.

Typ UART

Urządzenia wykorzytujące układ scalony UART do asynchronicznej komunikacji są zaprojektowane specjalnie dla EV3 i działają tylko z tą generacją robotów LEGO. Oprócz zwykłych danych o przeprowadzonych pomiarach, przekazują one informacje o swoich możliwościach. Oznaczałoby to, że sensory powinny od razu działać, bez konieczności pisania osobno każdego sterownika. Sensory typu EV3/UART, z racji tego, że "U" oznacza uniwersalny, powinny działać z każdym urządzeniem obsługującym UART, nie tylko portami wejściowymi klocka LEGO. Wszystkie domyślne sensory będące częścią zestawów EV3 są typu UART.

W zakładce "sensors" witryny ev3dev.org widnieje lista producentów oraz ich sensorów, które są automatycznie obsługiwane przez EV3. Najbardziej interesującym rodzajem sensorów[1], są te wykorzystujące protokół I^2C. Stanowią one przeważająca większość obecnych na rynku urządzeń pomiarowych dla LEGO.

[1]Obsługa sensorów I^2C - `http://www.ev3dev.org/docs/sensors/using-i2c-sensors/`