

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Automatyki i Informatyki Stosowanej

Praca dyplomowa magisterska

na kierunku Automatyka i Robotyka

Robot mobilny TIAGo wykorzystujący planowanie symboliczne
przy wspieraniu osób starszych

Łukasz Zieliński

Numer albumu 259155

promotor
dr inż. Tomasz Winiarski

WARSZAWA 2020

Robot mobilny TIAGo wykorzystujący planowanie symboliczne przy wspieraniu osób starszych

Streszczenie. Niniejsza praca zawiera projekt i opis implementacji aplikacji, której celem jest wykorzystanie robota społecznego przy wspieraniu osób starszych. Jest ona oparta na popularnym systemie Robot Operating System. Ważnym założeniem dotyczącym tej aplikacji jest wyodrębnienie z możliwych działań robota pewnych scenariuszy jego pracy.

Część związaną z planowaniem symbolicznym zaimplementowano wykorzystując język PDDL, następnie zintegrowano ją z resztą systemu ROS wykorzystując szkielet aplikacyjny ROSPlan. Dziedziny PDDL zostały zamodelowane tak, by każda reprezentowała oddzielny scenariusz pracy robota. Może on w ramach każdej dziedziny wykonać pewne konkretne działania (zwane akcjami). Każda taka akcja PDDL ma postać automatu skońzonego prototypowanego przy pomocy paczki SMACH, co wymusiło podział wszystkich działań robota na pewne podstawowe, niepodzielne stany. Całą aplikację zaimplementowano wykorzystując robota mobilnego TIAGo. W celach badań i testów korzystano także z jego symulatora.

Dzięki charakterowi języka PDDL oraz wykorzystaniu automatów skończonych aplikacja ma modularną budowę - jest przez to łatwa w rozwijaniu i przystępna w zrozumieniu. System może być uruchamiany wykorzystując naprzemiennie rzeczywistego robota i jego symulator, co usprawnia badania i testy. Może on w szczególności zostać łatwo zaimplementowany na innym, podobnym robocie.

Realizacja pracy pomogła zrozumieć trudności związane z zagadnieniem opieki nad starszymi ludźmi. Podkreśla także zalety i wady wykorzystania planowania symbolicznego w robotyce społecznej, a szczególnie istotę jego użycia w omawianym środowisku.

Słowa kluczowe: PDDL, PLANOWANIE SYMBOLICZNE, ROS, ROSPLAN, SMACH, STARSI LUDZIE

TIAGo mobile robot utilizing automated planning and scheduling in support for the elderly

Abstract. This work contains the design and description of the implementation process of an application that aims to use a social robot to support elderly. It is based on famous Robot Operating System. One of the main assumptions for this application is to extract certain scenarios of the robot's work from all the actions it can perform.

The part related to automated planning and scheduling is implemented using the PDDL language. It was then integrated with the rest of the ROS based application using the ROSPlan framework. PDDL domains were modelled so they would represent mentioned scenarios of the robot's work. The robot is able to perform some specific tasks (called actions) within each of them. Each such PDDL action has a form of a finite state machine created using the SMACH package, which forced the division of all robot actions into certain basic, indivisible states. The whole application was implemented using the TIAGo mobile robot. Its simulator was also used for research and testing purposes.

Thanks to the nature of the PDDL language and the use of finite state machines, the application has a modular structure - it is easy to develop and understand. The system can be run using both real robot and its simulator effortlessly, which improves research and testing. In particular, it can be easily implemented on another similar mobile robot.

The realization of the work has helped to understand the difficulties in the field of elderly care. It also highlights the advantages and disadvantages of using automated planning in social robotics, especially the essence of its use in this environment.

Keywords: PDDL, AUTOMATED PLANNING, SCHEDULING, ROS, ROSPLAN, SMACH, ELDERLY



Politechnika Warszawska

załącznik nr 3 do zarządzenia
nr 28 /2016 Rektora PW

Lukasz Zielinski

imię i nazwisko studenta

259155

numer albumu

AUTOMATYKA I ROBOTYKA

kierunek studiów

Warszawa, 31.01.2020

miejscowość i data

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

Zielinski Lukasz

czytelny podpis studenta

Praca była realizowana w ramach projektu INCARE AAL-2017-059
„Zintegrowany system innowacyjnych rozwiązań
dla opieki nad osobami starszymi”

Podziękowanie

Dziękuję mojemu promotorowi drowi inż. Tomaszowi Winiarskiemu za przeprowadzenie mnie przez całą drogę ku uzyskaniu dyplomu.

Składam także wyrazy uznania całemu Zespołowi Robotyków za wsparcie merytoryczne i wszystkie przekazane rady.

Ogromnie dziękuję moim rodzicom za wszelką pomoc, którą otrzymywałem od nich podczas ostatnich długich lat.

*Szczególne podziękowania należą się mojej dziewczynie Oli.
Dałaś mi wsparcie, bez którego nie ukończyłbym studiów.*

Spis treści

1. Wprowadzenie	15
1.1. Robotyka społeczna	15
1.2. Zadania podejmowane przez roboty	17
1.3. Cel i zakres projektu	17
1.4. Struktura pracy	18
2. Planowanie symboliczne	19
2.1. Wprowadzenie do zagadnienia symbolicznego planowania	19
2.2. Problem Domain Definition Language	19
2.2.1. Historia i inspiracje	19
2.2.2. Różne wersje języka i ich możliwości	20
2.2.3. Składnia	21
2.2.4. Przykłady	24
2.3. Symboliczne planowanie przy użyciu planerów	28
3. Robot TIAGo i narzędzia związane z systemem ROS	31
3.1. Oprogramowanie Gazebo i Blender	31
3.2. Platforma robota	32
3.2.1. Architektura sprzętowa	32
3.2.2. Oprogramowanie	34
3.2.3. Symulator	35
3.3. Robot Operating System	36
3.3.1. Charakterystyka ROS	36
3.3.2. Opis istotnych elementów systemu	37
3.4. Paczka actionlib	40
3.4.1. Komunikacja	40
3.4.2. Plik z opisem akcji <i>actionlib</i>	41
3.4.3. Uproszczona implementacja	41
3.5. Paczka SMACH	43
3.5.1. Przykład implementacji	43
3.5.2. Opakowanie kontenera SMACH przez actionlib	45
3.6. Szkielet aplikacyjny ROSPlan	46
3.6.1. Omówienie	46
3.6.2. Wewnętrzna struktura i komunikacja	46
3.6.3. Zarządzanie stanem przechowywanej wiedzy	51
3.6.4. Ograniczenia	52
4. Projekt rozwiązania	55
4.1. Założenia i oczekiwane efekty	55
4.1.1. Praca robota oparta na scenariuszach	55
4.1.2. Założenia dotyczące wykorzystania omówionych narzędzi	57
4.1.3. Oczekiwania związane z użytkowaniem aplikacji	58
4.2. Wykorzystanie planowania symbolicznego i narzędzi ROS	59
4.2.1. Integracja narzędzi. Szkielet systemu	59

0. Spis treści

4.2.2. Akcje realizowalne przez robota	60
4.2.3. Projekt scenariuszy pracy robota	63
4.2.4. Fizyczna realizacja zaplanowanych akcji - projekt automatów skończonych	70
4.3. Użycie dostępnej platformy i otoczenia	80
4.3.1. Robot TIAGo z zainstalowanym oprogramowaniem	80
4.3.2. Symulator	80
4.3.3. Środowisko pracy robota	80
5. Implementacja systemu	81
5.1. Konfiguracja symulatora i środowiska pracy robota	81
5.2. Implementacja komponentów systemu	84
5.2.1. Instalacja i konfiguracja wymaganych paczek ROS	84
5.2.2. Implementacja warstwy sterującej robotem	85
5.2.3. Uruchamianie systemu	86
5.3. Opis w języku PDDL	87
5.3.1. Omówienie dziedzin PDDL związanych z przygotowanymi scenariuszami	87
5.3.2. Przykładowe problemy PDDL	88
5.4. Automaty realizujące zaplanowane akcje	91
5.5. Uzupełnianie wiedzy o świecie	93
5.6. Ponowne planowanie na podstawie nowych faktów	94
6. Badania i wyniki oparte na przedstawionych scenariuszach	97
6.1. Projekt badań i środowisko testowe	97
6.2. Działanie interfejsu od strony technicznej - planowanie i realizacja	97
6.2.1. Hazard Detection	97
6.2.2. Transportation Attendant	99
6.2.3. Active Human Fall Prevention	102
6.2.4. Wyniki badań	104
6.3. Porównanie pracy systemu z rzeczywistym robotem i symulatorem	105
6.4. Działanie systemu od strony percepcyjnej	105
7. Podsumowanie	107
7.1. Zalety i ograniczenia zaimplementowanego systemu	107
7.2. Możliwości rozwoju	108
7.3. Wnioski	109
Bibliografia	111
Spis rysunków	115
Spis tabel	116
Spis załączników	116
A. Dziedziny problemów PDDL	117

1. Wprowadzenie

1.1. Robotyka społeczna

Starzenie się ludności nie jest zjawiskiem nowym, ale dopiero na przestrzeni ostatnich dziesięcioleci stało się zjawiskiem powszechnym. Coraz częściej wspomina się o *siwiejącej populacji* [1], zwraca się także uwagę na niepokojące dane demograficzne. Wskazuję one, że obecnie (rok 2019) żyje na całej planecie 703 miliony ludzi w wieku powyżej 65 lat. Przewiduje się natomiast, że liczba ta podwoi się (wzrośnie do 1,5 miliarda) do roku 2050. Wartości te rosną także po odniesieniu do całej populacji: społeczeństwo w wieku powyżej 65 lat stanowiło w 1990 roku jedynie 6% całej ludności. Dziś (w roku 2019) wartość ta wynosi 9% i przewiduje się jej dalszy wzrost - w roku 2050 ma ona wynieść aż 16% [2]. W krajach wysoko rozwiniętych takich jak USA (do 2050 roku populacja ludzi w wieku przekraczającym 65 lat wyniesie 26%) czy Japonia (populacja starszych osób już wynosi 20%) opieka nad starszymi ludźmi stanowi problem odczuwalny już dziś [3].

Wzrost liczby ludności starszej w stosunku do całego społeczeństwa jest zatem wysoce prawdopodobny. Powoduje to ekonomiczny i społeczny nacisk związany z potrzebą rozwijania programów opieki nad starszymi ludźmi. Już teraz powstaje wiele pomysłów i rozwiązań tego problemu, który tym silniej w przyszłości dotknie społeczeństwo na całym świecie. Są wśród nich metody konwencjonalne polegające między innymi na zmianach regulowanych przez władze ustawodawcze oraz metody niekonwencjonalne, wśród których czołowe miejsce zajmuje robotyka.

Można zauważać intensywny rozwój dwóch gałęzi robotyki wykorzystywanej w celu wspierania osób starszych. Pierwsza z nich dotyczy robotów terapeutycznych [4]. Warto tu wspomnieć o japońskim robocie Paro [5] (rysunek 1a). Jest to urządzenie przypominające maskotkę w kształcie foki i jest wyposażone w czujniki taktyльne, światła, położenia, obecne są również mikrofony. Pozwalają one temu robotowi na imitowanie zachowania domowego zwierzęcia. Należy się nim opiekować, Paro może także uczyć się pewnych zachowań człowieka oraz reagować na swoje nowe imię lub dotyk. Według badań [6] obecność robotów tego typu (w szczególności robota Paro) powoduje u ludzi zmniejszanie poziomów stresu, ułatwia komunikację osób starszych z opiekunami i wpływa pozytywnie na szereg innych czynników psychologicznych.

Druga gałąź robotyki służącej opiece nad starszymi ludźmi dotyczy robotów usługowych. Są to zazwyczaj roboty znacznie bardziej zaawansowane technologicznie niż roboty terapeutyczne. Są one robotami mobilnymi, autonomicznymi, których zadaniem jest aktywna opieka nad ludźmi i wspomaganie pracy opiekunów. Świetszym przykładem jest robot Stevie [7] (rysunek 1b), którego budowa przypomina już robota humanoidalnego. Można zauważać, że dużo uwagi przywiązano budowie jego głowy: wyposażona jest ona w dwa wyświetlacze imitujące oczy i usta. Pozwala to robotowi na łatwe „wyrażanie uczuć” co jest jednym z kluczowych czynników w tej dziedzinie. Robot ten dzięki swojej

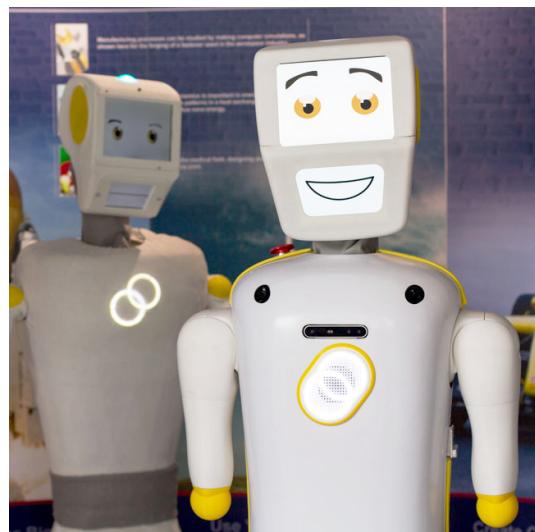
1. Wprowadzenie

mobilnej platformie może się swobodnie poruszać po otoczeniu. Stevie dzięki zainstalowanym aplikacjom umożliwia wideorozmowy z krewnymi, organizuje gry i zabawy logiczne oraz jest w stanie prowadzić proste rozmowy z ludźmi. Wszystkie te funkcje pozytywnie wpływają na starsze osoby, w szczególności pozwalają one na stymulowanie myślenia i utrzymywanie aktywności umysłowej.

Poruszając temat mobilnej robotyki usługowej (i nie tylko) należy koniecznie wspomnieć o jej oprogramowaniu. Wszelkie zachowania robotów są zasługą właśnie oprogramowania, które w omawianym zagadnieniu opieki nad starszymi ludźmi okazuje się zwykle trudne w wytworzeniu. Opracowanie nawet pojedynczych grup zachowań czy funkcji robota usługowego może pochłonąć ogrom czasu i wymagać specjalistycznego podejścia. Dobrym tego przykładem są prowadzone obecnie badania nad zagadnieniem związanym z równoległym wykonywaniem wielu czynności przez jednego robota [8]. Należy podkreślić, że takie zachowania dla człowieka są w zupełności naturalne, w robotyce natomiast jest wprost przeciwnie. Z uwagi na obszerność i poziom skomplikowania oprogramowania robotycznego istotna jest także kwestia jego samej architektury. Jest to zagadnienie bardzo rozległe, a w omawianym kontekście coraz częściej poruszane z uwagi na powstające nowe aplikacje i chęć wygodnego korzystania z obecnych już osiągnięć. Wśród prowadzonych obecnie badań nad tą dziedziną szczególną uwagę przyciąga szkielet aplikacyjny FABRIC (Framework for Agent-Based Robot Control Systems) [9]. Jest to narzędzie wspomagające budowę opartych na strukturze agentowej [10] systemów sterowania robotami.



(a) Robot Paro



(b) Robot Stevie

Rysunek 1. Roboty Paro i Stevie

Niniejsza praca skupiać się będzie na zagadnieniach związanych z drugą wspomnianą gałęzią, a zatem z robotyką usługową.

1.2. Zadania podejmowane przez roboty

Należy zwrócić uwagę, że roboty usługowe muszą być zdolne do wykonywania zadań zdefiniowanych na wysokim poziomie abstrakcji. Będą to przykładowo:

- przejazd do określonego miejsca,
- podanie człowiekowi napoju,
- nawiązanie interakcji z człowiekiem

i tym podobne. Są to zadania łatwo zrozumiałe i trywialne do wykonania dla człowieka. W większości sytuacji należy wykonać wiele takich zadań w odpowiedniej kolejności w celu realizacji pewnych bardziej złożonych celów, co także nie stanowi dla człowieka większego problemu. Ocena sytuacji, zaplanowanie i wykonanie odpowiednich podstawowych zadań przychodzi ludziom stosunkowo łatwo - ocena i zaplanowanie działań odbywa się często mimowolnie i nie jesteśmy tego świadomi. Programowanie takich zachowań w robotyce jest jednak ogromnym wyzwaniem i stanowi odrębny obszar badań nazywany często sztuczną inteligencją.

W robotyce (nie tylko usługowej) istnieje dziś wiele różnych podejść do tego zagadnienia. Wykorzystują one często wyrafinowane algorytmy i systemy decyzyjne, są wśród nich również algorytmy związane głównie z samym planowaniem zadań. Jeśli chodzi o planowanie należy szczególnie zwrócić uwagę na zagadnienie planowania symbolicznego [11]. Jest to dziedzina znana już przynajmniej od kilku dziesięcioleci, jednak rozwój robotyki pozwolił na jej szersze wykorzystanie. Dobrym przykładem jest wykorzystanie pewnej formy planowania symbolicznego w łazikach marsjańskich [12] - jest to uzasadnione przede wszystkim przez samą naturę problemu, a mianowicie przez długi czas podróży informacji do i od robota. Pozwala to na autonomiczną, a jednak jawnie obostrzoną przez pewne ograniczenia pracę robota. Podejmowane są również próby wykorzystania planowania symbolicznego w robotyce przemysłowej [13] oraz także w omawianej robotyce usługowej [14].

Z uwagi na zainteresowanie autora zagadnieniem planowania symbolicznego oraz chęcią współpracy przy projekcie będącym podejściem do rozwiązania rzeczywistego problemu powstała niniejsza praca. Jest ona propozycją systemu integrującego robotykę usługową i zagadnienie planowania symbolicznego. Praca ta pozwoli zatem na przeprowadzenie stosownych badań i wyciągnięcie wniosków dotyczących możliwości wykorzystania takiego rozwiązania z naciskiem na użycie go w środowiskach osób starszych.

1.3. Cel i zakres projektu

Celem pracy jest projekt i wykonanie systemu pozwalającego na wykorzystanie robota mobilnego TIAGo we wspomaganiu osób starszych. Użyte zostanie również planowanie symboliczne, co przy wykorzystaniu informacji o otoczeniu pozwoli na optymalną realizację zadanych celów przez robota. Cały projekt oparty będzie na systemie ROS (Robot

1. Wprowadzenie

Operating System) [15] co oznacza, że jego część polegać będzie na pracach związanych ze sprzęgnięciem systemu ROS z zagadnieniem planowania symbolicznego. Warto dodać, że praca robota oparta będzie o przygotowane wcześniej scenariusze, w ramach których opisane będą działania możliwe do wykonania przez robota.

W zakres niniejszej pracy wchodzą zarówno projekt systemu jak i jego wykonanie. Będą to zatem wszelkie działania polegające na wyborze odpowiednich narzędzi umożliwiających zrealizowanie pracy oraz ich wykorzystanie. Należy także wspomnieć, że do zakresu pracy także zaliczać się będą badania zaimplementowanego systemu oraz jego weryfikacja. W niektórych częściach pracy (szczególnie przy etapie projektowania systemu) zostanie użyty język SysML [16] w celu ułatwienia prac nad projektem oraz odpowiedniego przedstawienia pewnych jego elementów.

1.4. Struktura pracy

Praca została podzielona na 4 główne części:

— **Wprowadzenie teoretyczne** - rozdziały 2 i 3.

Zostaną tu przedstawione narzędzia wykorzystane w tej pracy. Zostanie przede wszystkim objaśnione zagadnienie planowania symbolicznego w zakresie niezbędnym do realizacji pracy. Następnie omówione będą pozostałe narzędzia: programistyczne - związane z systemem ROS oraz sama platforma robota, na której będzie oparta implementacja omawianego systemu, oraz która pozwoli na przeprowadzenie niezbędnych badań.

— **Projekt rozwiązania** - rozdział 4.

W tym rozdziale będzie przedstawiony projekt kompletnego systemu wraz z niezbędnymi założeniami. Wykorzystana będzie wiedza opisana w rozdziałach dotyczących wprowadzenia teoretycznego, by przedstawić pomysł autora na realizację niniejszej pracy.

— **Implementacja systemu** - rozdział 5.

Opisana będzie tu budowa zaprojektowanego rozwiązania z wykorzystaniem konkretnych narzędzi programistycznych i robotycznych. Zostanie udokumentowany sposób implementacji systemu.

— **Badania i wyniki** - rozdział 6.

Przedostatni rozdział zawierać będzie omówienie możliwości i ograniczeń zaimplementowanego rozwiązania. Zostaną zaproponowane i wykonane odpowiednie badania i testy co pozwoli na podsumowanie pracy i wyciągnięcie stosownych wniosków.

— **Podsumowanie** - rozdział 7.

2. Planowanie symboliczne

2.1. Wprowadzenie do zagadnienia symbolicznego planowania

Planowanie jest zadaniem polegającym na wyborze odpowiedniego ciągu akcji, który doprowadzi do realizacji celu (rysunek 2). Wspomniane akcje są działaniami opartymi na wysokopoziomowym opisie świata. Dokładniejszym opisem tego zagadnienia może być definicja 1:

Definicja 1 ([17]). *Zadanie planowania można wyrazić jako wektor*

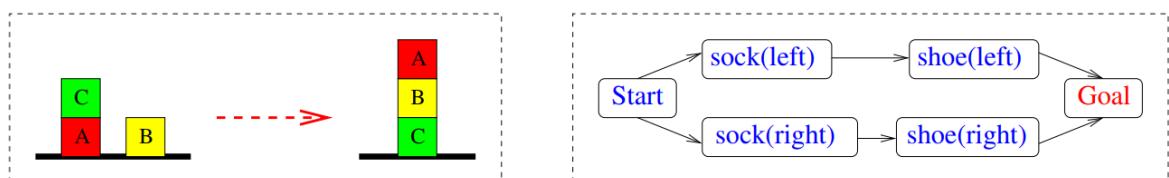
$$\Pi = (V, A, s_0, s_*),$$

gdzie:

- $V = \{v_1, \dots, v_n\}$ jest zbiorem zmiennych stanu,
- A jest zbiorem akcji a , gdzie każda akcja a jest parą (war_a, efe_a) wyrażeń zwanych **warunkami i efektami**,
- s_0 jest wyrażeniem zwanym **stanem początkowym**,
- s_* jest wyrażeniem zwanym **celem**.

Zadanie planowania jest zatem związane ze swoją przestrzenią stanu (grafem skierowanym wszystkich stanów). Krawędzie takiego grafu łączą stany s ze stanami s' , jeśli istnieje taka akcja $a \in A$ której warunki war_a spełnione są przez s a zmiana stanu s zgodnie z efektami efe_a da stan s' . Należy zwrócić uwagę, że efekty efe_a nadpisują wartości poprzednich wyrażeń.

Plan jest ścieżką zbudowaną z krawędzi omawianego grafu łączącą stan s_0 ze stanem opisanym przez cel s_* . Plan jest **optymalny** jeśli ścieżka ta jest możliwie najkrótsza.



Rysunek 2. Strywializowane planowanie (przebieg i efekt) [17]

2.2. Problem Domain Definition Language

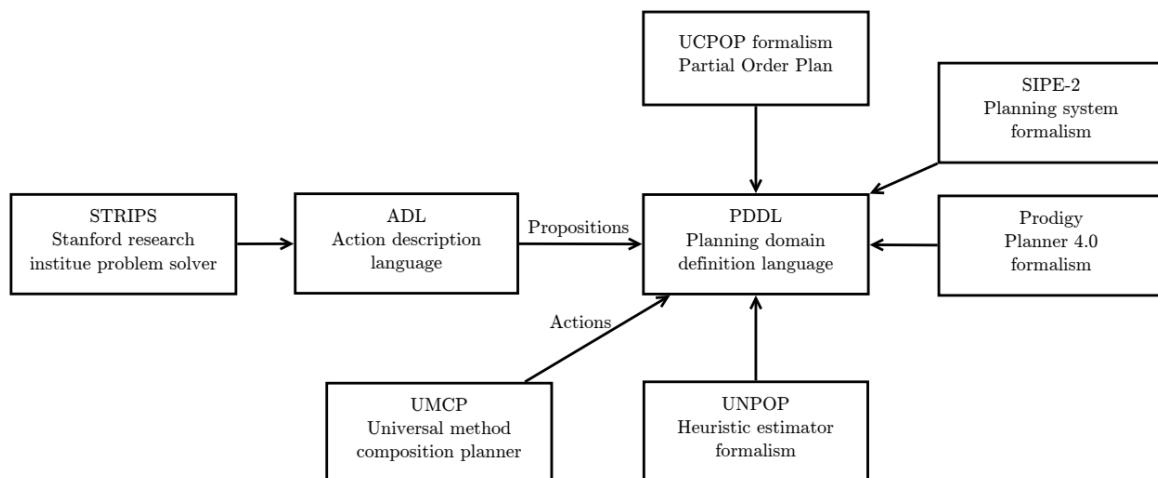
2.2.1. Historia i inspiracje

Problem Domain Definition Language (PDDL) [18] jest językiem symbolicznego planowania rozwijanym na początku swojego istnienia przez Drew McDermott. Jego praca została opublikowana w 1998 roku [18] i była próbą ustandaryzowania obecnych już języków opisu zadań z tej dziedziny. Język ten był zainspirowany przez *Stanford Research*

2. Planowanie symboliczne

Institute Problem Solver (STRIPS), Action Description Language (ADL) i kilka innych obecnych wtedy podejść (rysunek 3). Na PDDL został oparty pierwszy turniej *International Planning Competition* (IPC) [19].

STRIPS [20] powstały w 1971 roku jest oficjalnie najstarszym językiem symbolicznego planowania. Na jego podstawie powstało wiele bardziej nowoczesnych rozwiązań, co również przyczyniło się do dalszego rozwijania samej dziedziny. Język ten pozwolił także na ustalenie pewnego standardu składni opisu problemów symbolicznego planowania i sposobów rozwiązywania ich. Naukowiec Edwin Pednault w 1987 roku rozszerzył możliwości STRIPS (przede wszystkim przez wprowadzenie operatorów warunkowych [21]), w wyniku czego powstał język ADL.



Rysunek 3. PDDL wyewoluował z różnych obecnych rozwiązań [22]

- *ADL* - jeden z języków symbolicznego planowania, rozszerzenie STRIPS.
- *UMCP* - kompletny system planowania symbolicznego wykorzystujący hierarchiczne sieci zadań (HTN) [23].
- *UNPOP* - planer języka STRIPS wykorzystujący tak zwane heurystyki stanów [24].
- *UCPOP* - jeden z planerów języka ADL [25].
- *SIPE-2* - system planowania symbolicznego również wykorzystujący HTN.
- *Prodigy* - jedna z formalizacji metod planowania symbolicznego [26].

Jak już wspomniano (i wskazano na rysunku 3) sam język PDDL jest przede wszystkim pochodną języków ADL i STRIPS oraz modyfikacji, które pozwoliły na wykorzystanie go z istniejącymi systemami.

2.2.2. Różne wersje języka i ich możliwości

Od wykreowania PDDL powstało kilka jego wersji (zmiany zachodziły z każdym kolejnym konkursem IPC).

- PDDL 1.2
Wykorzystany podczas pierwszej i drugiej edycji konkursu IPC (w latach 1998 i 2000) [19].

Został wprowadzony podział opisu zagadnienia planowania na dwa pliki: problem (z opisem konkretnego zadania do rozwiązania) i dziedzinę (z opisem elementów wspólnych dla wielu zadań) [18].

— PDDL 2.1

Wykorzystany podczas trzeciej edycji IPC. Wprowadzono ciągłe zmienne, metryki oraz akcje o niezerowym czasie trwania.

— PDDL 2.2

Wykorzystany podczas czwartej edycji IPC. Wprowadzono dziedziczne predykaty i zależne od czasu zdarzenia.

— PDDL 3.0

Wykorzystany podczas piątej edycji IPC. Wprowadzono twardie i miękkie ograniczenia optymalizacji planowania. Usprawniono język pod kątem aktualnych planerów.

— PDDL 3.1

Najnowsza wersja języka, wykorzystana podczas szóstej i siódmej edycji IPC. Rozszerzono możliwości zmiennych - ich wartości mogą być nie tylko liczbą, ale także dowolnym obiektem.

2.2.3. Składnia

Każde zadanie opisane w języku PDDL składa się z dwóch plików, którymi są **dziedzina** oraz **problem**.

Dziedzina powinna zawierać przedstawione poniżej elementy:

- **nazwa** - jest ona potrzebna przy łączeniu problemów z daną dziedziną.
- **lista wymagań** - należy tu wpisać dodatkowe funkcje, z których się korzysta. Jest to informacja dla planera - wymagania te wskazują między innymi wersję języka PDDL (rozdział 2.2.2). Różne planery obsługują różne wersje języka, często może okazać się więc, że używana wersja (z wymienioną w wymaganiach funkcjonalnością) jest obsługiwana jedynie przez nieliczne planery. Przykładem niech będzie wykorzystywane w części implementacyjnej niniejszej pracy wymaganie *:durative-actions*. Oznacza ono wykorzystanie akcji o niezerowym czasie trwania, a jak wskazano wcześniej (rozdział 2.2.2), jest to rozszerzenie wprowadzone dopiero w wersji 2.1. Planer, który tej wersji nie obsługuje powinien wtedy zakończyć pracę błędem z odpowiednim komentarzem wskazującym na to wymaganie.
- **opis typów obiektów** - pozwala na deklarację obiektów w pliku z opisem problemu. Poza tym dzięki typom możliwe jest odpowiednie definiowanie parametrów predykatów oraz akcji co z kolei pozwala na przejrzysty i łatwiejszy do zrozumienia opis.
- **opis predykatów** - pozwala na definicję właściwości istniejących obiektów i ich oddziaływanie między sobą za pomocą logicznych wyrażeń. Przykładem niech będzie predykat (*is_gripper_empty*) którego istnienie świadczyć będzie o tym, że chwytki jest pusty. Predykaty mogą również przyjmować parametry. Poprzedni przykład może

2. Planowanie symboliczne

być rozszerzony do postaci (*is_gripper_empty ?x* - *gripper ?y* - *robot*), która przyjmuje dwa argumenty o typach *gripper* oraz *robot*. Pierwszy niech przyjmuje wartość *left* lub *right*, a drugi (*robot*) informację określającą konkretnego robota, np. jego nazwę. Użycie takiej postaci pozwala na dokładniejszy opis w porównaniu z tym samym predykatem bez parametrów, ponieważ w tym przypadku dostępna jest informacja o konkretnym chwytaku na danym robocie, co jest niezbędne gdy rozważane jest środowisko wielorobotowe.

- **listę akcji** - umożliwia opis akcji, które posłużą planerowi do utworzenia odpowiedniego ich ciągu pozwalającego na realizację zadanego celu (definicja 1). Każda taka akcja jest identyfikowana po unikalnej nazwie. Powinna opcjonalnie zawierać poniższe elementy.
 - **parametry** - lista parametrów wraz z ich typami, które wykorzystane będą w predykatach będących częścią warunków i efektów.
 - **warunki** - takie wyrażenie logiczne, które musi być prawdziwe by dana akcja mogła zostać wykonana.
 - **efekty** - wyrażenie logiczne, które zostanie spełnione po wykonaniu danej akcji.

Poniżej zostaną objaśnione możliwe do wykonania działania na wyrażeniach logicznych, o których wspomniano wyżej.

- **and** - ekwiwalent sumy logicznej. Pozwala na łączenie predykatów wewnętrz warunków i efektów. Aby do wykonania akcji wymagane było spełnienie wielu predykatów powinny one być zawarte wewnątrz sumy logicznej, np. (*and (pred1 ?x) (pred2 ?y)*).
- **or** - ekwiwalent alternatywy logicznej. Możliwy do wykorzystania w wymaganiach akcji. Pozwala na spełnienie jedynie jednego z predykatów zawartych wewnątrz alternatywy, aby dana akcja mogła zostać wykonana. Przykładowo zapis wymagania (*or (pred1 ?x) (pred2 ?y)*) pozwoli akcji na wykonanie ze spełnionym tylko predykatem **pred1 lub pred2**.
- **not** - ekwiwalent logicznej negacji. Pozwala na zanegowanie predykatu. Posiłkując się podanym wcześniej przykładowym predykatem może to być (*not (is_gripper_empty ?x ?y)*).
- **forall** - pozwala na wpływanie na wszystkie obiekty danego typu. Przykładowo efekt akcji opisany wyrażeniem (*forall (?r - robot) (is_broken ?r)*) [27] spowoduje spełnienie predykatu *is_broken* dla wszystkich obiektów typu *robot*.
- **when** - wyrażenie to również może zostać wpisane w ramach efektów akcji. Jeżeli warunek zawarty w nim zostanie spełniony to zakończenie akcji spowoduje również spełnienie predykatów tam opisanych. Na przykład zapisanie takiego efektu akcji *and (when (pred1 ?x) (pred2 ?y))* spowoduje spełnienie predykatu *pred2 jeśli prawdziwy jest predykat pred1*.

Warto zwrócić teraz uwagę na wzór pliku dziedziny PDDL (listing 1).

Listing 1. PDDL - wzór dziedziny

```
(define (domain nazwa_dziedziny)

(:requirements :adl :typing ...)

(:types a
  b - c
  ...)

(:predicates (predykat1 ?x - a)
  (predykat2 ?x - a~?y - b)
  ...)

(:action akcja1
  :parameters (?param1 - a
    ?param2 - b)
  :precondition (and
    predykat1 ?param1
    ...)

  :effect (and
    predykat2 ?param1 ?param2
    ...)
  )

  (:action akcja2
    ...)
  ...
  ...)
```

Poniżej przedstawiono strukturę opisu **problemu** PDDL. Każdy plik, który ją zawiera powinien składać się z elementów:

- **nazwa problemu** (*problem*).
- **nazwa dziedziny** (*domain*) - powinna tu być wpisana dokładna nazwa dziedziny, do której podłączony ma zostać ten problem.
- **opis obiektów** (*objects*) - definicja obiektów o odpowiednich typach, które zostały wcześniej przedstawione w pliku z dziedziną.
- **opis stanu początkowego** (*init*) - lista predykatów, które opisują aktualny (początkowy) stan wiedzy o świecie. W przypadku przedstawionego później problemu wieży Hanoi (listing 4) jest to opis położenia wszystkich elementów, od którego będzie rozpoczęte przekładanie.
- **cel** (*goal*) - lista predykatów, które mają być prawdziwe po prawidłowym zrealizowaniu planu.

Listing 2. PDDL - wzór problemu

```
(define (problem nazwa_problemu)
  (:domain nazwa_dziedziny)
```

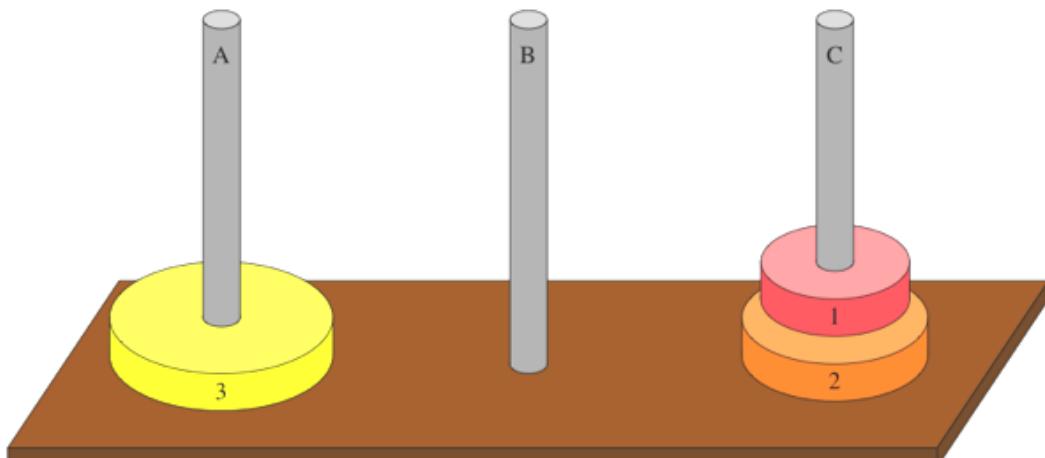
```
(:objects obj1 - a
         obj2 - b
         obj3 - c)
(:init predykat1 obj1
      ...
      )
(:goal (and
         (predykat2 obj1 obj2)
         ...
         )
      )
      )
```

2.2.4. Przykłady

Przedstawiono powyżej podstawowe zagadnienia związane z językiem PDDL. Zostaną teraz pokazane dwa rozwiążane przykłady - zawartość ich modeli PDDL oraz wygenerowany na ich podstawie plan.

Wieże Hanoi

Jest to gra logiczna, polegająca na przełożeniu zestawu krążków o różnej średnicy ułożonych na patyku od największego do najmniejszego na inny patyk, przy czym kolejność ułożenia krążków musi być zachowana. Dodatkowo sformułowane są następujące zasady: nie można kłaść krążka większego na krążek mniejszy oraz można jednocześnie przekładać tylko jeden krążek i można go zdejmować tylko z góry jednego ze stosów. Rysunek 5 przedstawia rozwiązywanie tego problemu na grafie, gdzie każdy węzeł jest możliwą do osiągnięcia konfiguracją przy wykorzystaniu ruchów reprezentowanych przez krawędzie grafu. Natomiast na listingach 3 (dziedzina) i 4 (problem) przedstawiono model tej gry w języku PDDL (przypadek dla trzech krążków).



Rysunek 4. Problem układania wieży Hanoi [28]

Listing 3. Wieże Hanoi - dziedzina

```
(define (domain hanoi)
(:requirements :strips)
(:predicates (clear ?x)
             (on ?x ?y)
             (smaller ?x ?y))
(:action move
:parameters (?disc ?from ?to)
:precondition (and
                (smaller ?to ?disc)
                (on ?disc ?from)
                (clear ?disc)
                (clear ?to)))
:effect (and
          (clear ?from)
          (on ?disc ?to)
          (not (on ?disc ?from)))
          (not (clear ?to))))
)
```

Listing 4. Wieże Hanoi - problem dla 3 dysków

```
(define (problem hanoi3)
(:domain hanoi)
(:objects peg1 peg2 peg3 d1 d2 d3)
(:init
  (smaller peg1 d1) (smaller peg1 d2) (smaller peg1 d3)
  (smaller peg2 d1) (smaller peg2 d2) (smaller peg2 d3)
  (smaller peg3 d1) (smaller peg3 d2) (smaller peg3 d3)
  (smaller d2 d1) (smaller d3 d1) (smaller d3 d2)
  (clear peg2) (clear peg3) (clear d1)
  (on d3 peg1) (on d2 d3) (on d1 d2))
(:goal (and
        (on d3 peg3)
        (on d2 d3)
        (on d1 d2)))
)
```

W przykładzie zdefiniowano sześć obiektów: trzy patyki (*peg1*, *peg2*, *peg3*) oraz 3 krążki (*d1*, *d2*, *d3*). Dodatkowo opis stanu przedstawiony jest przy użyciu trzech predykatów:

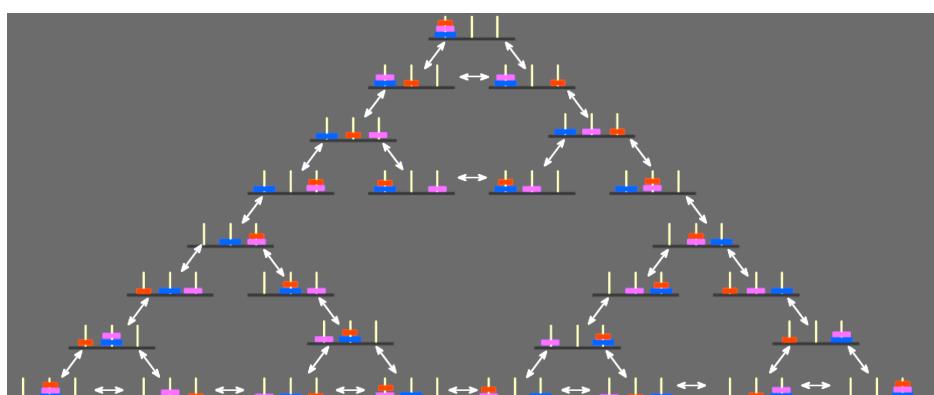
- *smaller* - określa czy jeden element jest mniejszy od drugiego. Przykładowo jeśli krążek *d2* jest mniejszy od krążka *d1* to predykat (*smaller d2 d1*) jest prawdziwy.
- *clear* - określa czy na danym elemencie można położyć inny element lub czy dany krążek można przełożyć.

2. Planowanie symboliczne

- *on* - predykat opisuje to czy dany krążek leży bezpośrednio na innym krążku lub patyku.

W pliku dziedziny (listing 3) zdefiniowane są predykaty i akcje całego modelu. W tym przypadku akcja jest tylko jedna (o nazwie *move*), której działanie polega na przełożeniu krążka w inne miejsce. Wśród wymagań tej akcji można zauważyć to, że krążek docelowy musi być na szczycie stosu (*clear ?to*) oraz że sam krążek przekładany również musi być na wierzchu (*clear ?disc*). Dodatkowo krążek przekładany musi być mniejszy od krążka, który znajdzie się pod spodem (*smaller ?to ?disc*). Ostatnie wymaganie opisane jako (*on ?disc ?from*) określa, że przekładany krążek musi leżeć na innym obiekcie (krążku lub tylko patyku).

Po wykonaniu akcji zmieni się stan wiedzy zgodnie z opisanymi efektami akcji. Pierwszym z nich jest (*clear ?from*) - obiekt, z którego został zdjęty krążek jest teraz na wierzchu. Dodatkowo krążek przełożony jest już w nowym miejscu (*on ?disc ?to*). Nie jest zatem już prawdziwy predykat opisujący jego poprzednie położenie (*not (on ?disc ?from)*), oraz element, na którym ułożony został nowy krążek nie znajduje się już na wierzchu (*not (clear ?to)*).



Rysunek 5. Problem układania wieży Hanoi przedstawiony na grafie [29]

Na przedstawionym przykładzie wieży Hanoi warto zauważyć, że **nie** został sformułowany żaden algorytm, który miałby posłużyć do rozwiązania tego problemu. Opisano jedynie model świata oraz czynność, która może zostać wykonana i której zrealizowanie ma pewien wpływ na cały model.

Plan wygenerowany na podstawie przedstawionego przykładu przedstawiono na listingu 5.

Listing 5. Plan wygenerowany na podstawie przyk³adu problemu wiezy Hanoi

```
(move d1 d2 peg3)  
(move d2 d3 peg2)  
(move d1 peg3 d2)  
(move d3 peg1 peg3)
```

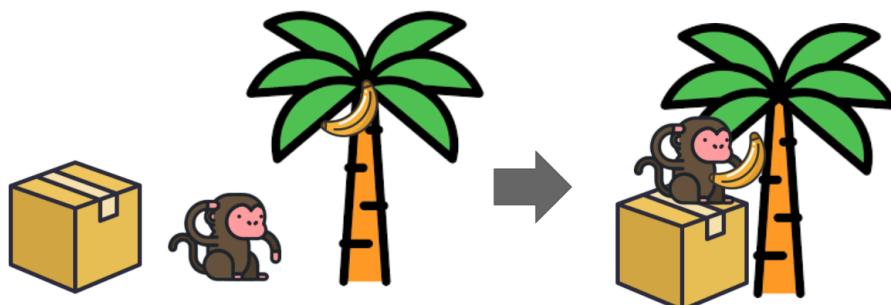
```
(move d1 d2 peg1)
(move d2 peg2 d3)
(move d1 peg1 d2)
```

Problem małpy i banana

Jest to problem postawiony przed małpą w laboratorium, która chce dostać się do bananów. Te jednak zawieszone są pod sufitem (lub sugerując się rysunkiem 6 banany zwisają z drzewa). Aby do nich się dostać małpa musi podsunąć w odpowiednie miejsce skrzynię, z której będzie zdolna zerwać banany przy użyciu noża. Z uwagi na znaczny rozmiar dziedziny PDDL tego modelu przedstawiono ją w całości w załączniku (listing 33). Zawarte są tam przedstawione niżej akcje, których dokonać może małpa:

- **goto** - małpa przemieszcza się do wybranego miejsca.
- **climp** - wejście na skrzynię.
- **push-box** - małpa popycha skrzynię w wybrane miejsce.
- **get-knife** - małpa wyposaża się w znaleziony w określonym miejscu nóż.
- **grab-bananas** - przy użyciu noża małpa znajduje się w posiadaniu bananów.

Z uwagi na to, że zostało przedstawione krótkie wprowadzenie teoretyczne do składni języka PDDL wraz ze szczegółowo omówionym przykładem wieży Hanoi (rysunek 4), nie zostaną tu omówione akcje tego modelu. Są one również przedstawione w dziedzinie (listing 33) w przystępny sposób, co nie powinno sprawić problemów z ich interpretacją.



Rysunek 6. Problem małpy i banana [30]

Na listingu 6 przedstawiono krótki problem PDDL tego modelu. Zdefiniowano tam obiekty (miejsca) *p1, p2, p3, p4*. Wewnątrz dyrektywy (*:init*) użyto następnie tych miejsc, by określić położenie kolejno małpy, skrzyni, bananów i noża. Dodatkowo zapisano, że małpa znajduje się aktualnie na podłodze (*on-floor*). Cel jest jeden, zapisany jako (*hasbananas*), jest on efektem wykonania akcji (*grab-bananas*) (listing 33). Plan wygenerowany na podstawie tego modelu przedstawiono na listingu 7.

2. Planowanie symboliczne

Listing 6. Plik z opisem problemu małpy i banana

```
(define (problem monkey-01)
  (:domain monkeyproblem)
  (:objects p1 p2 p3 p4)
  (:init

    (at monkey p1)
    (on-floor)
    (at box p2)
    (at bananas p3)
    (at knife p4)

  )

  (:goal
    (and
      (hasbananas)
    )
  )
)
```

Listing 7. Plan wygenerowany na podstawie przykładu problemu małpy i banana

```
(goto p2 p1)
(push-box p4 p2)
(get-knife p4)
(push-box p3 p4)
(climp p3)
(grab-bananas p3)
```

2.3. Symboliczne planowanie przy użyciu planerów

Przedstawione w poprzednim rozdziale listingi 5 oraz 7 zawierają przykładowe plany wygenerowane na podstawie modeli PDDL. Generacja ta odbywa się przy użyciu planera, czyli programu, który wczytuje problem i dziedzinę zapisane w języku PDDL i dzięki zaimplementowanemu algorytmowi tworzy ciąg akcji, którego efektem jest zrealizowanie zdefiniowanego celu. Omówienie wspomnianych algorytmów nie leży w zakresie tej pracy. Planer będzie więc traktowany jak czarna skrzynia (blackbox), która w odpowiednio krótkim czasie generuje gotowy plan.

Zestawienie popularnych planerów

Przedstawiono poniżej kilka wolno dostępnych planerów, omówiono także krótko możliwości jednego z nich.

- **OPTIC (Optimising Preferences and Time-Dependent Costs)** [31]
- **POPF** [32]
- **FF (Fast Forward)** [33]
- **LPG (Local search for Planning Graphs)** [34]
- **SMTPlan+ (Satisfiability Modulo Theories)** [35]
- **TFD (Temporal Fast Downward)** [36]

Omówienie wybranego planera TFD

W celu realizacji omawianego w kolejnych rozdziałach projektu (rozdział 4) zdecydowano się na wykorzystanie planera TFD (Temporal Fast Downward) [37]. Nie jest celem niniejszej pracy, aby omówione zostały techniczne aspekty jego działania. Przedstawione jednak zostaną wspierane przez niego możliwości języka PDDL, które zdecydowały o wyborze tego planera do wykonania projektu.

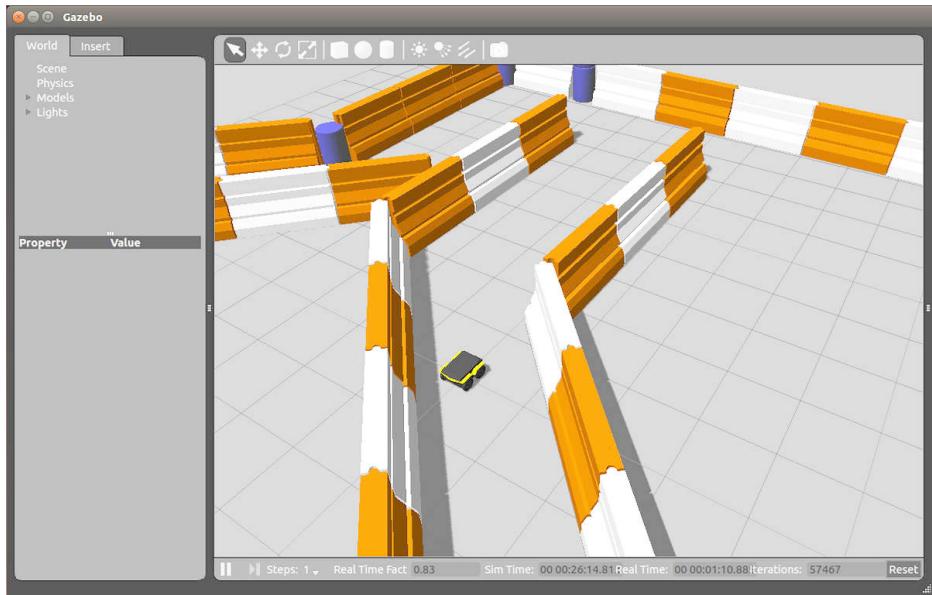
- Wykorzystanie akcji o niezerowym czasie trwania (*durative-actions*).
- Opis funkcji.
- Alternatywa logiczna zapisana wśród warunków (*conditions*) akcji.

Jest to jedyny planer (wśród zestawionych powyżej), który wspiera przedstawione cechy języka PDDL. Znacznie ułatwiają one prace związane z tworzeniem modeli PDDL i będą szeroko wykorzystywane w części implementacyjnej pracy (rozdział 5). Planer TFD został także wybrany ze względu na to, że należy on do grupy planerów domyślnie wspieranych przez jedno z wykorzystanych narzędzi omówionych później w rozdziale 3. Jest on także łatwy w integracji - jedną wymaganą czynnością potrzebną do rozpoczęcia pracy z planerem TFD jest jego pobranie z odpowiedniej strony internetowej [36] (jest to wolne oprogramowanie udostępnione na licencji GNU) oraz komplikacja.

3. Robot TIAGo i narzędzia związane z systemem ROS

3.1. Oprogramowanie Gazebo i Blender

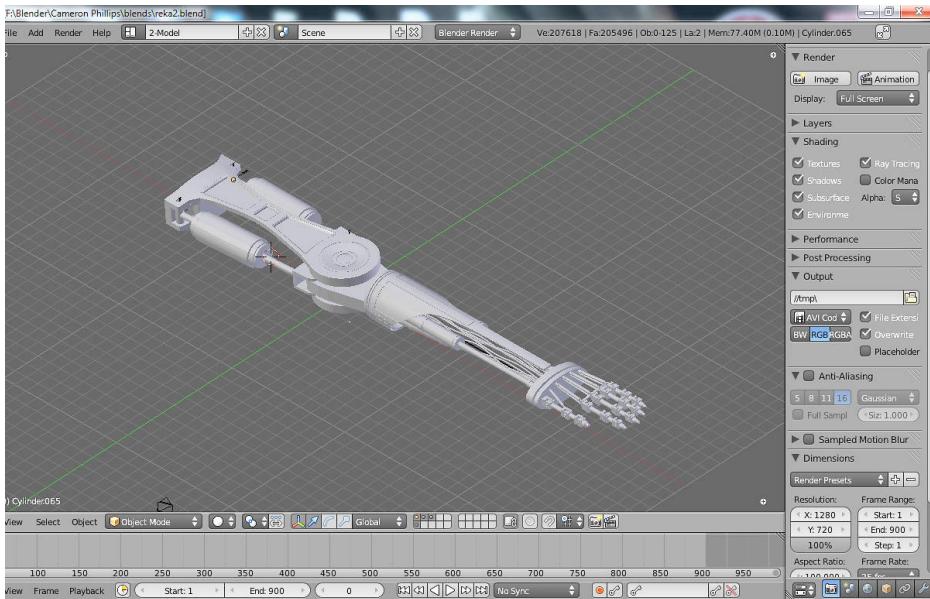
Oprogramowanie Gazebo jest narzędziem pozwalającym na przeprowadzanie trójwymiarowych symulacji urządzeń robotyki [38]. Może ono wykorzystywać kilka różnych silników do obliczeń fizyki, między innymi domyślny silnik ODE (Open Dynamics Engine) [39]. Do renderowania scen Gazebo wykorzystuje biblioteki OpenGL [40]. Zakres tej pracy nie obejmuje opisu struktury tego oprogramowania. Warto jednak wspomnieć, że ma ono duże możliwości, między innymi w zakresie symulacji sensoryki robotów. Mogą to być skanery laserowe, kamery 2D/3D, sensory głębi, czujniki sił, momentów i inne. Należy także zaznaczyć, że Gazebo jest programem darmowym. Na rysunku 7 przedstawiono zrzut ekranu z otwartym głównym jego oknem.



Rysunek 7. Okno główne programu Gazebo [38]

Gazebo będzie szeroko wykorzystywany w niniejszej pracy podczas przeprowadzania wszelkich symulacji robota w wirtualnym laboratorium. Samo wirtualne pomieszczenie zbudowane będzie z wielu przygotowanych wcześniej za pomocą programu Blender [42] (rysunek 8) modeli obiektów. Blender jest wolnym oprogramowaniem o ogromnych możliwościach. Jednak tak jak wspomniano, posłuży on w tej pracy jedynie do zamodelowania niektórych obiektów znajdujących się w laboratorium. Głównym powodem, dla którego wybrano go do tego zadania jest możliwość eksportowania modeli do plików w formacie *.dae (Digital Asset Exchange). Jest to istotne z uwagi na fakt, że jest to jeden z niewielu formatów kompatybilnych z Gazebo. Poza tym praca z wykorzystaniem środowiska Blender cechuje się dużą wygodą i swobodą.

3. Robot TIAGo i narzędzia związane z systemem ROS



Rysunek 8. Okno główne programu Blender [41]

3.2. Platforma robota

Platforma wykorzystywana w realizacji omawianej pracy to robot TIAGo [43] wyprodukowany przez hiszpańską firmę PAL Robotics [44]. Powstała ona około roku 2004 i ma siedzibę zlokalizowaną w Barcelonie w Hiszpanii. Poza TIAGo firma posiada w ofercie kilka innych robotów mobilnych (w tym humanoidalnych) oraz platformy znajdujące zastosowanie w przemyśle.

Omawiany robot według oferty producenta może być wyposażony w różne warianty manipulatora lub inne akcesoria (na przykład uchwyt na tablet). Egzemplarz wykorzystywany w niniejszej pracy został wyposażony właśnie w uchwyt umożliwiający montaż tabletu dotykowego oraz znajdującej się pod nim antenę RFID (rysunek 9).

3.2.1. Architektura sprzętowa

Konstrukcja

Na rysunku 9 przedstawiono robota ze wskazaniem na kluczowe elementy oraz przedstawiono w tabeli podstawowe parametry konstrukcyjne.

Jak widać konstrukcję robota podzielić można na trzy elementy: bazę jazdną, korpus oraz głowę. Dalej przedstawiono podstawowe parametry omawianej platformy (tabela 1).

Warto również wspomnieć o wyświetlaczu zamontowanym z tyłu podstawy jazdnej informującym o stopniu rozładowania akumulatora oraz o wciśniętym przycisku bezpieczeństwa.

Komputer pokładowy

Przedstawiono poniżej zestawienie podstawowych parametrów komputera pokładowego robota (tabela 2).



Rysunek 9. Robot TIAGo wykorzystywany w omawianym projekcie

Konstrukcja	Wysokość	110-145cm
	Waga	60kg
	Średnica podstawy	54cm
Stopnie swobody	Baza jezdna	2
	Korpus	1
	Głowa	2
Baza jezdna	Rodzaj napędu	Różnicowy
	Maksymalna prędkość	1 m/s
Korpus	Zakres zmiany wysokości	35cm

Tabela 1. Podstawowe parametry konstrukcyjne robota TIAGo

Sensoryka

W tabeli 3 zestawiono czujniki, w które wyposażony został robot. Kluczowym sensorem jest skaner laserowy - dane z niego pochodzące stanowią główne źródło informacji o otoczeniu podczas budowania mapy i nawigacji.

3. Robot TIAGo i narzędzia związane z systemem ROS

Procesor	Intel i7 Haswell
RAM	16GB
Dysk twardy	512GB SSD
Wi-Fi	802.11 a/b/g/n/ac
Bluetooth	Smart 4.0

Tabela 2. Podstawowe parametry komputera pokładowego robota

Baza jezdna	Skaner laserowy	SICK TIM651
	Sonary	SFR05
	IMU	MPU-6050
Korpus	Mikrofon	Andrea Electronics SuperBeam
	Antena RFID	
Głowa	Kamera RGB-D	Orbbec Astra

Tabela 3. Lista czujników dostępnych na platformie robota

Akumulator

Źródło energii robota stanowi akumulator Li-Ion o pojemności $20Ah$ i napięciu nominalnym równym $36V$. W pełni naładowany wystarcza na maksymalnie 8 godzin pracy [43].

3.2.2. Oprogramowanie

Warstwa systemu operacyjnego

Jak widać na rysunku 10, obecne są dwa główne bloki oprogramowania. Pierwszym z nich jest system operacyjny Ubuntu z nakładką o nazwie Xenomai, dzięki której możliwe jest uzyskanie możliwości systemu czasu rzeczywistego. Drugi blok stanowi właściwe oprogramowanie robota oparte na systemie Orocó. Dzięki niemu komunikacja między procesami może zachodzić w czasie rzeczywistym.

Warstwa systemu ROS

Całe oprogramowanie sterujące robotem TIAGo oparte jest na systemie ROS (objaśnionym szerzej w rozdziale 3.3). Wszystkie należące do niego paczki podzielić można na trzy grupy:

- paczki pochodzące oficjalnie z używanej dystrybucji ROS kinetic.
- paczki rozwijane przez producenta robota TIAGo (PAL Robotics) oznaczone nazwą wersji *erbium*.
- paczki rozwijane przez użytkownika robota.

Przedstawione powyżej grupy oprogramowania zainstalowane są na dysku SSD komputera pokładowego w różnych katalogach (tak jak przedstawiono na rysunku 11). Dwie pierwsze grupy zainstalowane są w obszarze dostępnym tylko do odczytu. Oczywiście możliwe jest zamontowanie tego obszaru w trybie pozwalającym na modyfikację tej części oprogramowania, ale wiąże się to z ryzykiem jego nieodwracalnego uszkodzenia. Lepszym

- Operating system:

Ubuntu LTS 64-bit



- Robotics middleware:

RT Preempt

ROS LTS



PAL distribution → ROS packages developed by PAL Robotics

Rysunek 10. Elementy składające się na oprogramowanie robota TIAGO

rozwiązaniem jest zastosowanie się w tej kwestii do zaleceń producenta i skorzystanie z jego narzędzia do instalacji oprogramowania. Pozwala ono na wprowadzanie zmian w oprogramowaniu zarówno użytkownika jak i producenta. Jest to realizowane w sposób w pełni bezpieczny, ponieważ pomimo zmian oryginalna postać paczek oprogramowania jest łatwo przywracalna.

- ROS Indigo packages:

Installation path: `/opt/ros/indigo`

Sub-folder	Description
<code>bin</code>	nodes (executables)
<code>include</code>	package header files
<code>lib</code>	package dynamic libraries and python files
<code>share</code>	packages <code>cmake</code> , <code>launch</code> and config files
<code>etc</code>	other files

- PAL dubnium packages:

Installation path: `/opt/pal/dubnium`

- User packages:

Installation path: `/home/pal/deployed_ws`

Rysunek 11. Struktura oprogramowania sterującego robotem TIAGO

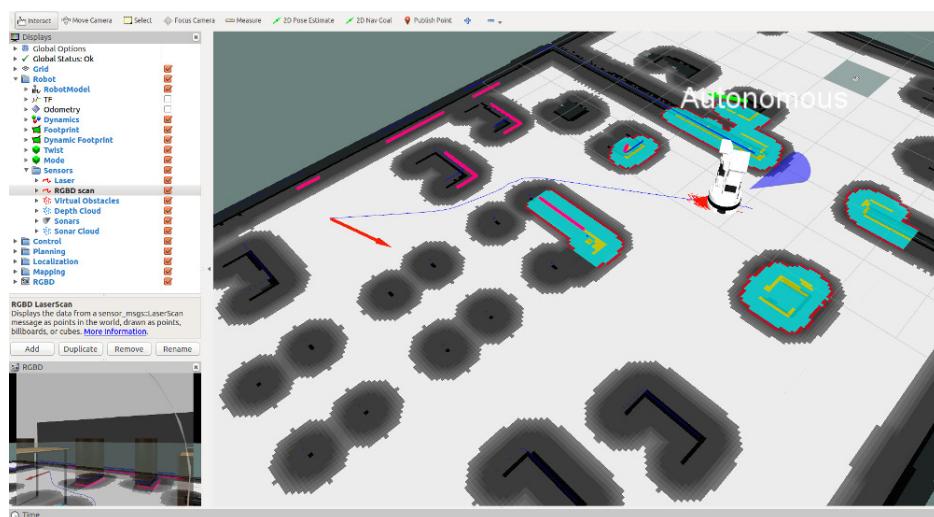
3.2.3. Symulator

Producenci robota TIAGO udostępniają za darmo jego symulator w środowisku Gazebo (rozdział 3.1). Jest to duży ukłon w stronę użytkowników robota, ponieważ symulacja jego działań jest kluczowa podczas pracy nad jakimkolwiek projektem z nim związanym. Symulator TIAGO obsługuje większość oprogramowania sterującego rzeczywistym robotem, wliczając przede wszystkim właściwości mechaniczne robota oraz nawigację i sensorykę (rysunek 12). Dzięki temu możliwe jest uruchomienie zewnętrznej aplikacji sterującej

3. Robot TIAGo i narzędzia związane z systemem ROS

na symulatorze i następnie przeniesienie jej (praktycznie bez żadnych modyfikacji) na prawdziwego robota.

Dzięki symulacji możliwe jest również rozpoczęcie pracy nad oprogramowaniem robota bez jego rzeczywistej obecności. Również prace opisane w części implementacyjnej zostały rozpoczęte bez udziału robota. Warto zwrócić uwagę, że korzystanie z symulatora podczas implementacji niniejszej pracy jest szczególnie ważne, ponieważ bardzo często robot pozostaje w bliskim kontakcie z człowiekiem. Dzięki temu podjęte zostały podstawowe kroki ku zachowaniu bezpieczeństwa.



Rysunek 12. Symulator robota TIAGo w oprogramowaniu Gazebo [38]

3.3. Robot Operating System

3.3.1. Charakterystyka ROS

Robot Operating System (ROS) [45] jest system pozwalającym na rozwijanie oprogramowania robotycznego. Jest to zestaw narzędzi, bibliotek i norm, które mają ułatwiać prace nad trudnymi i złożonymi aplikacjami związanymi z robotyką. Roboty są na tyle skomplikowanymi urządzeniami, że tworzenie oprogramowania na te platformy jest bardzo trudne. Na jego całość składa się wiele zagadnień związanych z mechaniką, sensoryką, elektroniką, nawigacją itp. Napisanie zatem gotowego oprogramowania sterującego jednym robotem od zera byłoby niesamowicie czasochłonne i wymagające wiedzy specjalistów z wielu dziedzin. Jednak dzięki narzędziu ROS, które łączy pracę wielu ludzi i pomaga im w realizacji swoich prac staje się to możliwe nawet dla jednej osoby. Specjalisci mogą dzielić się swoją pracą z innymi, dzięki czemu istnieją już gotowe biblioteki w systemie, które stanowią wiele gotowych rozwiązań z dziedziny robotyki, takich jak nawigacja czy sterowanie. Dzięki dużej społeczności pojawia się coraz więcej gotowych rozwiązań, a ich weryfikacja również postępuje.

3.3.2. Opis istotnych elementów systemu

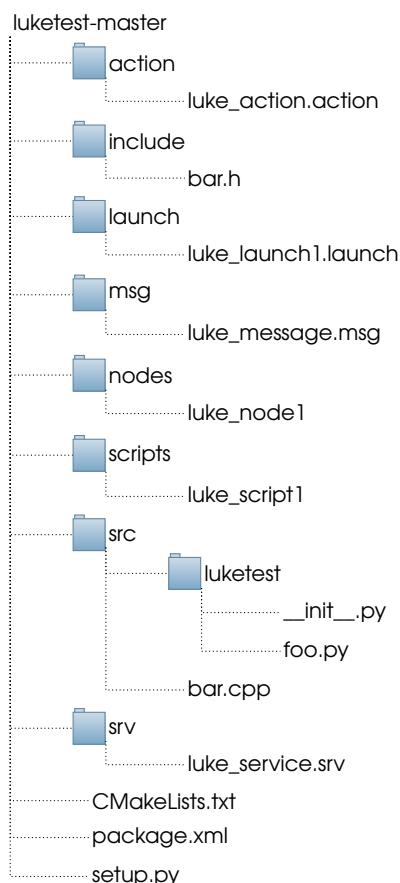
Ideę systemu ROS można podzielić na trzy poziomy:

- struktura plików
- graf przetwarzania danych
- społeczność

Są to kluczowe elementy całego systemu i zostaną poniżej w skrócie opisane. Opis ten jest przygotowany pod kątem niniejszej pracy oraz nawiązuje do wykorzystania omawianych elementów w części implementacyjnej. Zostanie on również rozszerzony o komentarze związane z elementami systemu ROS, które nie należą do przedstawionego powyżej podziału, a jednak stanowią ważną część tej pracy.

Struktura plików

W części implementacyjnej omawianej pracy utworzono kilka komponentów systemu ROS. Poniżej przedstawiono przykładową strukturę takiego komponentu (rysunek 13) wraz z nazewnictwem i krótkim objaśnieniem poszczególnych elementów.



Rysunek 13. Struktura przykładowej paczki ROS

3. Robot TIAGo i narzędzia związane z systemem ROS

Zaimplementowane paczki (komponenty) systemu mogą zawierać:

— **pliki źródłowe**

zawierają implementację węzłów komponentu oraz metod, które mogą zostać wykorzystane w innych częściach systemu. Jeśli jest to implementacja węzła w języku C++ to plik przechowywany jest w *[package_name]/src/*. Jeśli węzeł napisany jest w języku Python to plik powinien znajdować się w *[package_name]/nodes/*. Metody wykorzystywane w bieżącej paczce (i nie tylko) przechowywane są w *[package_name]/src/[package_name]/*.

— **pliki konfiguracyjne**

pliki zawierające konfigurację parametrów lub wartości zmiennych wczytywanych podczas pracy systemu. Przechowywane w *[package_name]/config/*.

— **pliki startowe**

zawierają instrukcje pozwalające na uruchamianie zespołów programów wraz z odpowiednimi parametrami w ustalonej konfiguracji. Umożliwiają uruchamianie bardziej rozbudowanych paczek. Przechowywane w *[package_name]/launch/*.

— **definicje wiadomości**

opisują strukturę definiowanej wiadomości (nośnika danych). Przechowywane w *[package_name]/msg/*.

— **definicje serwisów**

opisują strukturę danych potrzebnych do wywołania serwisu oraz jego odpowiedzi (plik podzielony jest na dwie części). Przechowywane w *[package_name]/srv/*.

— **definicje akcji**

opisują strukturę danych potrzebnych do obsługi akcji. Podobnie jak w przypadku definicji serwisu, tu jednak plik podzielony jest na trzy części:

— żądanie

— wiadomości zwrotne

— wynik

Przechowywane w *[package_name]/action/*.

— **nagłówki**

nagłówki plików źródłowych. Przechowywane w *[package_name]/include/*.

— **plik kompilacyjny**

opis struktury paczki, zmiennych środowiskowych i konfiguracja niezbędna do jej komplikacji. Przechowywany w pliku *[package_name]/CMakeLists.txt*.

— **plik manifestowy**

Informacje ogólne o paczce i jej konfiguracji (między innymi autor, zależności). Przechowywany w pliku *[package_name]/package.xml*.

Graf przetwarzania danych i komunikacja

Jest to sieć *peer-to-peer* procesów wspólnie przetwarzających dane. Poniżej przedsta-

wiono elementy należące do grafu przetwarzania danych, które na różne sposoby przetwarzają i dostarczają do niego dane.

— **master**

dostarcza informacje o nazwach oraz wgląd do reszty systemu. Dzięki niemu poszczególne części systemu (takie jak węzły) posiadają informację o istnieniu pozostałych i mogą wymieniać ze sobą wiadomości i wywoływać serwisy.

— **serwer parametrów**

umożliwia zapis i odczyt parametrów. Elementy systemu mogą je wyszukiwać po kluczu i wykorzystywać do przetwarzania danych.

— **węzły**

są podstawowymi jednostkami systemu przetwarzającymi dane. W praktyce węzeł jest procesem o ścisłe określonej funkcji (np. przetwarzanie danych z czujnika lub planowanie ścieżki). Dzięki takiemu rozwiązanemu całym system pozostaje modularny, ponieważ składa się z wielu takich odrębnych procesów (węzłów).

— **wiadomości**

za ich pomocą komunikują się elementy systemu ROS. Są to struktury danych, które składać się mogą z pól standardowych typów (integer, float, boolean itd.), z ich macierzy oraz zagnieżdzonych struktur i tablic.

— **tematy**

są *tunelami*, wewnętrznych których przesyłane są wiadomości. Odbiera się w to w konwencji *publikuj-subskrybuj* - aby na danym temacie pojawiły się wiadomości, muszą one być publikowane przez jeden lub więcej procesów. Węzły, które mają odczytywać wiadomości z tematu muszą być do niego zasubskrybowane. Węzły, które publikują na danym temacie oraz te, które są do niego zasubskrybowane nie posiadają informacji o swoim istnieniu. Do identyfikacji tematu służy jego nazwa.

— **serwisy**

podobnie jak tematy umożliwiają komunikację elementów systemu ROS. Różnica polega na tym, że tu komunikacja odbywa się w konwencji *żądanie-odpowiedź*. Serwis jest zatem ustanowiony przez **jeden** węzeł i również tylko **jeden** węzeł jednocześnie może żądać odpowiedzi. Struktura wiadomości podzielona jest zatem na dwie części, z których jedna stanowi opis żądania, a druga odpowiedzi.

— **bagi**

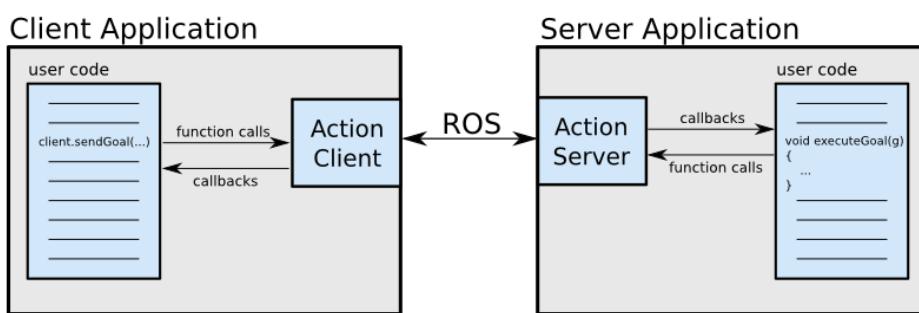
są ważnym mechanizmem służącym do zapisywania przesyłanych danych. Odpowiednio skonfigurowane umożliwiają odtworzenie pracy całego systemu, co w wielu przypadkach daje szansę na znalezienie błędu lub odczyt nieuchwytnych w inny sposób wiadomości.

W ramach omawiania podstawowych elementów systemu warto wspomnieć o paczce **actionlib** (rozdział 3.4). Znajdziemy na szerokie zastosowanie w części implementacyjnej pracy co tym bardziej motywuje do jej przybliżenia.

3.4. Paczka actionlib

W wielu przypadkach oczekuje się możliwości wysłania żądania do węzła, by ten wykonał jakąś operację. Dodatkowo po jej wykonaniu węzeł odpowiada wiadomością o zdefiniowanej wcześniej strukturze. Taki mechanizm uzyskać można wykorzystując wspomniane już serwisy (omówione w rozdziale 3.3.2).

Są jednak operacje, których wykonanie może pochłonąć dużo czasu, podczas którego pomocne byłyby dodatkowe wiadomości zwrotne od węzła realizującego zadanie. Takie możliwości daje właśnie paczka *actionlib*, która dostarcza wszelkie niezbędne narzędzia do ich realizacji. Oprócz tego wykonywane zadania mogą być w dowolnym momencie przerwane. Przy wykorzystaniu paczki tworzona jest struktura *klient-serwer* (rysunek 14), co upraszcza całość systemu i ułatwia jej wykorzystanie.



Rysunek 14. Schemat działania biblioteki *actionlib* [15]

Klient jest aplikacją, która wysyła żądanie (w postaci wiadomości zawierającej cel) wykonania akcji do serwera. Ten z kolei jest aplikacją realizującą zadanie. Serwer może wysyłać wiele wiadomości zwrotnych do klienta (jednak nie jest to konieczne) oraz na końcu wysyła ostatnią wiadomość nazywaną wynikiem. Otrzymanie tej wiadomości przez klienta równoznaczne jest z zakończeniem wykonywania zadania przez serwer.

3.4.1. Komunikacja

Aby klient i serwer mogły się ze sobą komunikować należy zdefiniować wiadomości (rozdział 3.3.2), które będą w tym celu wykorzystane. Należyty opis powinien określać trzy rodzaje wiadomości:

— cel

jest wiadomością, którą klient wysyła do serwera i zawiera dane niezbędne do realizacji danego zadania. Przykładowo jeśli akcja *actionlib* polega na przejeździe platformy mobilnej to wiadomość **cel** powinna opisywać docelową pozycję robota (i być typu np. PoseStamped).

— wiadomość zwrotna

pełni funkcję powiadamiania klienta przez serwer o stopniu w jakim zadanie jest aktualnie wykonane. Jeśli jest ono związane z przejazdem robota do zadanego punktu

Listing 8. Przykład pliku *.action* [15]

```
# Definicja celu
uint32 dishwasher_id
___

# Definicja wyniku
uint32 total_dishes_cleaned
___

# Definicja wiadomości zwrotnej
float32 percent_complete
```

to wiadomość może zawierać na przykład aktualną pozycję robota lub odległość jaka jeszcze mu pozostała do pokonania.

— **wynik**

jest ostatnią wiadomością wyslaną (tylko raz) przez serwer do klienta. Różni się tym od wiadomości zwrotnych, że może zawierać ważne informacje związane z ideą samej akcji *actionlib*. W omawianym przypadku poruszania platformą mobilną wiadomość ta może zawierać tę samą informację co wiadomość zwrotna, ale jeśli celem zadania jest zebranie pewnych danych z czujnika to właśnie one mogą się w tej wiadomości znaleźć.

3.4.2. Plik z opisem akcji *actionlib*

Akcja *actionlib* opisana jest w pliku z rozszerzeniem *.action*. To właśnie tu zawarta jest definicja opisanych wcześniej wiadomości celu, zwrotnych i wyniku (rozdział 3.4.1). Każda z tych części oddzielona jest linijką z trzema myślnikami (- - -), tak jak pokazano na przykładzie (listing 8).

Pliki z odpowiednią zawartością powinny znaleźć się w katalogu paczki w folderze *action* (np. */luketest-master/action/DoDishes.action* - plik, którego zawartość została wylistowana w listingu 8).

3.4.3. Uproszczona implementacja

Paczka *actionlib* zawiera biblioteki pozwalające na wykorzystanie klas *SimpleActionClient* i *SimpleActionServer*. Dzięki nim możliwe jest szybkie uruchomienie prostych węzłów ze współpracującymi klientem i serwerem *actionlib*. Prawdą jest, że przez wykorzystanie wspomnianych klas nie są w pełni wykorzystane możliwości paczki *actioblib*. Ograniczenia te jednak (opisane szerzej poniżej) w żadnym stopniu nie ujmują możliwości systemu opisanego w części implementacyjnej (rozdział 5) niniejszej pracy.

Na listingu 9 przedstawiono przykład implementacji klienta *actionlib* w języku C++. Jest on oparty o akcję *actionlib* zdefiniowaną w pliku *DoDishes.action* (listing 8). Na jego podstawie wygenerowane zostały w procesie komplikacji odpowiednie wiadomości,

Listing 9. Przykład implementacji klienta w języku C++ [15]

```
#include <chores/DoDishesAction.h> // Note: "Action" is appended
#include <actionlib/client/simple_action_client.h>

typedef actionlib::SimpleActionClient<chores::DoDishesAction> Client;

int main(int argc, char** argv)
{
    ros::init(argc, argv, "do_dishes_client");
    Client client("do_dishes", true); // true -> don't need ros::spin()
    client.waitForServer();
    chores::DoDishesGoal goal;
    // Fill in goal here
    client.sendGoal(goal);
    client.waitForResult(ros::Duration(5.0));
    if (client.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
        printf("Yay!_The_dishes_are_now_clean");
    printf("Current_State:_%s\n", client.getState().toString().c_str());
    return 0;
}
```

które pozwalają na zdefiniowanie celu (obiekt klasy *chores::DoDishesGoal*). Cel ten jest następnie wysłany do serwera (listing 10) za pomocą stosownej metody.

Z kolei na listingu 10 przedstawiono przykład implementacji serwera zdefiniowanej wcześniej akcji *actionlib*. Zauważać można implementację metody *execute()*, która wykona się po poprawnym odebraniu celu od klienta. Oczywiście najpierw uruchomiony musi zostać sam serwer - utworzony jest odpowiedni obiekt *server*, po czym wywołana zostaje jego metoda *start()*. Od tej chwili serwer *actionlib* pozostaje aktywny i czeka na cel pochodzący od klienta. Wewnątrz metody *execute()* znaleźć się może implementacja wykonania pewnego zadania wraz z obsługą wysyłania wiadomości zwrotnych oraz wyniku (w przykładzie przedstawiono tylko to ostatnie).

Podsumowanie i reguły

Przedstawiono przykładowe wykorzystanie klas *SimpleActionClient* i *SimpleActionServer*, co stanowi nałożenie pewnego ograniczenia na możliwości paczki *actionlib*. Ograniczenie to polega na obsłudze tylko jednego celu jednocześnie przez parę klient-serwer. Poniżej wypunktowano szczegółowe informacje [15] związane z tą cechą:

- Tylko jeden cel może być w danej chwili aktywny.
- Nowy cel przerywa poprzedni opierając się na czasie zapisanym w polu *GoalID* (jest to zazwyczaj czas wysłania celu).
- Istnieje możliwość wysłania żądania przerwania celu. Żądanie to przerywa wszystkie cele, których czas w polu *GoalID* jest mniejszy lub równy czasowi żądania.

Listing 10. Przykład implementacji serwera w języku C++ [15]

```
#include <chores/DoDishesAction.h> // Note: "Action" is appended
#include <actionlib/server/simple_action_server.h>

typedef actionlib::SimpleActionServer<chores::DoDishesAction> Server;

void execute(const chores::DoDishesGoalConstPtr& goal, Server* as)
// Note: "Action" is not appended to DoDishes here
{
    // Do lots of awesome groundbreaking robot stuff here
    as->setSucceeded();
}

int main(int argc, char** argv)
{
    ros::init(argc, argv, "do_dishes_server");
    ros::NodeHandle n;
    Server server(n, "do_dishes", boost::bind(&execute, 1, &server), false);
    server.start();
    ros::spin();
    return 0;
}
```

- Akceptacja nowego celu implikuje prawidłowe przerwanie celu poprzedniego. Status tego celu jest na tej podstawie automatycznie aktualizowany.

3.5. Paczka SMACH

SMACH [46] jest opartą na języku Python biblioteką pozwalającą na prototypowanie deterministycznych automatów skończonych (DAS). Wszelkie zadania, które mogą być przedstawione w postaci automatów DAS mogą być zatem bardzo szybko zaprogramowane z wykorzystaniem paczki SMACH. Automaty prototypowane z wykorzystaniem tej paczki mogą mieć bardzo złożoną postać - w szczególności mogą być tworami złożonymi z wielu mniejszych automatów skończonych (zagnieżdżanie stanów) [46].

Na rysunku 15 przedstawiono przykładowy automat zaprogramowany dzięki SMACH.

3.5.1. Przykład implementacji

Na listingu 11 przedstawiono przykład implementacji pojedynczego stanu o nazwie *Foo*. Jak można zauważyć każdy stan automatu jest implementowany przy użyciu odzielnej klasy, która dziedziczy po klasie *smach.State* reprezentującej generyczny stan. Natomiast listing 12 przedstawia przykład implementacji całego automatu, na którego skład wchodzą dwa stany: *Foo* oraz *Bar*. Za pomocą słownika *transitions* możliwe jest definiowanie przejść wewnątrz danego automatu. Zarówno na listingu jak i na rysunku 15 widać, że stan *Foo* zakończyć się może słowem *outcome1* lub *outcome2*. Stan *Bar* zakoń-

Listing 11. Przykład implementacji jednego ze stanów automatu SMACH [46]

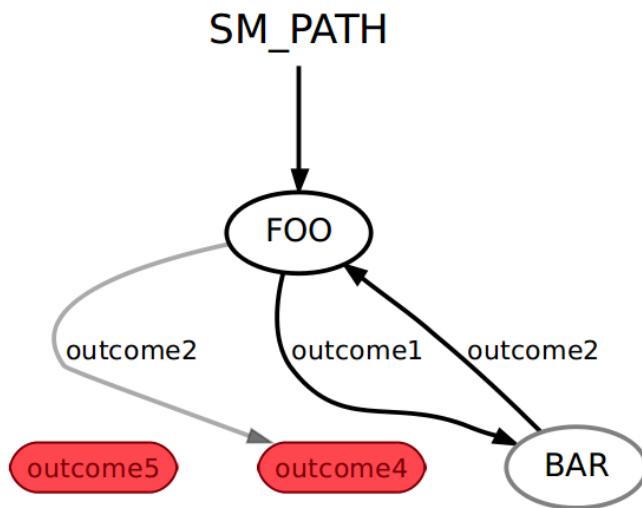
```
class Foo(smach.State):
    def __init__(self, outcomes=['outcome1', 'outcome2']):
        # Your state initialization goes here

    def execute(self, userdata):
        # Your state execution goes here
        if xxxx:
            return 'outcome1'
        else:
            return 'outcome2'
```

Listing 12. Przykład implementacji automatu SMACH [46]

```
sm = smach.StateMachine(outcomes=['outcome4', 'outcome5'])
with sm:
    smach.StateMachine.add('FOO', Foo(),
                           transitions={'outcome1': 'BAR',
                                         'outcome2': 'outcome4'})
    smach.StateMachine.add('BAR', Bar(),
                           transitions={'outcome2': 'FOO'})
```

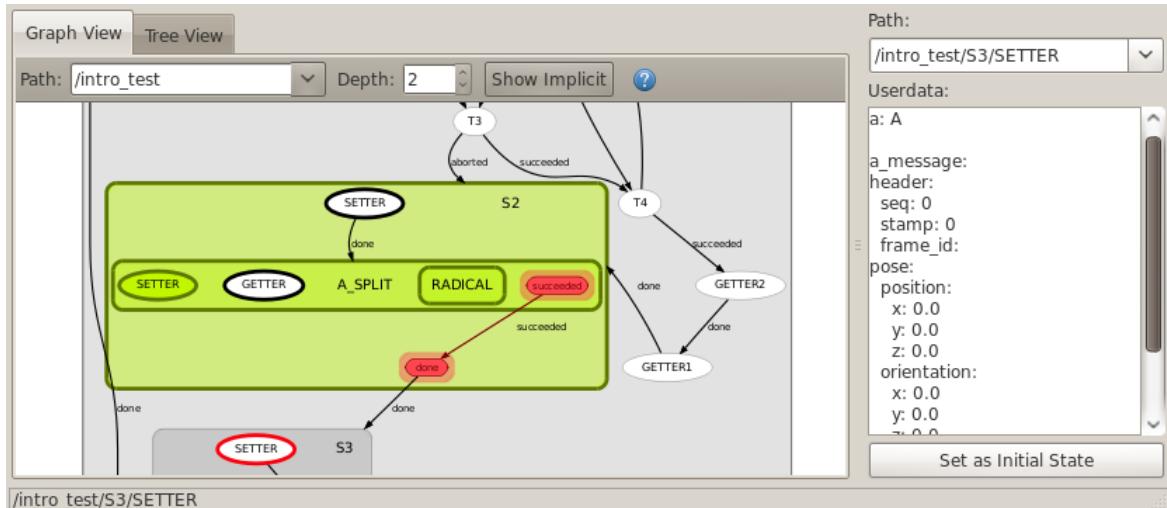
czyć się może jedynie słowem *outcome2*. Cały automat ma natomiast zdefiniowane dwa słowa wyjściowe: *outcome4* oraz *outcome5*. Przykładowa implementacja pozwala temu automatowi zakończyć się jednak tylko słowem *outcome4* (jeśli stan *Foo* zwróci *outcome2*).



Rysunek 15. Efekt wykonania przedstawionego na listingach 11 oraz 12 kodu [46]

Warto również wspomnieć o bardzo przydatnym narzędziu, które również znajduje się w paczce SMACH. Jest nim *smach_viewer* [47] - program, który pozwala na wizualizację

wybranych automatów, przejść pomiędzy stanami, podglądarkanie stanów aktywnych oraz informacji przekazywanych przez stany. Na rysunku 16 przedstawiono widok włączonego programu *smach_viewer*. Wewnątrz niego zauważać można dość rozbudowany twór złożony z kilku automatów (*A_SPLIT*, *S2*, *S3*, *RADICAL*), które zamknięte są pod postacią pojedynczych stanów głównego automatu *intro_test*.



Rysunek 16. Okno programu *smach_viewer* [47] z uruchomionym przykładowym automatem (ze stanami reprezentującymi zagnieżdżone automaty)

3.5.2. Opakowanie kontenera SMACH przez actionlib

Wśród wielu możliwości, które daje paczka SMACH w ramach swojej integracji z systemem ROS warto wspomnieć o tej, która będzie wykorzystywana w części implementacyjnej pracy. Jest nią możliwość opakowania kontenera SMACH (DAS) przez serwer akcji *actionlib* (rozdział 3.4). Oznacza to, że tak opakowany automat uruchomi się po otrzymaniu od klienta celu. Po zakończeniu pracy automat zwróci odpowiednią wiadomość do klienta informując go o tym (dokładnie tak, jak opisano to w omówieniu paczki *actionlib* (rozdział 3.4)). Przykład implementacji takiego automatu pełniącego jednocześnie funkcję serwera akcji *actionlib* przedstawiono na listingu 13.

Warto zwrócić uwagę na kluczową klasę *ActionServerWrapper*, która zapewnia całość realizacji opisywanej funkcji. Przekazane tu są między innymi: nazwa akcji *actionlib*, nazwa automatu oraz zmapowane jego słowa wyjściowe.

Oczywiście w ramach opakowania automatu możliwe jest przekazywanie zawartości celu od klienta akcji *actionlib* do automatu. Komunikacja zwrotna również jest zachowana - odpowiednie metody pozwalają na wysyłanie wiadomości zwrotnych oraz dowolnie zdefiniowanego wyniku realizacji do klienta.

Listing 13. Przykład opakowania automatu SMACH w serwer akcji *actionlib*

```
# Construct state machine
sm = StateMachine(outcomes=[ 'did_something' ,
                             'did_something_else' ,
                             'aborted' ,
                             'preempted' ])
with sm:
    ### Add states in here...
# Construct action server wrapper
asw = ActionServerWrapper(
    'my_action_server_name' , MyAction,
    wrapped_container = sm,
    succeeded_outcomes = [ 'did_something' , 'did_something_else' ] ,
    aborted_outcomes = [ 'aborted' ] ,
    preempted_outcomes = [ 'preempted' ] )
```

3.6. Szkielet aplikacyjny ROSPlan

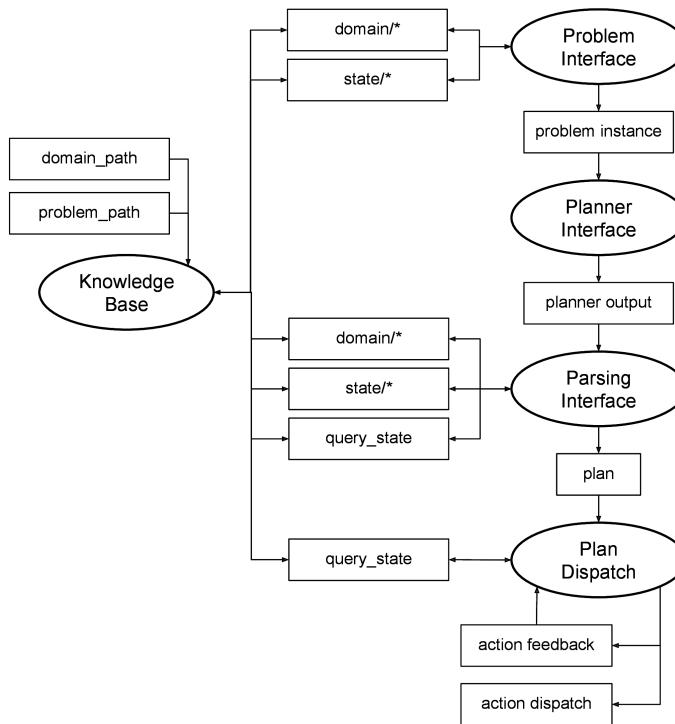
3.6.1. Omówienie

ROSPlan [48] jest szkieletem aplikacyjnym pozwalającym na wygodne wykorzystanie planowania symbolicznego (rozdział 2) w aplikacji opartej na systemie ROS. Jest to nic innego jak zestaw narzędzi (w szczególności paczek ROS), wśród których wyszczególnić można te, których zadaniem jest planowanie, generowanie problemu czy też realizacja planu. Sam szkielet ROSPlan rozwijany jest przez naukowców z uczelni King's College London [49]. Wskazują oni, że szkielet ten charakteryzuje się modularną strukturą (rozdział 3.6.2), która pozwala na wprowadzanie modyfikacji według indywidualnych potrzeb.

3.6.2. Wewnętrzna struktura i komunikacja

W strukturze szkieletu ROSPlan wyróżnić można komponenty:

- **Knowledge Base** - obsługuje przechowywanie modelu zapisanego w PDDL (rozdział 2.2).
- **Problem Interface** - generuje problem PDDL na podstawie modelu pochodzącego z komponentu Knowledge Base. Problem jest następnie publikowany na temacie ROS lub zapisywany do pliku.
- **Planner Interface** - uruchamia planer dostarczając mu wygenerowany wcześniej problem. Plan również może być opublikowany na temacie lub zapisany do pliku.
- **Parsing Interface** - wygenerowany przez planer plan jest tu parsowany do formatu wiadomości ROS tak, aby akcja po akcji mógł zostać zrealizowany przez aplikację opartą o ROS.
- **Plan Dispatch** - obsługuje realizację kolejnych akcji, na które składa się plan.



Rysunek 17. Struktura szkieletu ROSPlan [48]

Na rysunku 17 przedstawiono strukturę całego szkieletu ROSPlan. Zostaną poniżej omówione poszczególne komponenty wchodzące w jego skład oraz zależności istniejące pomiędzy nimi.

Knowledge Base

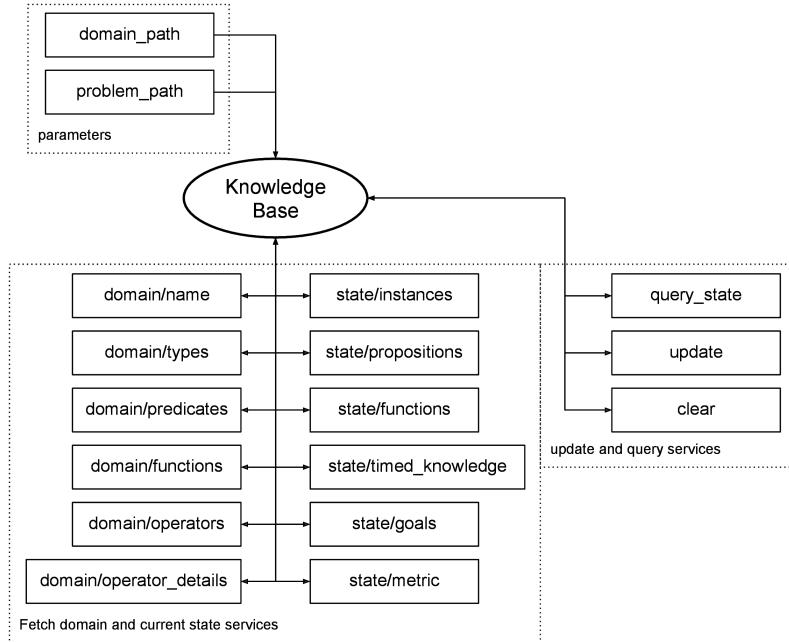
Komponent Konwledge Base (rysunek 18) jak już wcześniej wspomniano przechowuje model PDDL. Składa się on z dziedziny oraz aktualnego problemu. Podsumowując, komponent Knowledge Base:

- Wczytuje dziedzinę i problem PDDL.
- Przechowuje znany mu stan wiedzy w formacie PDDL.
- Przechowywany w nim stan wiedzy (rozdział 3.6.3) może być aktualizowany przez wiadomości ROS.
- Może być odpytywany przez zewnętrzny program w celu uzyskania wiedzy w nim przechowywanej.

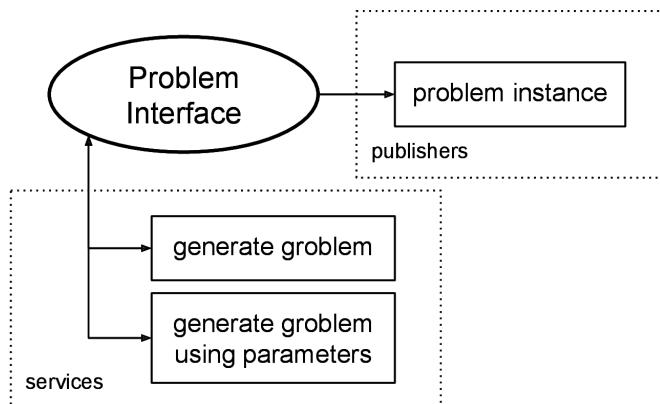
Problem Interface

Komponent Problem Interface (rysunek 19) po uruchomieniu odpytuje Knowledge Base w celu uzyskania dziedziny PDDL oraz aktualnego stanu wiedzy oraz celu. Na podstawie

3. Robot TIAGo i narzędzia związane z systemem ROS



Rysunek 18. Struktura komponentu Knowledge Base szkieletu ROSPlan [48]

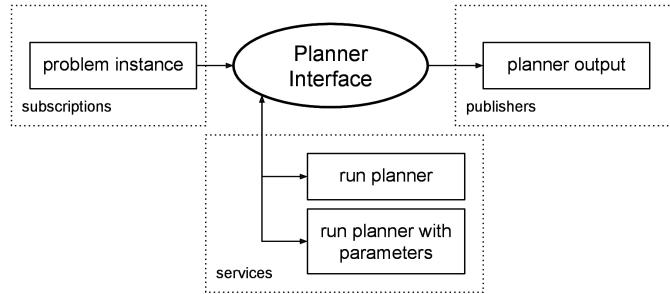


Rysunek 19. Struktura komponentu Problem Interface szkieletu ROSPlan [48]

uzyskanych danych (przy pomocy serwisów ROS) generuje właściwy problem PDDL i publikuje go na temat i/lub zapisuje do pliku.

Planner Interface

Komponent ten opakowuje wybrany planer planowania symbolicznego. Planer jest wywoływany przez serwis ROS, następnie zwraca odpowiednią wartość jeśli planowanie zakończyło się sukcesem i wygenerowany został plan. Komponent Planner Interface (rysunek 20) dostarcza planerowi dziedzinę oraz aktualny problem PDDL (wygenerowany wcześniej i opublikowany przez komponent Problem Interface). Wygenerowany plan jest publikowany na temat i/lub zapisywany do pliku.



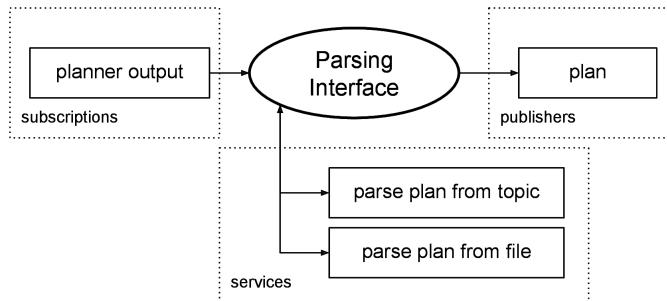
Rysunek 20. Struktura komponentu Planner Interface szkieletu ROSPlan [48]

Nazwa węzła	Kompatybilny planer	Strona internetowa
popf_planner_interface	POPF	nms.kcl.ac.uk/planning/software/popf
ff_planner_interface	FF, Metric-FF, Contingent-FF	fai.cs.uni-saarland.de/hoffmann/ff
lpg_planner_interface	LPG	lpg.unibs.it/lpg
tfd_planner_interface	TFD	gki.informatik.uni-freiburg.de/tools/tfd
smt_planner_interface	SMTPlan	kcl-planning.github.io/SMTPlan

Tabela 4. Wspierane przez szkielet ROSPlan planery

W tabeli 4 przedstawiono listę wspieranych aktualnie przez Planner Interface planerów PDDL.

Parsing Interface



Rysunek 21. Struktura komponentu Parsing Interface szkieletu ROSPlan [48]

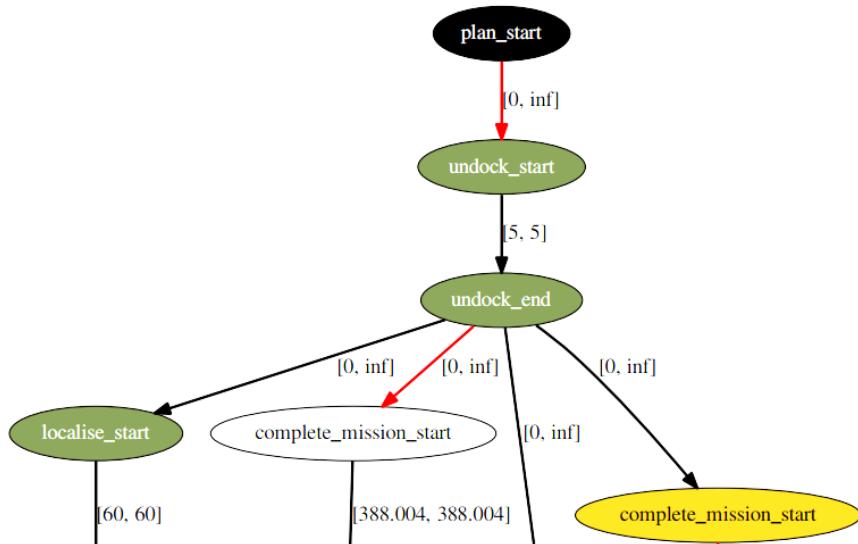
Komponent Parsing Interface (rysunek 21) konwertuje plan wygenerowany przez Planner Interface do postaci specjalnych wiadomości ROS. Wiadomości te są taką reprezentacją planu, która ma ułatwić jego realizację. Warto również wspomnieć, że komponent ten umożliwia aktualnie parsowanie do dwóch reprezentacji planu:

- **Plan podstawowy**, którego postać jest tablicą złożoną z kolejnych elementów typu *rosplan_dispatch_msgs/ActionDispatch* opisujących w podstawowy sposób konkretną akcję PDDL. Wiadomości te są ułożone w kolejności, która odpowiada wygenerowanemu planowi. Taka reprezentacja planu jest wystarczająca w sytuacjach, gdzie

3. Robot TIAGo i narzędzia związane z systemem ROS

akcje danego planu mają wykonywać się jedna po drugiej (sekwencko). Jednak w przypadkach, w których niektóre akcje mają z założenia wykonywać się równolegle potrzebne są dodatkowe informacje związane z konkretnym czasem i warunkami wykonania akcji. Wtedy należy korzystać z reprezentacji planu Esterel.

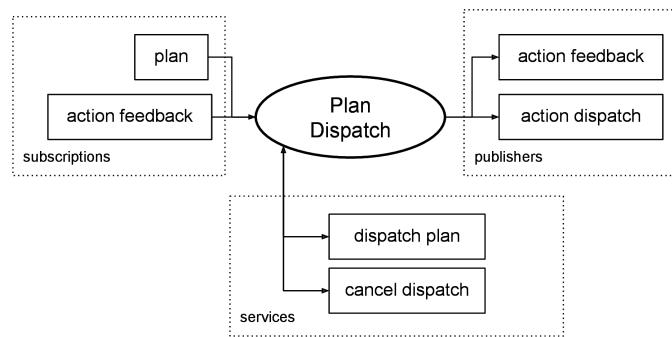
- **Plan Esterel** [50] w przeciwnieństwie do planu podstawowego jest **grafem** elementów typu *rosplan_dispatch_msgs/ActionDispatch*. Każdy węzeł tego grafu jest (poza węzłem startu planu) początkiem akcji lub jej końcem. Węzły mogą mieć status oczekiwania, wykonywania lub zakończenia wykonywania. Węzeł reprezentujący początek akcji zmienia status na wykonany gdy dana akcja zostanie zlecona do realizacji. Natomiast węzeł reprezentujący koniec akcji zmieni swój status gdy realizacja akcji zakończy się i zwróci odpowiednią informację. Krawędzie grafu reprezentują zależności pomiędzy węzłami (początkami i zakończeniami akcji), tzn. dana akcja (węzeł reprezentujący jej początek) może zostać rozpoczęta wtedy i tylko wtedy gdy wszystkie węzły reprezentujące poprzednie akcje zostały poprawnie zakończone. Na rysunku 22 zauważyc można węzły reprezentujące początki i końce akcji PDDL oraz krawędzie wskazujące zależności między nimi.



Rysunek 22. Graficzne przedstawienie planu Esterel. Zauważyc można węzły ze statusem zakończenia wykonania (kolor zielony), węzły aktualnie wykonywane (kolor żółty) oraz węzły oczekujące (kolor biały). Poza tym warto zwrócić uwagę na krawędzie, które sugerują, że akcja aktualnie wykonywana (*complete_mission*) wymagała do swojego rozpoczęcia zakończenia akcji *undock*

Plan Dispatch oraz Action Interface

Wśród zadań jakie realizuje komponent Plan Dispatch (rysunek 23) należy wyróżnić przekazywanie akcji w odpowiedniej kolejności do realizacji. Dodatkowo tutaj znajduje



Rysunek 23. Struktura komponentu Plan Dispatch szkieletu ROSPlan [48]

się implementacja procesu łączenia pojedynczych akcji do procesów odpowiedzialnych za ich fizyczną realizację.

Do użytkownika szkieletu ROSPlan należy napisanie interfejsu łączącego komponent Plan Dispatch z aplikacją fizycznie realizującą konkretne akcje. Autorzy ROSPlan przedstawiają dwie metody według których można ten cel osiągnąć:

- Rozszerzenie komponentu Action Interface - jest to klasa zawierająca szablon, który można wykorzystać w celu realizacji akcji. Polega to na implementacji obecnej w tej klasie metody *concreteCallback()*, w której umieścić należy rzeczywistą obsługę wykonania akcji. W zależności od tego czy wykonanie danej akcji się powiedzie metoda ta powinna zwrócić odpowiednią wartość. Dzięki temu informacja o sukcesie wykonania lub porażce trafi do reszty systemu ROSPlan (tak jak wskazano na listingu 14) za pomocą obecnych już kanałów komunikacji. Umożliwia to także automatyczne uzupełnianie stanu wiedzy zawartej w komponencie Knowledge Base po każdym (zakończonym sukcesem) zrealizowaniu dowolnej akcji.
- Implementacja komponentu interfejsującego od zera - pozwala na wszelką dowolność w kwestii uzupełniania stanu wiedzy zawartej w Knowledge Base. W tym przypadku też nie są wiążące wszelkie nazwy parametrów czy też akcji ponieważ całość implementacji leży po stronie użytkownika.

3.6.3. Zarządzanie stanem przechowywanej wiedzy

Wiedza jest przechowywana w bazie danych zaimplementowanej w komponencie Knowledge Base. Zawiera ona wszystko co może znaleźć się w modelu PDDL (opis obiektów, funkcje, predykaty, metryki, aktualny stan oraz cel). Przechowywana jest w formacie akceptowanym przez język PDDL, a podczas pierwszego uruchomienia systemu (podczas wczytania pliku z problemem) uzupełniane są jej pierwsze instancje. Wracając zatem do przykładu wieży Hanoi (warto spojrzeć na listing 4 przedstawiający problem) wczytanie tego modelu przez ROSPlan spowoduje uzupełnienie początkowego stanu wiedzy komponentu Knowledge Base o zawarte tam elementy. Od tego momentu więc komponent

Listing 14. Pseudokod przedstawiający działanie komponentu Action Interface [51].

```
IF action name matches THEN:  
    check action for malformed parameters  
    update knowledge base (at start effects)  
    publish (action enabled)  
    success = concreteCallback()  
IF success THEN:  
    update knowledge base (at end effects)  
    publish (action achieved)  
ELSE  
    publish (action failed)  
RETURN success
```

Knowledge Base poza dziedziną przechowywać będzie odpowiednio pogrupowane zdefiniowane obiekty (*(peg1)*, *(peg2)*, *(peg3)*, *(d1)*, *(d2)*, *(d3)*), wszystkie predykaty zawarte w grupie *init* problemu (początkowo znane właściwości elementów świata) oraz wszystkie cele (*(on d3 peg3)*, *(on d2 d3)*, *(on d1 d2)*).

Stan wiedzy może być poza tym uzupełniany na dwa inne sposoby (poza wczytaniem pliku problemu):

- Pozytywne zakończenie dowolnej akcji - dodawane są wtedy predykaty zapisane w efekcach tej akcji (w pliku z dziedziną).
- Za pomocą serwisów - komponent Knowledge Interface udostępnia możliwość dowolnej modyfikacji stanu wiedzy przy pomocy serwisów i odpowiednio ustrukturyzowanych wiadomości ROS [52]

Warto teraz zaznaczyć, że generowany przez komponent Problem Interface problem PDDL budowany jest na podstawie wiedzy zawartej w bazie danych Knowledge Interface. Oznacza to, że podczas pracy systemu można dowolnie modyfikować stan wiedzy, a później ponownie (na jego podstawie) wygenerować nowy problem PDDL i plan.

3.6.4. Ograniczenia

Jak już wspomniano szkielet aplikacyjny ROSPlan umożliwia wygodne wykorzystanie planowania symbolicznego w aplikacjach opartych na systemie ROS. Posiada on jednak pewne ograniczenia i wady, które przedstawione zostaną poniżej.

Pierwszą negatywną cechą jest przede wszystkim to, że jest to oprogramowanie nadal intensywnie rozwijane, przez co często można napotkać na błędy podczas korzystania z niego. Należy się więc nastawić na wyszukiwanie ich i wprowadzanie odpowiednich poprawek. Poza tym w systemie ROSPlan zaimplementowane jest obecnie wsparcie dla nielicznych planerów (tabela 4). W razie potrzeby korzystania z planera nieobecnego w tabeli 4 należy samemu zaimplementować stosowny interfejs. Warto też wspomnieć, że obsługa błędów i ostrzeżeń jest niezbyt rozwinięta. Oznacza to, że w przypadku błędu czy

też nagięcia pewnych reguł przez użytkownika (np. nieprawidłowa składnia modelu PDDL) system może przestać działać i zwrócić przy tym informację, która wcale nie nakieruje na źródło problemu. Debugowanie aplikacji może w pewnych sytuacjach okazać się więc niezmiernie trudne.

4. Projekt rozwiązania

W tym rozdziale przedstawiony zostanie projekt całego systemu. Omówione będą dotyczącego założenia oraz opis docelowej pracy robota (z naciskiem na wspomaganie ludzi starszych, co jest częścią tematu tej pracy). Opisany również zostanie udział wszelkich narzędzi, które zostały omówione w części teoretycznej pracy (rozdziały 2 i 3).

4.1. Założenia i oczekiwane efekty

4.1.1. Praca robota oparta na scenariuszach

Głównym założeniem systemu, którego opis stanowi niniejsza praca jest działanie w oparciu o przedstawione scenariusze [53]. Zostały one zaczerpnięte z dokumentacji projektu INCARE [53], gdzie znajduje się również ich szczegółowy opis. Na potrzeby niniejszej pracy scenariusze zachowań robota zostaną krótko omówione również poniżej.

— **Hazard Detection (wykrywanie niebezpieczeństw)**

Robot ma wykrywać niebezpieczeństwa i informować o nich użytkownika. Niebezpieczeństwem mogą być przykładowo otwarte drzwi lub okna. Może nim być również pozostawione włączone światło lub działający piekarnik. W celu wykrywania takich sytuacji robot powinien wykorzystywać dostępną na swojej platformie sensorykę oraz (opcjonalnie) urządzenia systemu intelligentnego domu. Oznacza to zatem, że robot musi mieć możliwość komunikacji z takim systemem tak, by mógł on uzyskać wszelkie dostępne informacje o otoczeniu. Warto wyróżnić tu kluczowe zachowania robota związane z tym scenariuszem: nawigacja, zaawansowana obsługa sensorów i przetwarzanie danych z nich pochodzących, interakcja z człowiekiem.

— **Transportation Attendant (wspomaganie transportu)**

Robot ma pomagać człowiekowi w poruszaniu się i przewożeniu niedużych ładunków. Konstrukcja robota (omówiona już w rozdziale 3.2.1) umożliwia spełnianie funkcji podparcia dla osoby poruszającej się. Dodatkowo robot na swoim korpusie może przykładowo przewozić posiłki lub napoje. Osoba starsza, która podczas przemieszczania się zmuszona jest korzystać z kuli ortopedycznych lub balkonika rehabilitacyjnego ma zajętą jedną lub obie ręce. Przygotowany wcześniej posiłek może zatem zostać przwieziony z miejsca na miejsce przez robota, człowiek natomiast przemieści się sam (lub również opierając się o niego). W punkcie docelowym człowiek może zestać posiłek z robota na stół. Kluczowymi zachowaniami robota są zatem nawigacja, podążanie za człowiekiem, interakcja z człowiekiem.

— **Active Human Fall Prevention (aktywne zapobieganie upadków)**

Robot ma wykonywać działania polegające na wykrywaniu obiektów leżących na podłodze, które mogą być przyczyną przyszłego upadku człowieka. Ma on zatem wykonywać przejazdy po zadanym obszarze klasyfikując jednocześnie obiekty na neutralne oraz na obiekty potencjalnie niebezpieczne. Przykładem obiektu neutral-

4. Projekt rozwiązania

nego może być domowe zwierze. Obiektem potencjalnie niebezpiecznym może być pozostawiony na podłodze but lub element garderoby - cokolwiek, o co człowiek może się potknąć lub na czym może się poślizgnąć. Po zakończeniu przejazdu po pewnym obszarze robot powinien zdać użytkownikowi raport o wykrytych przedmiotach i ich położeniu. Gdyby człowiek zaczął zbliżać się do pomieszczenia, którego skan nie zakończył się robot powinien zatrzymać człowieka i poinformować go o postępach wykonywanego procesu i odpowiednio go ostrzec. Kluczowe zachowania zatem to nawigacja, wykrywanie obiektów oraz interakcja z człowiekiem.

— **Fall Verification / Assistance (weryfikacja upadku / asysta)**

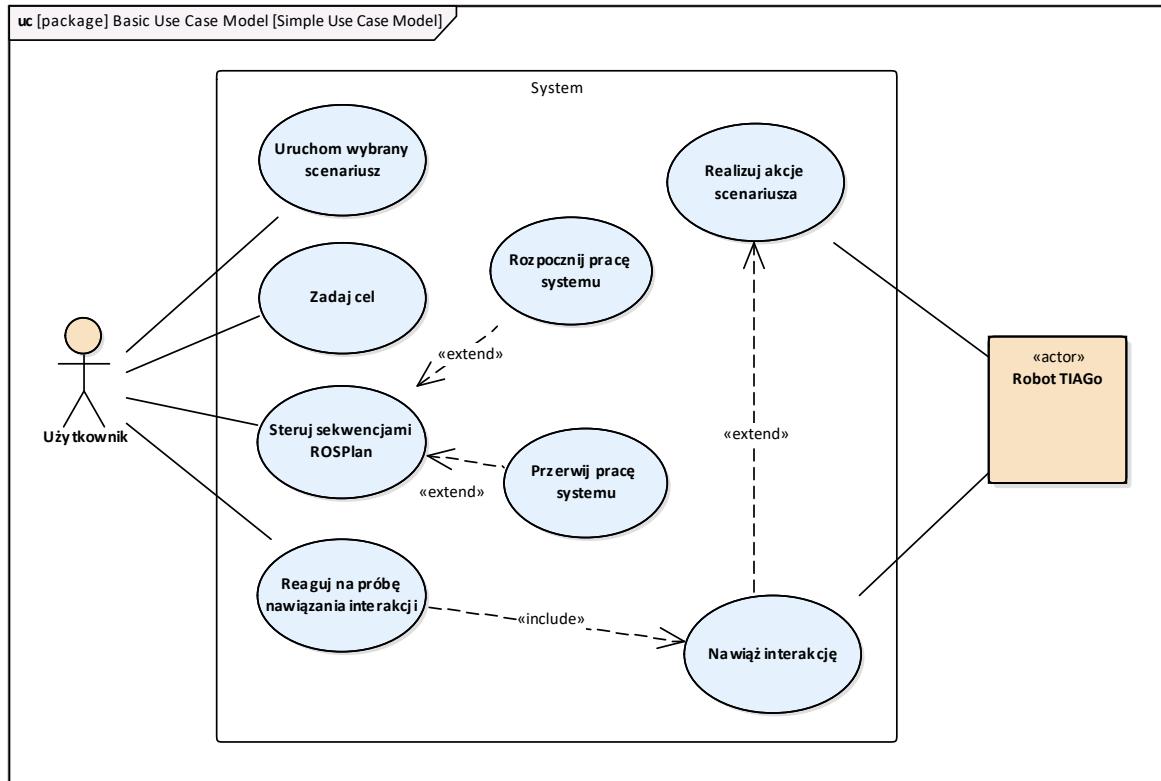
Robot ma wykonywać działania wspomagające człowieka po wykryciu jego upadku. Samo zadanie wykrywania upadków i ich lokalizowania realizowane będzie przez dedykowane urządzenie - robot powinien w pierwszym kroku potwierdzić tę informację przy użyciu dostępnych sensorów. W przypadku potwierdzenia wystąpienia upadku robot ma przystąpić do próby nawiązania interakcji z człowiekiem i utrzymania jej. Zdarza się, że w takich sytuacjach starsza osoba nie jest w stanie się z nikim skontaktować z powodu braku możliwości sięgnięcia po telefon. Robot ma taki kontakt umożliwić. Poza tym może on dostarczyć poszkodowanemu niezbędną wodę i towarzyszyć mu do czasu uzyskania profesjonalnej pomocy. Kluczowymi zachowaniami robota są więc nawigacja, wykrywanie upadku i interakcja z człowiekiem

Powyższe scenariusze stanowić będą podstawę zachowań robota. Praca ta skupiać się będzie jednak tylko na trzech pierwszych, czyli **Hazard Detection, Transportation Attendant i Active Human Fall Prevention**. Scenariusz **Fall Verification / Assistance** po wstępnych pracach nad projektem został od niego odsunięty przede wszystkim z uwagi na ograniczony czas. Jak wspomniano w poprzednim akapicie - scenariusz zakłada obsługę specjalistycznego sprzętu (detektora upadku), lokalizowanie leżącego człowieka i interakcję z nim. Kolejnym powodem tej decyzji był zatem poziom trudności narzucony przez sumę przedstawionych zagadnień.

Dodatkowym ważnym założeniem wynikającym później ze struktury wykorzystywanych narzędzi ROS (co zostanie omówione w rozdziale 4.2) jest charakter ogólnej pracy. Akcje, które wykonuje robot w ramach danego scenariusza mają być dedykowane dla tego i tylko tego scenariusza. Założono zatem, że na jednym robocie działać może jednocześnie tylko jeden scenariusz. Może on zostać uruchomiony przez użytkownika lub pewien system nadzędny - nadal jednak w celu włączenia kolejnego scenariusza, aktualny musi zostać przerwany. Dopuszczone powinno być jednak zachowanie dotychczas zdobytej wiedzy robota (zapisanej w komponencie Knowledge Base szkieletu aplikacyjnego ROSPlan) pomiędzy różnymi scenariuszami.

Warto także wspomnieć, że czynności użytkownika systemu powinny ograniczać się do uruchomienia wybranego scenariusza i ustalenia celu jaki ma zostać zrealizowany przez robota w ramach tego scenariusza (rysunek 24). Człowiek powinien mieć poza

tym możliwość przerwania pracy systemu. Może on także reagować w odpowiedni sposób na próby nawiązania interakcji przez robota.



Rysunek 24. Diagram przewidywanych przypadków użycia projektowanego systemu

4.1.2. Założenia dotyczące wykorzystania omówionych narzędzi

Platforma robota

Przede wszystkim w procesie implementacji omawianego systemu wykorzystana zostanie platforma robota TIAGo wraz z jej symulatorem. Głównym założeniem dotyczącym obu tych elementów jest możliwość odtwarzania warstwy sterującej robotem zarówno rzeczywistym jak i symulowanym w niezmienionej formie. Oczywiście wymaga to z kolei symulatora wiernie odtwarzającego warunki rzeczywiste. Jednakże podczas wstępnych prac z tym oprogramowaniem stwierdzono, że założenie to jest w pełni możliwe do zrealizowania, zatem dobrze jest je określić na początku projektu niniejszej pracy.

Szkielet aplikacyjny ROSPlan

Biorąc pod uwagę to, że oprogramowanie to jest ciągle rozwijane, dopuszczalne są pewne błędy oraz perspektywa wkładania niewielkiej pracy w rozwiązywanie ich. Należy jednak pamiętać, że rozwijanie tego szkieletu nie jest celem omawianej pracy, zatem założono, że wkład w naprawę błędów i rozwijanie tego oprogramowania będzie minimalny. Oznacza to także, że wszelkie komponenty tego systemu powinny być używane w standardowej,

4. Projekt rozwiązania

niezmienionej względem źródła formie. W razie wystąpienia pewnych niedogodności to reszta systemu będzie dopasowana do istniejącego szkieletu ROSPlan. Efektem tego jest również ograniczony wybór odpowiedniego planera PDDL do realizacji wszelkich zadań związanych z tym systemem - szkielet ROSPlan wspiera domyślnie jedynie kilka z nich (tabela 4).

Planowanie symboliczne

Problemy rozwiązywane przez planer w ramach planowania symbolicznego będą trywialne. Ich złożoność w żadnym wypadku nie będzie przypominać problemów przedstawianych na turniejach IPC (rozdział 2.2.2). Nie jest również celem omawianej pracy generowanie planu w jak najkrótszym czasie - szczególnie, że przez trywialność opisanych później modeli PDDL będą mogły one być rozwiązane przez dowolny planer. Założono więc, że wybór samego planera odbędzie się na podstawie jego możliwości interpretacji języka PDDL. Jak wspomniano w rozdziale 2.3 różne planery mają różne możliwości jeśli chodzi o interpretację kolejnych wersji języka PDDL. Zostanie zatem wykorzystany taki planer (nadal mając na uwadze ograniczenia nałożone przez ROSPlan), który będzie mógł generować plan na podstawie zaprojektowanych dziedzin PDDL i wykorzystanych przy tym możliwości składni PDDL.

Paczki SMACH i *actionlib*

Głównym założeniem dotyczącym obu tych paczek jest wykorzystanie ich podstawniowych możliwości związanych z opakowaniem automatów skończonych SMACH w serwer *actionlib* (co zostało szerzej opisane w rozdziale 3.5.2).

4.1.3. Oczekiwania związane z użytkowaniem aplikacji

Zarówno efekty pracy robota jak i systemu oraz jego implementacja powinny spełniać wszelkie założenia. Uruchomienie dowolnego scenariusza ma spowodować wolne od błędów i efektywne działanie robota, które objawiać się będzie realizacją wynikających z planu zadań. System ma być responsywny, efekty jego pracy natomiast powinny być zrozumiałe i przewidywalne. Jest to kluczowe z uwagi na to, że użytkownikami robota i systemu będą głównie osoby starsze - w większości nieabyte z technologią (w szczególności z robotyką).

Jeśli chodzi o zagadnienia czysto techniczne - całość implementowanego systemu powinna być przejrzysta i łatwa do debugowania. Zakłada się, że jego budowa będzie w dużej mierze modularna - pojedyncze komponenty systemu powinny być łatwo wymienialne lub możliwe do rozbudowy o nowe funkcje. Możliwe również powinno być podłączenie zewnętrznych systemów, takich jak system inteligentnego domu lub zewnętrzna baza danych / parametrów. W razie potrzeby lub problemów możliwe powinno być również zdalne podłączenie się administratora systemu w celu wspomagania jego pracy lub pozbycia się nieprzewidzianych błędów.

Przede wszystkim jednak działanie robota ma być **możliwie bezpieczne** dla otoczenia. Wysoka niewskazana jest sytuacja, w której nastąpi negatywne oddziaływanie robota na otoczenie. Jest to kwestia pierwszorzędna z uwagi na docelową grupę użytkowników systemu - ludzi.

4.2. Wykorzystanie planowania symbolicznego i narzędzi ROS

4.2.1. Integracja narzędzi. Szkielet systemu

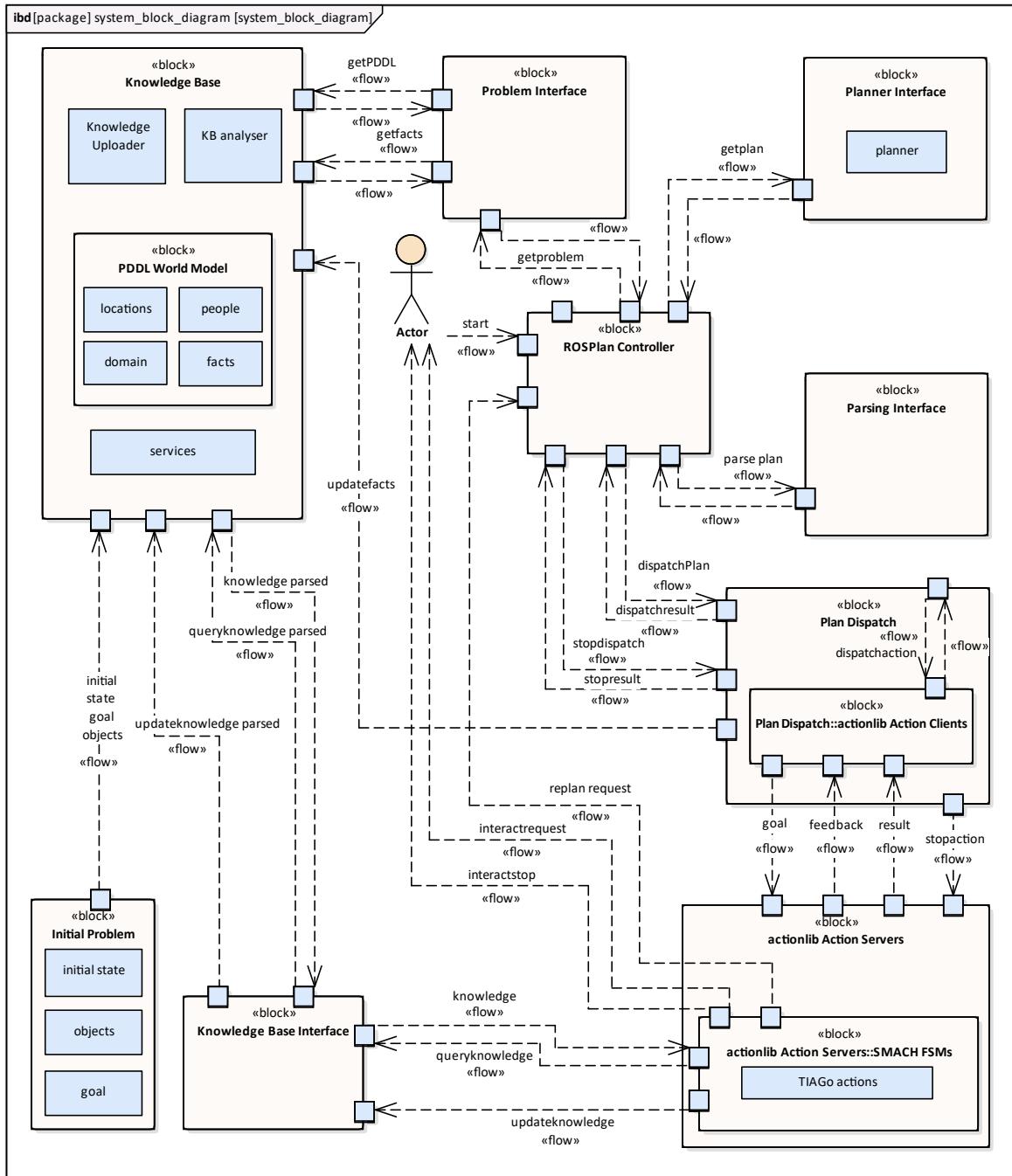
Poniżej zostanie omówiony ogólny projekt całego systemu. Przedstawiony będzie zarówno schemat blokowy zawierający projekt całego systemu (rysunek 25) jak i diagram sekwencji pracy systemu (rysunek 26) dla ogólnego przypadku.

Poniżej przedstawiono wykorzystany na diagramach i w dalszych częściach pracy opis rozkazów i nazw przepływów informacji. Dodatkowy opis nazw znajdzie się w dalszej części rozdziału przy opisie projektów scenariuszy - będą to przepływy informacji związane tylko z nimi, niewykorzystane w ogólnych przypadkach.

- *start* - żądanie uruchomienia systemu i danego scenariusza.
- *getproblem* - żądanie pobrania modelu PDDL danego scenariusza (wlicza się w to dziedzina oraz problem z aktualnym stanem wiedzy).
- *getPDDL* - żądanie o model PDDL.
- *getfacts* - żądanie o aktualny stan wiedzy oraz cele.
- *getplan* - żądanie komponentu *ROSPlan controller* o wygenerowanie planu na podstawie aktualnej wiedzy i celów.
- *parseplan* - żądanie o sparsowanie planu z postaci, którą wygenerował planer do postaci współpracującej z resztą systemu.
- *dispatchplan* - rozkaz uruchomienia realizacji planu.
- *dispatchaction* - rozkaz realizacji konkretnej akcji.
- *goal* - cel akcji *actionlib*.
- *feedback* - wiadomość zwrotna akcji *actionlib*.
- *result* - rezultat akcji *actionlib*.
- *updateknowledge* - aktualizacja stanu wiedzy robota (w komponencie *Knowledge Base*) na podstawie działań akcji.
- *updateknowledge parsed* - akcja powyższa sparsowana do wiadomości formatu *KnowledgeItem*.
- *queryknowledge* - zapytanie o istnienie konkretnego elementu wiedzy robota (w komponencie *Knowledge Base*).
- *queryknowledge parsed* - akcja powyższa sparsowana do wiadomości formatu *KnowledgeItem*.
- *updatefacts* - aktualizacja stanu wiedzy robota (w komponencie *Knowledge Base*) na podstawie zakończenia akcji.

4. Projekt rozwiązania

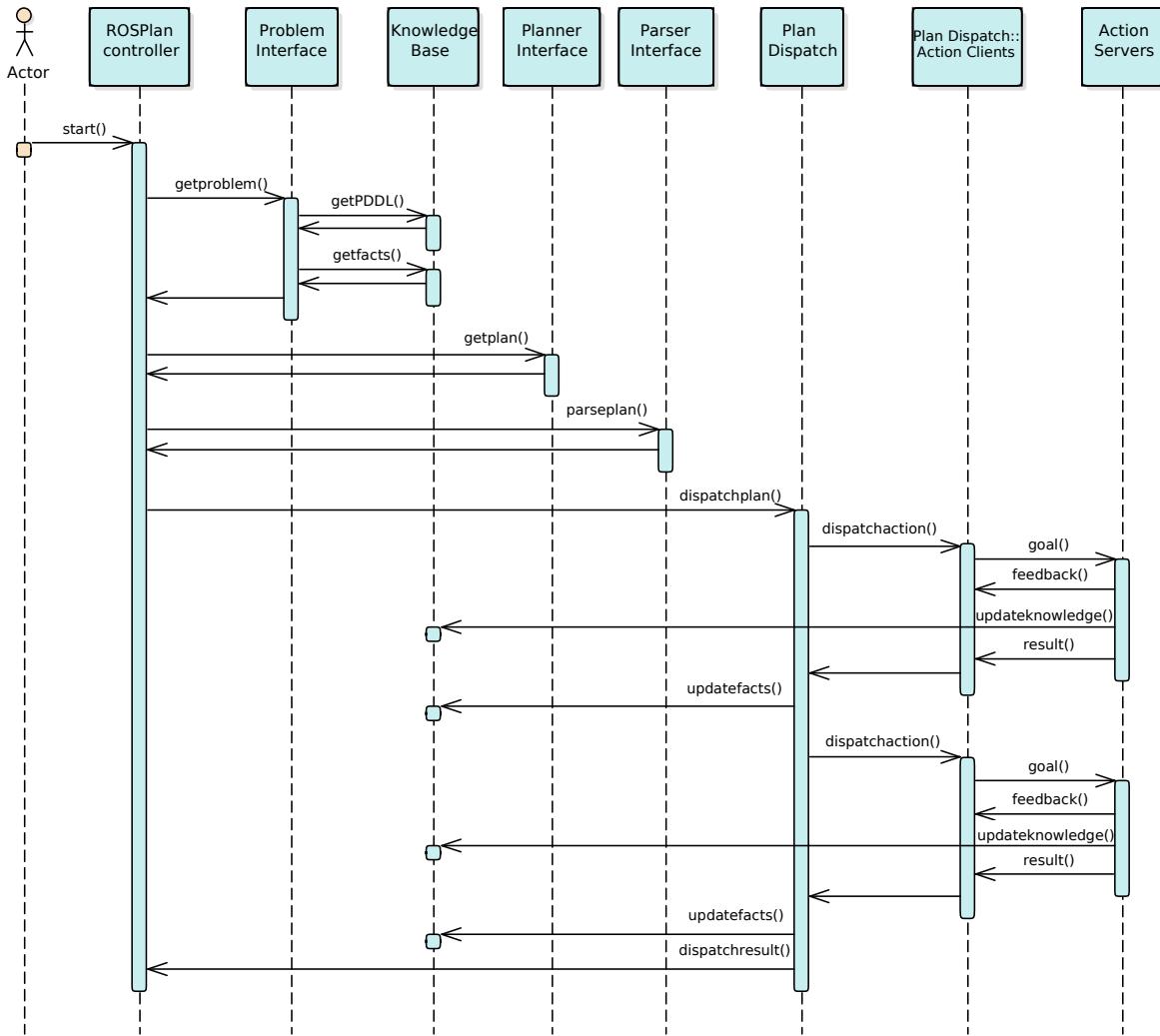
— *dispatchresult* - wynik realizacji całego planu.



Rysunek 25. Schemat blokowy przedstawiający projekt całego systemu wraz z przepływem informacji

4.2.2. Akcje realizowalne przez robota

Przedstawiono poniżej listę akcji wraz krótkim omówieniem każdej z nich, które wykorzystywane będą do realizacji celów PDDL. Oznacza to, że plany generowane przez planer



Rysunek 26. Diagram sekwencji systemu dla generycznego scenariusza

będą składać się właśnie z przedstawionych (odpowiednio uszeregowanych i sparametryzowanych) akcji.

Go

Najbardziej trywialna akcja - przejazd robota z punktu do punktu z określoną docelową orientacją. Przejazd powinien być realizowany wykorzystując zadane parametry, tak by możliwe było wcześniejsze określenie takich wartości jak przyspieszenie czy też prędkość maksymalna.

CheckHazard

Akcja, której celem jest wykrycie niebezpiecznej sytuacji w razie jej wystąpienia lub sytuacji wymagającej uwagi człowieka. Zakłada się, że każdy rodzaj wykrywanego docelowo niebezpieczeństwa wykrywany będzie przy użyciu wydzielonej akcji. Przykładowo jeśli rozpatrywać dwa zjawiska: włączone światło w pokoju oraz pozostawione otwarte drzwi i miałyby być one identyfikowane poprzez działanie tej akcji, to zdefiniowane będą dwie

4. Projekt rozwiązania

oddzielne akcje o przykładowych nazwach: **CheckHazardLight** oraz **CheckHazardDoor**. Zostanie to również przedstawione przy opisie diagramu sekwencji realizacji przykładowego planu scenariusza *Hazard Detection* w paragrafie 4.2.3. Dodatkowo identyfikacja wymagającej uwagi sytuacji powinna odbywać się na dwa sposoby (w zależności od sparametryzowania danego hazardu):

- wykorzystując sensorykę systemu inteligentnego domu
- wykorzystując sensorykę robota

GetLoad

Celem tej akcji jest umieszczenie pewnego (opisanego w parametrach) ładunku na korpusie robota TIAGO. Z uwagi na to, że nie jest on wyposażony w efektory niezbędne do samodzielnego wykonania tej akcji to jej częścią będzie również pewna interakcja z człowiekiem. Przewiduje się, że to człowiek spełni rolę efektora, który umieści żądany ładunek na robota, kiedy ten go o to poprosi.

LeaveLoad

Akcja antagonistyczna do poprzedniej. Jej celem jest pozostawienie ładunku umieszczonego wcześniej na robocie w wybranym miejscu. W tym przypadku do jej realizacji również niezbędna jest obecność człowieka występującego w roli efektora.

GoWithAttendance

Akcja również polegająca przede wszystkim na przejeździe robota z jednego punktu do drugiego z osiągnięciem ustalonej orientacji. Różnica między nią a akcją **Go** polega na tym, że jej efektem będzie (poza przemieszczeniem robota) przewiezienie ładunku na robocie i/lub przemieszczenie człowieka. Zakłada się więc, że robot TIAGO będzie również pełnił rolę urządzenia wspomagającego przemieszczenie się człowieka (rola podpory). Akcja ta, która powinna uwzględniać szereg parametrów takich jak wysokość człowieka czy też odpowiednio niskie przyspieszenia powinna to umożliwiać.

GoScanning

Celem tej akcji jest wykrywanie i identyfikowanie obiektów leżących na podłodze. Aby zrealizować to zadanie robot powinien podczas trwania tej akcji wykonać powolny przejazd od punktu do punktu z opuszczoną głową w celu wykorzystania umieszczonego na niej sensora. Umożliwi to rejestrowanie obrazu oraz jego późniejszą analizę, której wynikiem będzie lista wykrytych obiektów oraz stopień potencjalnego zagrożenia każdego z nich dla człowieka.

HumanInteract

Realizacja tej akcji opierać ma się na interakcji z człowiekiem. Będzie ona wykorzystana

w celu ostrzeżenia go o potencjalnych niebezpieczeństwach lub przekazaniu mu wiedzy robota na temat pewnego otoczenia.

HumanApproachDetection

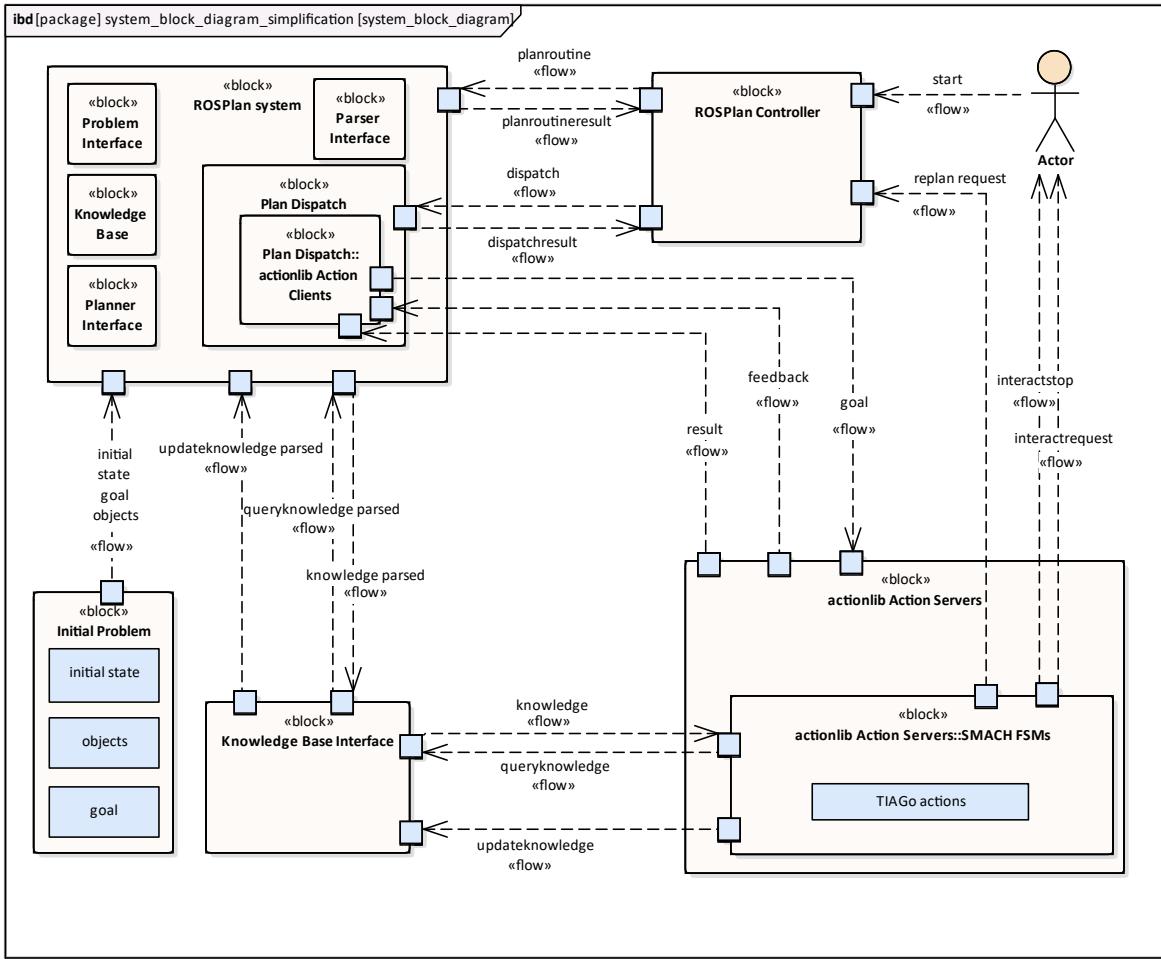
Akcja, która z założenia ma pracować w tle innych akcji (zostanie to przedstawione na projekcie przykładowego diagramu sekwencji scenariusza **Active Human Fall Prevention** (paragraf 4.2.3)). Jej celem jest zwrócenie odpowiedniego sygnału, statusu jeśli wykryte zostanie zjawisko zbliżającego się do robota człowieka. Będzie to niezbędne podczas wykonywania wspomnianego scenariusza (tak jak opisane to w sekcji 4.1.1).

4.2.3. Projekt scenariuszy pracy robota

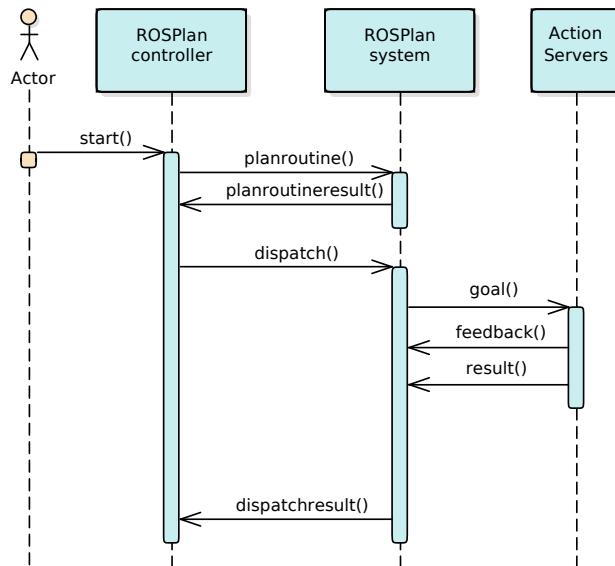
W tym rozdziale zostaną przedstawione na diagramach przykłady przewidywanych sekwencji pracy każdego scenariusza. Wspomniane diagramy przedstawić będą pracę danego scenariusza, którego główną częścią jest realizacja określonego, wygenerowanego przez planer planu. Niemożliwe jest zatem ukazanie takiego diagramu bez podania przykładu właściwego mu planu. Odpowiedni przykład uproszczonego szeregu akcji (który powinien być wygenerowany przez planer) zostanie więc podany przed każdym diagramem sekwencji - stanowić on będzie podstawę zachowań robota w danym przypadku. Każdy z diagramów zostanie omówiony, zwrócona zostanie również uwaga na charakterystyczne elementy pracy systemu przy każdym ze scenariuszy.

Ponadto w celu uproszczenia diagramów sekwencji w tym rozdziale przyjęto przedstawione na rysunku 27 zastąpienie komponentów szkieletu ROSPlan jednym blokiem o nazwie ***ROSPlan system***. W każdym przypadku również zastąpiony fragment diagramu wygląda tak samo - informacje tam zawarte nie są również kluczowe w danym rozdziale. Cały zespół przepływów informacji do komponentów szkieletu ROSPlan (poza komponentem *Plan Dispatch*) oraz informacji zwrotnych do komponentu *ROSPlan Controller* zastąpiony został zespołem generycznych sygnałów ***planroutine()*** oraz ***planroutineresult()***. Przepływ informacji między komponentami *ROSPlan Controller* i *Plan Dispatch* skrócony został do pary sygnałów ***dispatch()*** oraz ***dispatchresult()***. Zabieg taki pozwolił na znaczne uproszczenie diagramów sekwencji przedstawiających przebiegi pracy systemu podczas działania przykładowych scenariuszy. Warto zwrócić w tym momencie uwagę na wprowadzoną różnicę i porównać diagramy sekwencji przedstawione na rysunkach 26 oraz 28. Omówiony zabieg pozwolił nadal zachować kluczowe z punktu widzenia działania samych scenariuszy informacje. Diagramy sekwencji w przypadku każdego scenariusza zawierają będące bloki reprezentujące konkretne akcje realizowalne przez robota. Projekt każdej z nich został szerzej omówiony w rozdziale 4.2.2, gdzie przedstawiono przewidywane zachowania platformy robota w przypadku każdej akcji. Warto również dodać, że projekt każdego scenariusza zaopatrzony został w listę akcji, które robot powinien w ramach niego wykorzystywać.

4. Projekt rozwiązania



Rysunek 27. Przedstawienie komponentów systemu ROSPlan jako jeden blok na potrzeby uproszczenia diagramów sekwencji

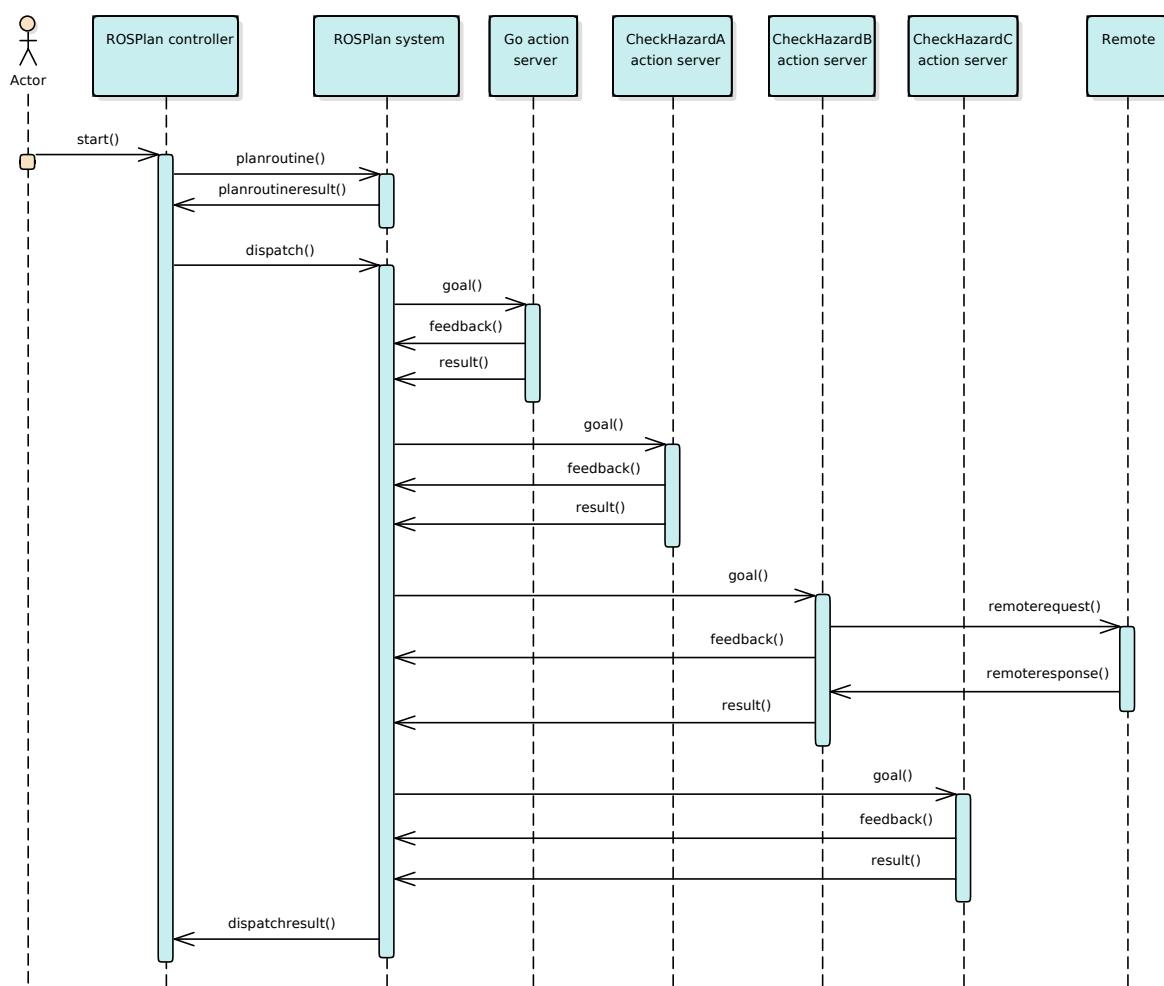


Rysunek 28. Przykładowy diagram sekwencji po zastąpieniu komponentów szkieletu ROSPlan jednym blokiem o nazwie *ROSPlan system*

Przedstawiono także projekt przykładowego planu, który mógłby być wygenerowany przez planer. Ponadto omówione zostaną metody, które pojawią się na diagramach sekwencji każdego scenariusza, a których nie przedstawiono dotychczas.

Hazard Detection

Realizacja tego scenariusza przewiduje przejazd robota po zadanych punktach, w których powinien on dokonać identyfikacji hazardu. Przedstawiony poniżej przykład referencyjny (rysunek 29) jest realizacją planu złożonego z czterech kolejnych akcji (listing 15). Jak widać przykład przewiduje istnienie trzech różnych hazardów (A, B oraz C), których identyfikacja realizowana jest w tym samym miejscu (wynika to z samego planu i diagramu sekwencji 29). Hazard A oraz B istnieją dodatkowo w tym samym miejscu (brak wymaganego przejazdu pomiędzy ich identyfikacją), podczas gdy umiejscowienie hazardu C jest nieznane - jego identyfikacja przebiega poprzez wykorzystanie instalacji inteligentnego budynku (zauważyc to można na diagramie sekwencji 29, gdzie występuje zapytanie do zdalnego systemu).



Rysunek 29. Diagram sekwencji pracy scenariusza Hazard Detection dla przykładowego planu

4. Projekt rozwiązania

Wyróżniające diagram scenariusza metody sekwencji:

- *remoterequest* - zapytanie do systemu zdalnego (w szczególności do systemu inteligentnego budynku) o stan elementów związanych z hazardem.
- *remoteresponse* - odpowiedź systemu zdalnego do komponentu akcji.

Akcje wykorzystane w ramach tego scenariusza:

- **Go**
- **CheckHazardA**
- **CheckHazardB**
- **CheckHazardC**

Listing 15. Projekt przykładowego planu dla scenariusza Hazard Detection

```
0.0: (go [parameters]) [60.0]
60.0: (check_hazard_a [parameters]) [10.0]
70.0: (check_hazard_b [parameters]) [10.0]
80.0: (check_hazard_c [parameters]) [10.0]
```

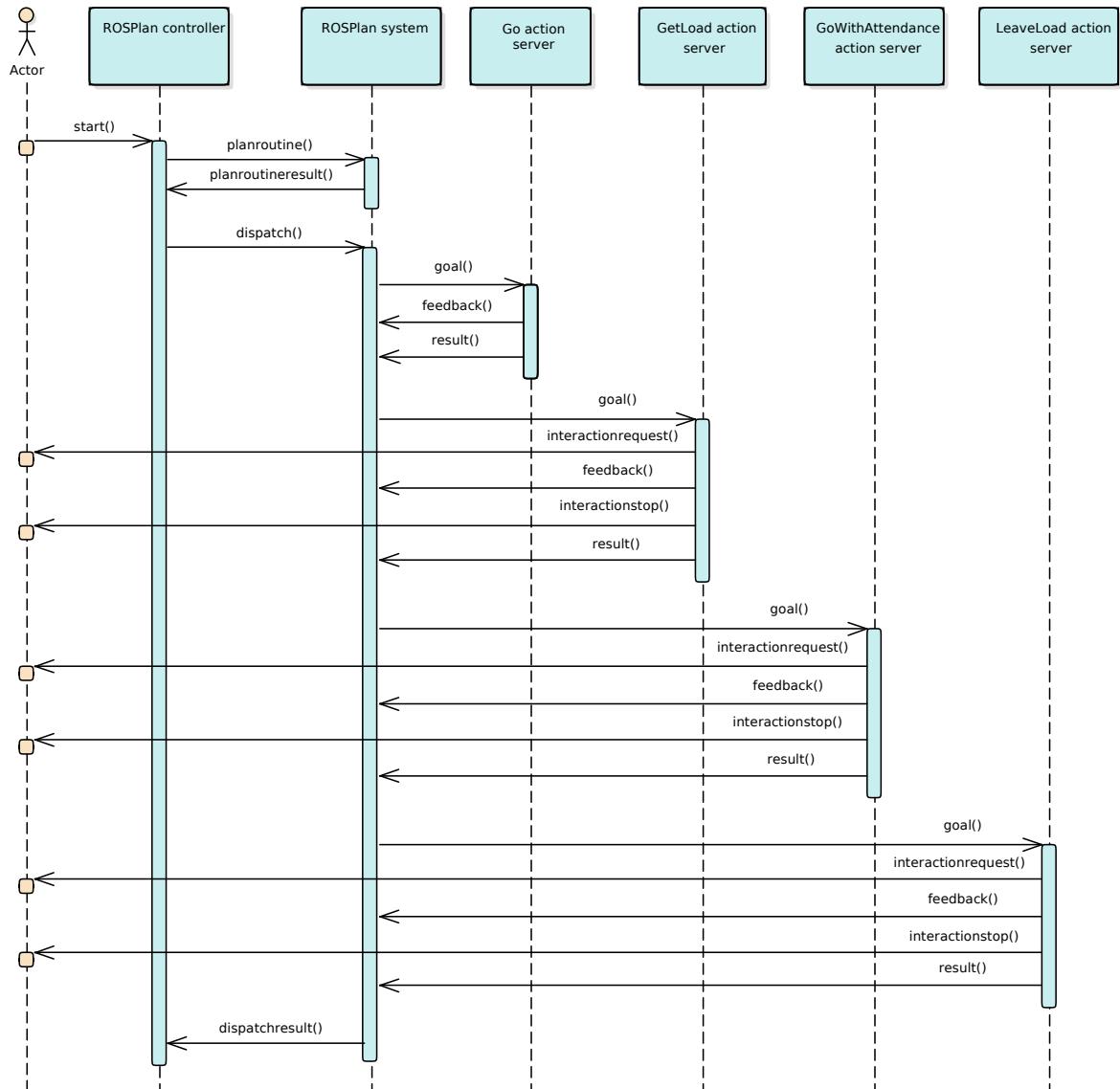
Transportation Attendant

Realizacja tego scenariusza przewiduje przewożenie położonych na robocie towarów z punktu do punktu oraz wspomaganie transportu człowieka. Na poniższym przykładzie referencyjnym (rysunek 30) zauważać można realizację planu złożonego z czterech kolejnych akcji (listing 16). Warto zwrócić uwagę, że robot podczas realizacji tego przykładu powinien wykonać standardowy przejazd, następnie z pomocą człowieka pobrać towar (co wiąże się z interakcją robota i człowieka). Kolejną czynnością jaką powinien wykonać robot jest przejazd z towarem (i prawdopodobnie z opierającym się o robota człowiekiem) zachowując odpowiednią konfigurację. Ostatnią czynnością jest zdjęcie ładunku przez człowieka (akcja **LeaveLoad**). Akcje wykorzystane w ramach tego scenariusza:

- **Go**
- **GetLoad**
- **LeaveLoad**
- **GoWithAttendance**

Listing 16. Projekt przykładowego planu dla scenariusza Transportation Attendant

```
0.0: (go [parameters]) [60.0]
60.0: (get_load [parameters]) [20.0]
80.0: (go_with_attendance [parameters]) [200.0]
280.0: (leave_load [parameters]) [20.0]
```



Rysunek 30. Diagram sekwencji pracy scenariusza Transportation Attendant dla przykładowego planu

Wyróżniające diagram scenariusza metody sekwencji:

- *interactionrequest* - początek nawiązania interakcji robota z człowiekiem (z poziomu komponentu akcji).
- *interactionstop* - zakończenie interakcji robota z człowiekiem.

Active Human Fall Prevention

Realizacja tego scenariusza przewiduje przeprowadzanie przez robota skanowania określonego obszaru w poszukiwaniu pozostawionych na podłodze obiektów i klasyfikacji ich na obiekty bezpieczne i obiekty powodujące potencjalne niebezpieczeństwo. Wspomniane skanowanie powinno trwać dopóki nie zostanie ono przeprowadzone na wszystkich zadanych obszarach lub dopóki nie zostanie wykryty zbliżający się do zadanego

4. Projekt rozwiązania

obszaru człowiek. W takiej sytuacji robot powinien przerwać skanowanie obszaru, wykonać przejazd do aktualnej pozycji tego człowieka, nawiązać z nim interakcję i przekazać kluczowe informacje o procesie skanowania. Powinny być to informacje o wykrytych dotąd obiektach potencjalnie niebezpiecznych oraz o obszarach jeszcze nie przeskanowanych. W celu przekazania czytelnikowi wszelkich informacji związanych z tym scenariuszem przedstawiono na poniższym projekcie przykładu referencyjnego (rysunek 31) przypadek, gdzie podczas pracy robota wykryty został zbliżający się człowiek. Przykład ten podzielić można zatem na dwie części: przed wykryciem zbliżającego się człowieka i po. Pierwsza część opiera się jedynie na przejazdach podczas których robot wykonuje skanowanie obszaru (realizując akcję **GoScanning**) zgodnie z planem przedstawionym na listingu 17. Akcje wykorzystane w ramach tego scenariusza:

- **Go**
- **GoScanning**
- **HumanApproachDetection**
- **HumanInteract**

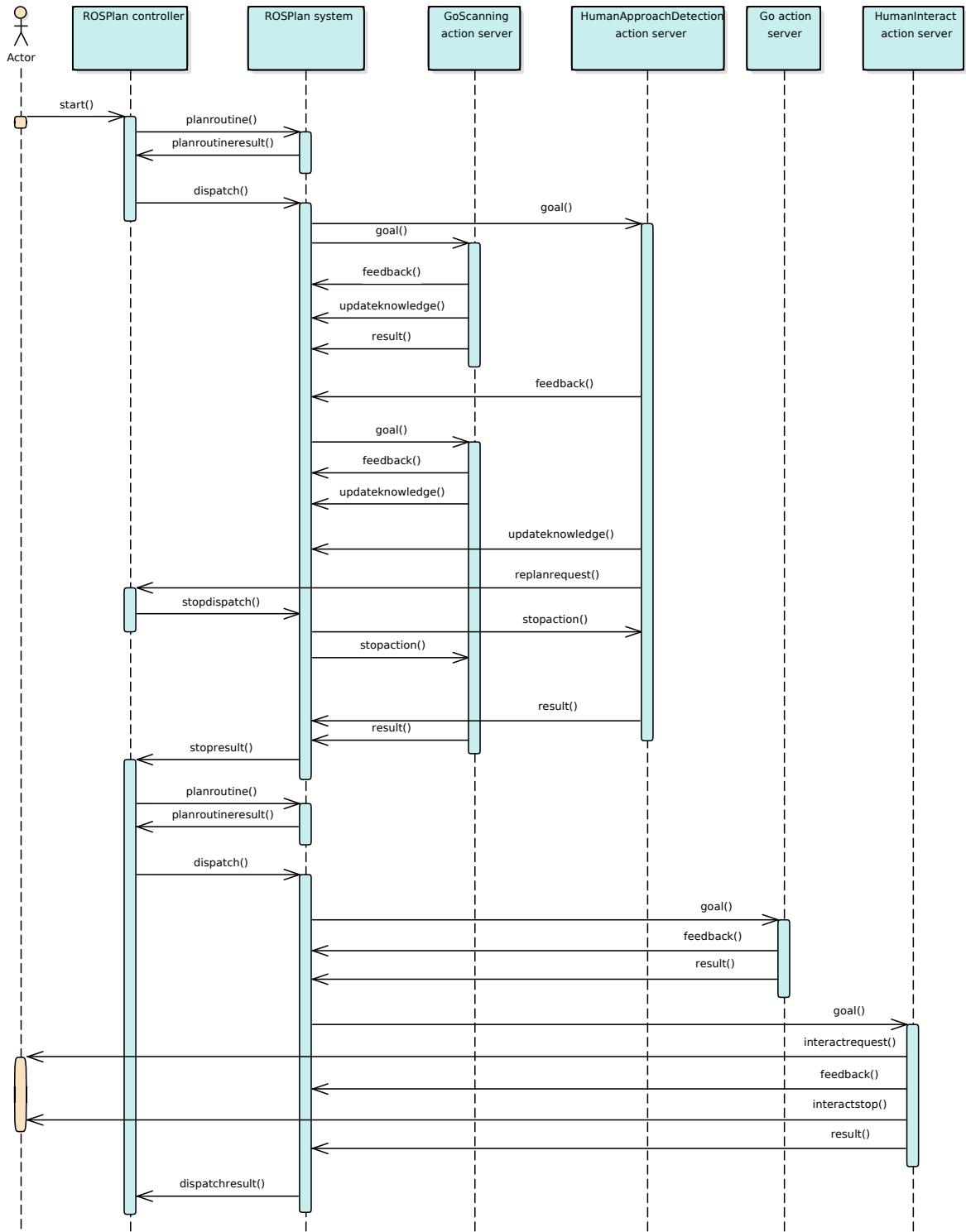
Listing 17. Projekt przykładowego planu dla scenariusza Active Human Fall Prevention

```
0.0 : (human_approach_detection [parameters]) [1000.0]
0.0 : (go_scanning [parameters]) [60.0]
60.0 : (go_scanning [parameters]) [60.0]
120.0 : (go_scanning [parameters]) [60.0]
```

Przy czym podczas wykonywania drugiej akcji **GoScanning** wykryty zostaje zbliżający się do pomieszczenia człowiek (rezultat działania w tle akcji **HumanApproachDetection**). Efektem tego powinno być przerwanie realizacji obecnego planu oraz wygenerowanie i realizacja planu kolejnego z uwzględnieniem nowych faktów dostępnych w bazie danych komponentu *Knowledge Base* (jednym z nich jest przede wszystkim informacja o zbliżającym się człowieku). Nowy wygenerowany plan niech ma postać przedstawioną na listingu 18. Dzięki temu rozpoczyna się realizacja drugiej części przykładu referencyjnego złożonego z dwóch akcji. Jak widać jej celem powinien być przejazd do pozycji człowieka i nawiązanie z nim interakcji.

Listing 18. Projekt przykładowego planu dla scenariusza Active Human Fall Prevention wygenerowanego po wykryciu zbliżającego się człowieka

```
0.0 : (go [parameters]) [60.0]
60.0 : (human_interact [parameters]) [60.0]
```



Rysunek 31. Diagram sekwencji pracy scenariusza Active Human Fall Prevention dla przykładowego planu

- *replanrequest* - żądanie zatrzymania działania systemu i przygotowania nowego planu na podstawie aktualnego stanu wiedzy.
- *stopdispatch* - żądanie zatrzymania realizacji aktualnego planu.
- *stopaction* - żądanie przerwania wykonywania akcji.

4. Projekt rozwiązania

- *stopresult* - wynik realizacji żądania **stopdispatch**.
- *interactrequest* - początek nawiązania interakcji robota z człowiekiem (z poziomu komponentu akcji).
- *interactstop* - zakończenie interakcji robota z człowiekiem.

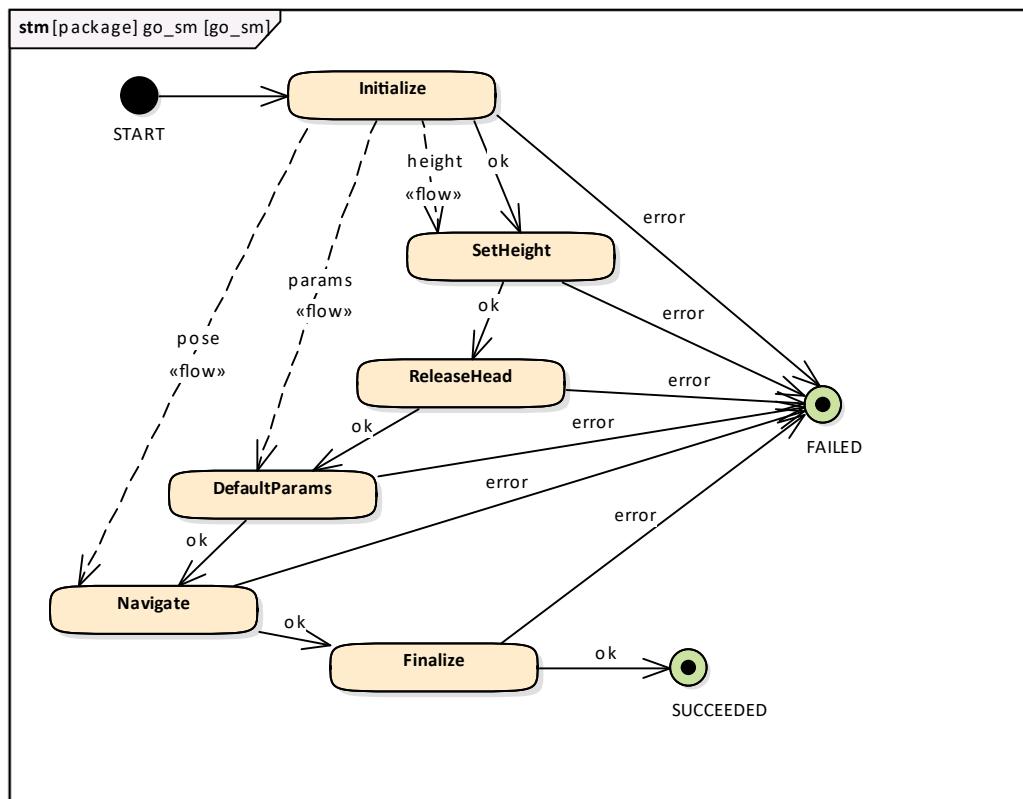
4.2.4. Fizyczna realizacja zaplanowanych akcji - projekt automatów skończonych

Każda z przedstawionych w poprzednim rozdziale akcji będzie realizowana przez robota przy pomocy deterministycznych automatów skończonych. W tej części pracy przedstawiony zostanie projekt tych automatów wraz z krótkim omówieniem ich stanów.

Warto jednak najpierw omówić dwa stany szczególne, które obecne będą w każdym automacie akcji, a z poziomu których nie będą wykonywane jakiekolwiek działania robota:

- *Initialize()* - stan początkowy. Stanowić on będzie element, do którego najpierw trafią informacje niezbędne do wykonania działań przez robota z poziomu kolejnych stanów automatu. Informacje te trafią do tego stanu poprzez zawartość wiadomości *żądania* (rozdział 3.4) wysłanej przez klienta odpowiedniej akcji. Dodatkowo stan ten odpowiedzialny będzie za podział wspomnianych informacji (parametrów) i dystrybucję ich do odpowiednich stanów automatu, gdzie zostaną wykorzystane przez poszczególne procesy i działania robota. Na przedstawionych dalej projektach automatów przekazywanie parametrów przedstawiono jako przepływ informacji «flow».
- *Finalize()* - stan końcowy. Odpowiedzialny będzie za sformułowanie rezultatu wykonania odpowiedniej akcji. Z tego miejsca odesłana zostanie do klienta akcji *actionlib* wiadomość *wynik* zawierająca ten rezultat co będzie również sygnałem zakończenia działania akcji *actionlib*.

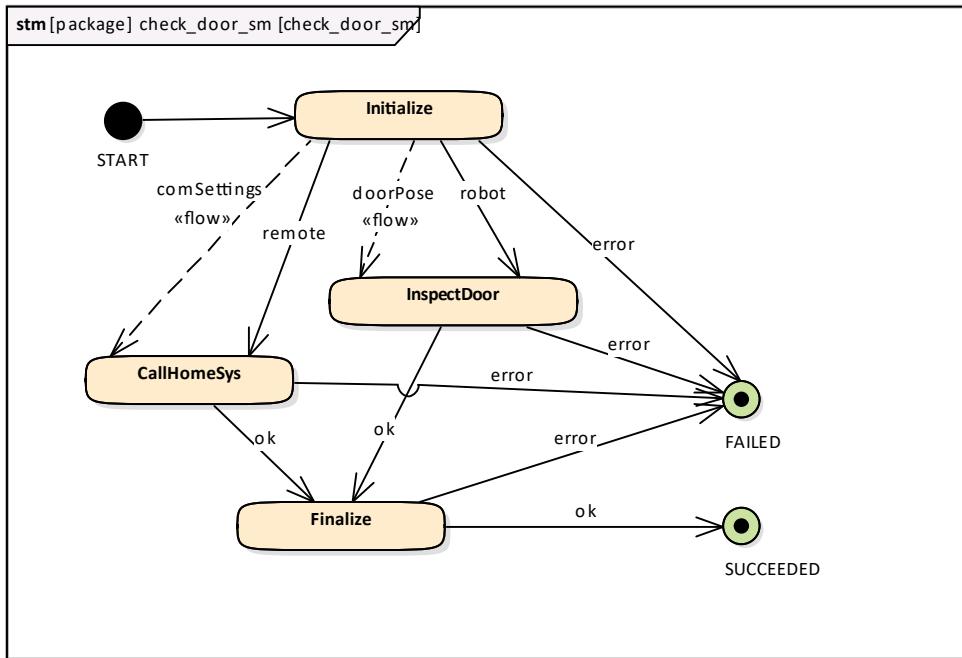
Wszystkie stany projektowanych automatów akcji **Go** (rysunek 32), **GoWithAttention** (rysunek 37), **GetLoad** (rysunek 38), **LeaveLoad** (rysunek 39), **GoScanning** (rysunek 40), **HumanApproachDetection** (rysunek 34) i **HumanInteract** (rysunek 41) mogą zakończyć się słowem *ok* lub *error*. Oznacza to odpowiednio, że stan zakończył swoją pracę poprawnie (zadanie, za którego realizację był odpowiedzialny zostało wykonane) lub wystąpił pewien błąd, który uniemożliwi wykonanie całej akcji. Jeśli stan zakończy swoją pracę słowem *ok* to dalsze działanie automatu powinno potoczyć się zgodnie z projektem (pracę powinien rozpoczęć odpowiedni, kolejny stan). Zakończenie pracy jakiegokolwiek stanu słowem *error* spowodować powinno natychmiastowe zakończenie pracy automatu. Serwer *actionlib*, który będzie ten automat opakowywał powinien w tej sytuacji zwrócić wiadomość *wynik* zawierającą odpowiedni status błędu. Informacja o niepowodzeniu wykonania takiej akcji zostanie więc przekazana dalej do szkieletu ROSPlan. Wyjątkami są stany automatów akcji **CheckDoor** (rysunek 33), **CheckLight** (rysunek 35) oraz **CheckWindow** (rysunek 36) gdzie wprowadzono dodatkowe słowa, którymi mogą się one zakończyć.



Rysunek 32. Projekt automatu skończonego akcji Go

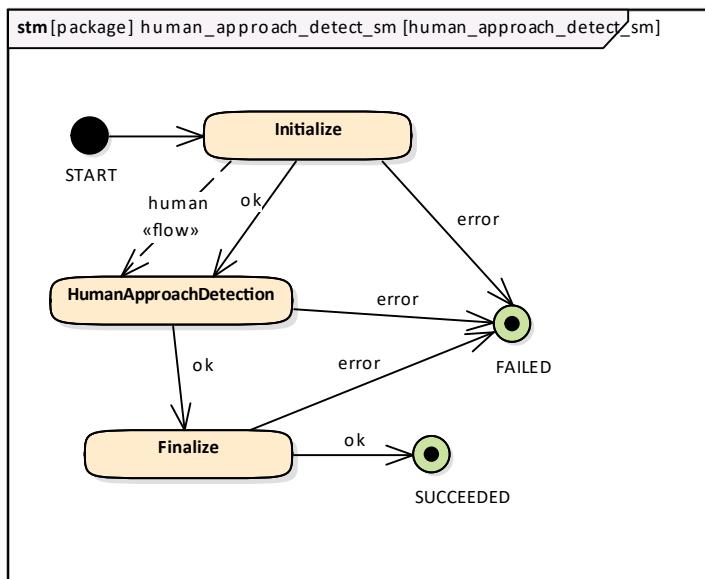
- *SetHeight(height)* - ustawienie wysokości korpusu robota TIAGo (funkcja zmiennej *height*).
- *ReleaseHead()* - uwolnienie przegubów głowy robota - dzięki temu kontrolę nad nimi przejąć może system robota.
- *DefaultParams(params)* - ustawienie domyślnych wartości parametrów nawigacji platformy jezdnej.
- *Navigate(pose)* - wykorzystanie algorytmów nawigacji TIAGo w celu dojechania do punktu opisanego parametrem *pose*

4. Projekt rozwiązania



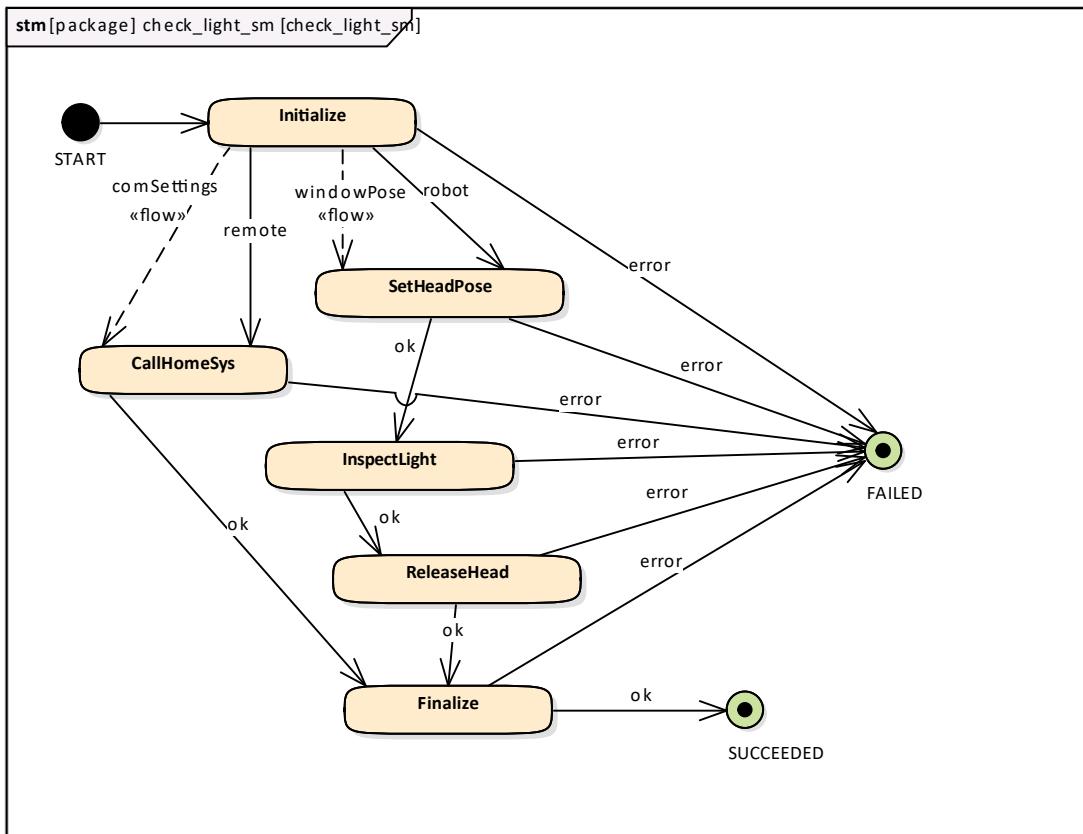
Rysunek 33. Projekt automatu skońzonego akcji CheckDoor

- *InspectDoor(doorPose)* - inspekcja drzwi zlokalizowanych w `doorPose`. Określone ma zostać to czy są otwarte czy zamknięte.
- *CallHomeSys(comSettings)* - wysłanie zapytania do systemu zdalnego (przykładowo do systemu inteligentnego budynku) w celu otrzymania informacji o hazardzie. `comSettings` jest zbiorem parametrów opisujących sposób komunikacji



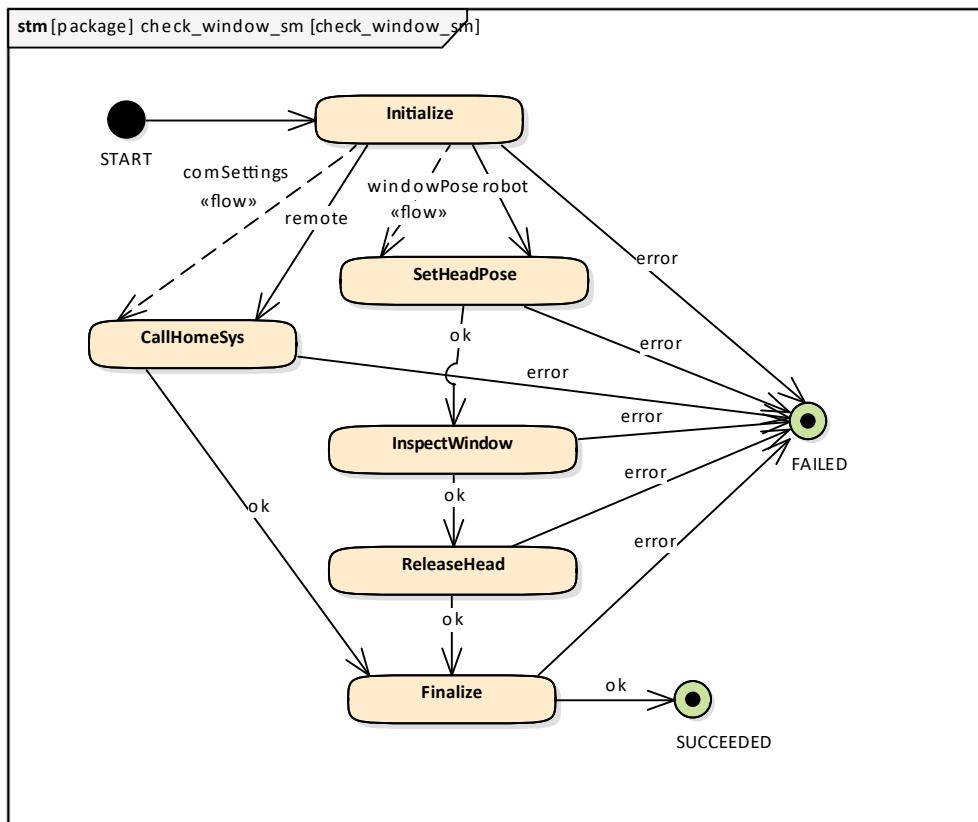
Rysunek 34. Projekt automatu skońzonego akcji HumanApproachDetection

- *HumanApproachDetection(human)* - detekcja zbliżającego się człowieka opisanego parametrem `human`. Stan ten powinien zakończyć swoją pracę odpowiednim statusem w przypadku detekcji omówionego przypadku



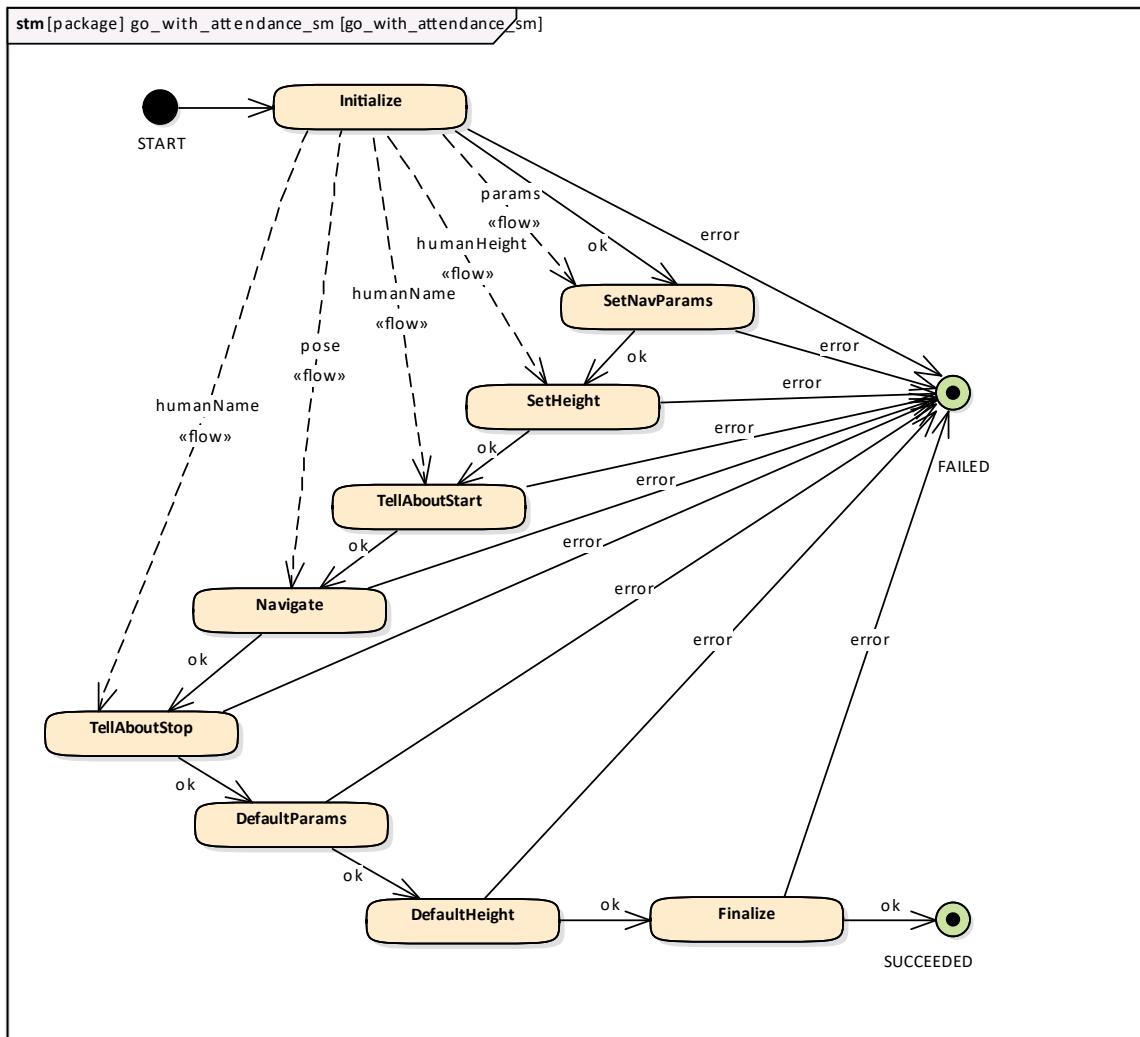
Rysunek 35. Projekt automatu skończonego akcji CheckLight

- `SetHeadPose(lightPose)` - ustawienie przegubów głowy robota tak, by sensor na niej umieszczony skierowany był na wykrywane źródło światła `lightPose`.
- `CallHomeSys(comSettings)` - wysłanie zapytania do systemu zdalnego (przykładowo do systemu inteligentnego budynku) w celu otrzymania informacji o hazardzie. `comSettings` jest zbiorem parametrów opisujących sposób komunikacji.
- `InspectLight()` - inspekcja stanu źródła światła.
- `ReleaseHead()` - uwolnienie przegubów głowy robota



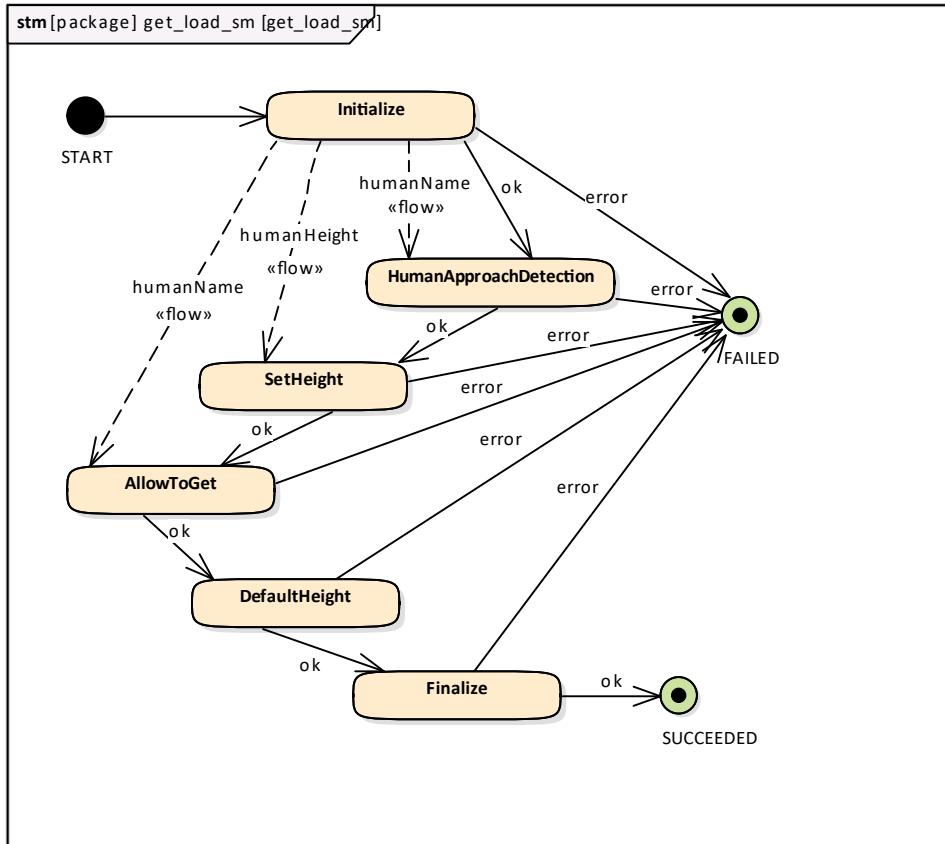
Rysunek 36. Projekt automatu skońzonego akcji CheckWindow

- *SetHeadPose(windowPose)* - ustawienie przegubów głowy robota tak, by sensor na niej umieszczony skierowany był na badane okno w położeniu opisanym parametrem *windowPose*.
- *CallHomeSys(comSettings)* - wysłanie zapytania do systemu zdalnego (przykładowo do systemu inteligentnego budynku) w celu otrzymania informacji o hazardzie. *comSettings* jest zbiorzem parametrów opisujących sposób komunikacji.
- *InspectWindow()* - inspekcja okna, proces ten powinien określić czy okno jest otwarte czy zamknięte.
- *ReleaseHead()* - uwolnienie przegubów głowy robota



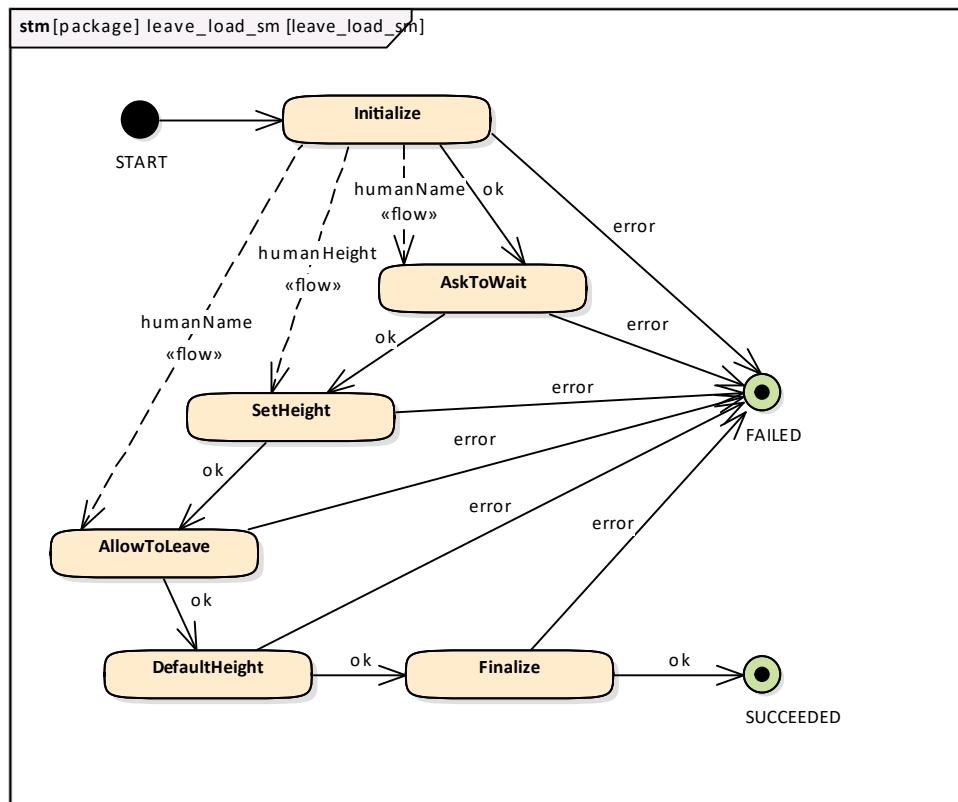
Rysunek 37. Projekt automatu skończonego akcji GoWithAttendance

- *SetNavParams(params)* - ustawienie parametrów nawigacji opisanych parametrem *params*.
- *SetHeight(height)* - ustawienie wysokości korpusu robota TIAGo (funkcja zmiennej *height*).
- *TellAboutStart(humanName)* - przekazanie człowiekowi o imieniu opisanym parametrem *humanName* informacji o rozpoczęciu poruszania się.
- *Navigate(pose)* - wykorzystanie algorytmów nawigacji TIAGo w celu dojechania do punktu opisanego parametrem *pose*.
- *TellAboutStop(humanName)* - przekazanie człowiekowi o imieniu opisanym parametrem *humanName* informacji o zakończeniu poruszania się.
- *DefaultParams()* - przywrócenie domyślnych wartości parametrów nawigacji platformy jezdnej.
- *DefaultHeight()* - przywrócenie domyślnej wysokości korpusu robota TIAGo



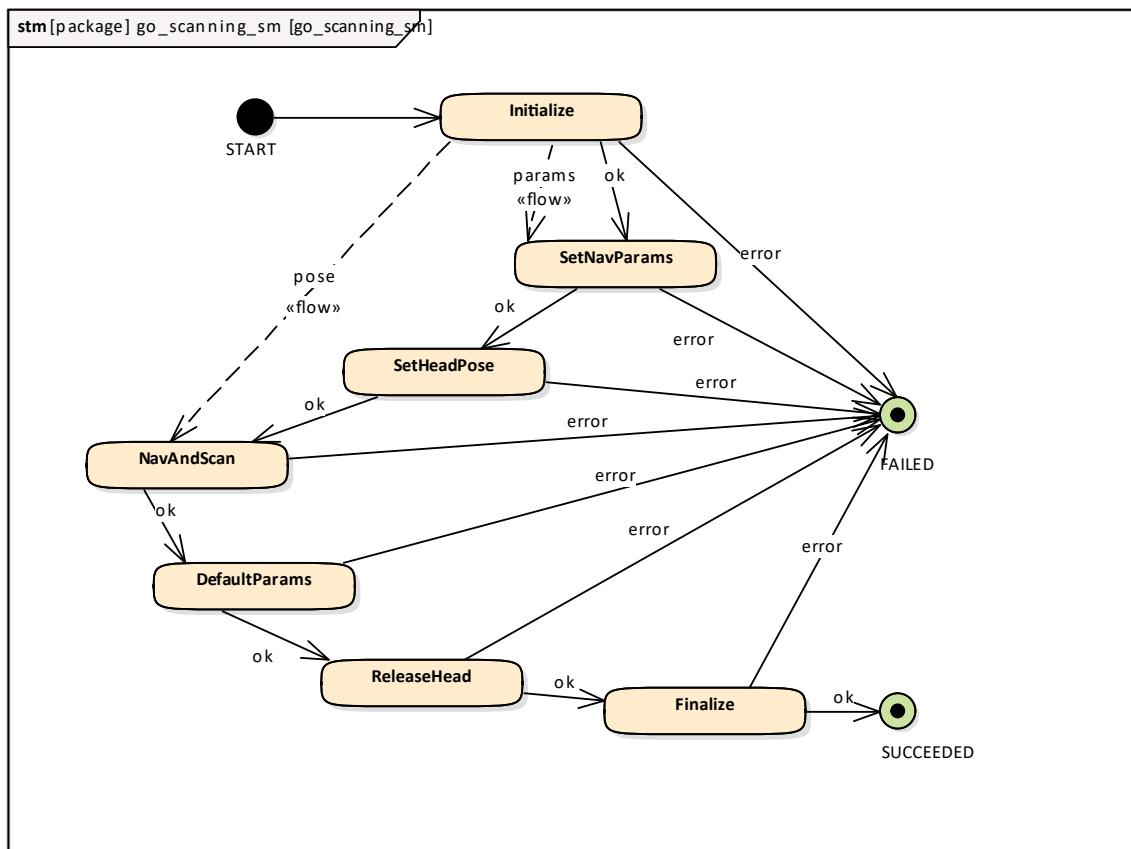
Rysunek 38. Projekt automatu skońzonego akcji GetLoad

- `AskToWait(humanName)` - poproszenie człowieka o imieniu opisanyem parametrem *humanName* o to, by ten zaczekał.
- `SetHeight(humanHeight)` - ustawienie wysokości korpusu robota TIAGO (funkcja zmiennej *height*).
- `AllowToGet(humanName)` - pozwolenie człowiekowi o imieniu opisanyem parametrem *humanName* na umieszczenie ładunku na korpusie.
- `DefaultHeight()` - przywrócenie domyślnej wysokości korpusu robota TIAGO



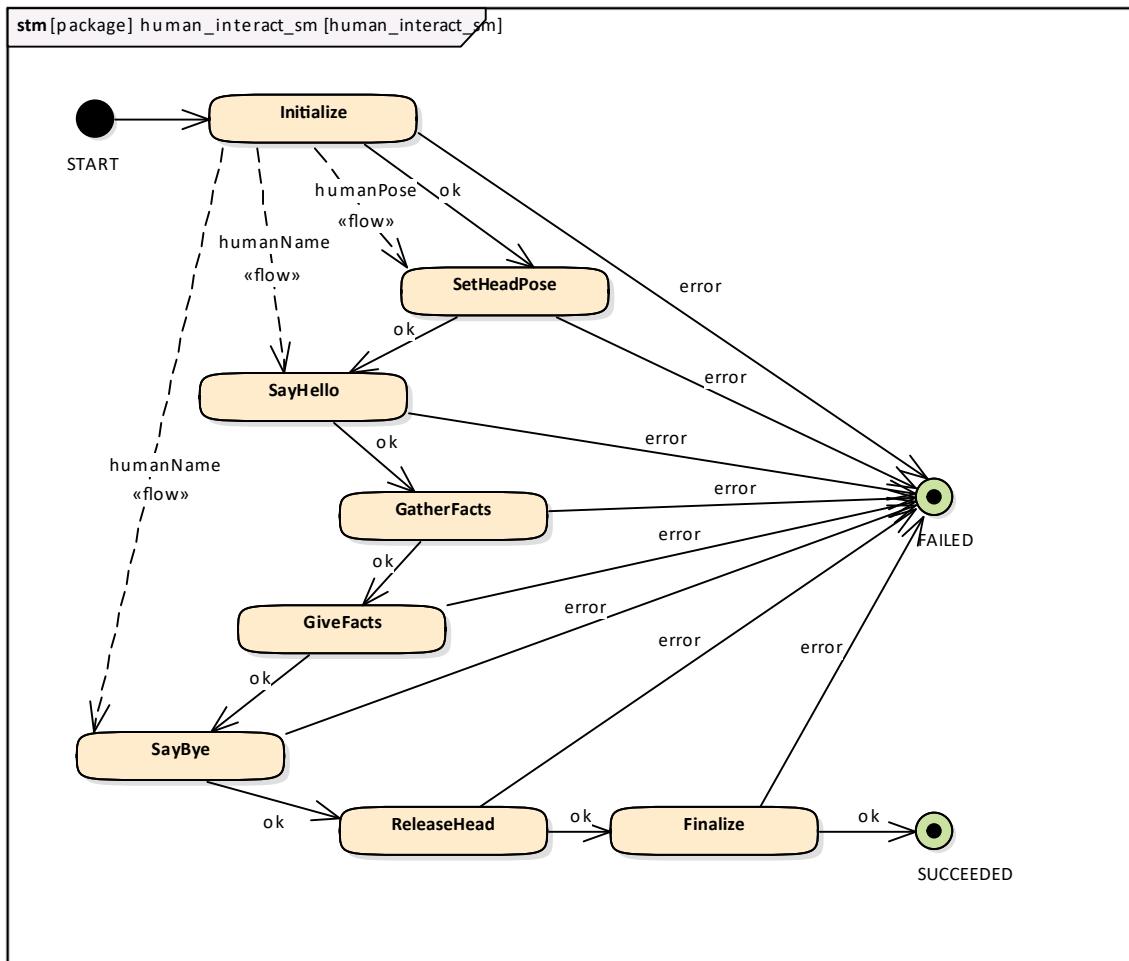
Rysunek 39. Projekt automatu skończonego akcji LeaveLoad

- *AskToWait(humanName)* - poproszenie człowieka o imieniu opisanyem parametrem *humanName* o to, by ten zaczekał.
- *SetHeight(height)* - ustawienie wysokości korpusu robota TIAGo (funkcja zmiennej *height*).
- *AllowToLeave(humanName)* - pozwolenie człowiekowi o imieniu opisanyem parametrem *humanName* na zdjecie ładunku z korpusu robota.
- *DefaultHeight()* - przywrócenie domyślnej wysokości korpusu robota TIAGo



Rysunek 40. Projekt automatu skońzonego akcji GoScanning

- `SetNavParams(params)` - ustawienie parametrów nawigacji opisanych parametrem `params`.
- `SetHeadPose()` - ustawienie przegubów głowy robota tak, by sensor na niej umieszczony skierowany był na obszar przed robotem w celu identyfikacji leżących na podłodze obiektów.
- `NavAndScan(pose)` - nawigacja robota z punktu do punktu z jednoczesną analizą obrazu przy użyciu sensora na głowie TIAGo.
- `DefaultParams()` - przywrócenie domyślnych wartości parametrów nawigacji platformy jezdnej.
- `ReleaseHead()` - uwolnienie przegubów głowy robota



Rysunek 41. Projekt automatu skończonego akcji HumanInteract

- *SetHeadPose(humanPose)* - ustawienie przegubów głowy robota tak, by sensor na niej umieszczony skierowany był na człowieka na pozycji opisanej parametrem *humanPose*.
- *SayHello(humanName)* - werbalne powitanie człowieka o imieniu zwartym w parametrze *humanName*.
- *GatherFacts()* - wysłanie zapytania do komponentu *Knowledge Base* w celu uzyskania wiedzy o zidentyfikowanych obiektach podczas dotychczasowej pracy robota.
- *GiveFacts(facts)* - przekazanie uzyskanych wcześniej faktów człowiekowi, z którym nawiązana została interakcja.
- *SayBye(humanName)* - werbalne pożegnanie człowieka o imieniu zwartym w parametrze *humanName*.
- *ReleaseHead()* - uwolnienie przegubów głowy robota

4.3. Użycie dostępnej platformy i otoczenia

4.3.1. Robot TIAGo z zainstalowanym oprogramowaniem

Realizacja projektowanego systemu odbędzie się z wykorzystaniem robota mobilnego TIAGo (rozdział 3). Wykorzystane zostaną jego możliwości sprzętowe takie jak wysoka mobilność, regulacja wysokości górnej jego części (korpusu) oraz możliwość poruszania głową z zainstalowanym w niej sensorem głębi i kamerą. Wewnątrz robota obecne są również głośniki co w połączeniu z jego pozostałymi cechami pozwoli na nawiązywanie interakcji z człowiekiem. Jego platforma i ruchoma głowa wykorzystywane będą w procedurach związanych z analizą otoczenia takich jak wykrywanie hazardów lub identyfikacja obiektów. Płaska powierzchnia korpusu robota TIAGo w połączeniu z możliwością jego zmiany wysokości pozwoli na sprawne wspomaganie człowieka w przemieszczaniu się po domu oraz w przewożeniu ładunków takich jak jedzenie czy picie.

4.3.2. Symulator

Symulator jest (jak już wspomniano) elementem niezbędnym podczas rozwijania systemu robotycznego. W omawianej pracy spełni on dwie role - pierwszą z nich jest możliwość implementacji fragmentów systemu przed pojawiением się robota w laboratorium lub podczas pracy w innym miejscu. Pozwala to na poświęcenie większej ilości czasu nad projektem ponieważ prace można rozpoczęć odpowiednio wcześniej. Drugą rolą symulatora jest możliwość uprzedniego sprawdzenia efektów działania nowych funkcji bez wprowadzania potencjalnego niebezpieczeństwa dla robota lub jego otoczenia. Oczywiście jest, że robot może w pewnych warunkach i podczas uruchamiania zawierającego błędy oprogramowania zachować się nieprzewidywalnie.

4.3.3. Środowisko pracy robota

Robot w celu realizacji niniejszej pracy powinien pracować w przygotowanym wcześniej środowisku. Do dyspozycji powinien mieć odpowiednio dużą przestrzeń wolną od wysokich progów czy podjazdów, po której zdolny będzie nawigować. Przestrzeń ta (z uwagi na temat pracy) powinna jak najbardziej przypominać otoczenie, w którym może żyć osoba starsza. Mogłoby to być zwyczajne mieszkanie lub odpowiedni ośrodek. W otoczeniu robota powinny znaleźć się elementy, na których mogą być testowane jego funkcje, czyli źródła pewnych hazardów czy obiekty będące źródłem potencjalnego niebezpieczeństwa dla obecnego tam człowieka.

5. Implementacja systemu

5.1. Konfiguracja symulatora i środowiska pracy robota

Symulator

Pierwszą czynnością wykonaną w celu realizacji omawianej pracy było przygotowanie symulatora robota TIAGo. Podczas pierwszych tygodni pracy robot nie był obecny w laboratorium, była to zatem czynność niezbędna do wykonania, aby możliwe było rozpoczęcie prac. Jak już wcześniej wspomniano (rozdział 3.2.3) symulator jest zestawem narzędzi rozwijanym przez producenta robota i jest dostępny do pobrania za darmo ze strony z oprogramowaniem robota [43]. Warto jednak zwrócić uwagę, że w implementacji niniejszej pracy jednym z kluczowych czynników jest środowisko pracy robota. Wykorzystując zatem oprogramowanie *Gazebo* oraz *Blender* (rozdział 3.1) zamodelowano pomieszczenie laboratorium, w którym docelowo miały przebiegać testy robota TIAGo. Polegało to na wykorzystaniu programu *Blender* do zamodelowania pojedynczych obiektów obecnych w laboratorium, takich jak stoły, krzesła, szafy, komputery itd. Każdy taki model został wyeksportowany do pliku z rozszerzeniem *.dae (Digital Asset Exchange). Umożliwiło to później importowanie tych plików do programu *Gazebo*, w którym rozmieszczone poszczególne modele w przestrzeni na wzór rzeczywistego pomieszczenia.

Należy również dodać, że każdy taki plik z modelem pojedynczego obiektu (*.dae) wzbogacony został o dwa dodatkowe pliki, których opis przedstawiono niżej, a które są niezbędne, by model taki mógł być wykorzystany w *Gazebo*.

- Plik z rozszerzeniem *.sdf - opis właściwości fizycznych modelu.
- Plik z rozszerzeniem *.config - opis zawierający metadane opisowe (nazwa modelu, wersja, autor itd.)

Sam model *.dae jest jedynie graficzną reprezentacją obiektu. Aby mógł on być wykorzystany jako element wirtualnego świata w symulatorze musi on zostać wzbogacony o opis powierzchni kolizyjnych oraz właściwości fizyczne (przykładowo tensor bezwładności oraz współczynniki tarcia powierzchni obiektu). Założono, że jakakolwiek kolizja robota z elementem otoczenia jest niedopuszczalna. Nieistotne jest zatem definiowanie (dość skomplikowanych) tensorów bezwładności modelowanych obiektów. Opisano je zatem jako obiekty statyczne, co w rozumieniu środowiska *Gazebo* znaczy, że obiekty te są nieprzesuwalne. Znacznie uprościło to prace nad modelowaniem środowiska. Dzięki temu również sama symulacja przebiega o wiele sprawniej z uwagi na mniejsze zużycie mocy obliczeniowej komputera. Jest to czynnik, na który trzeba zwrócić uwagę, ponieważ symulacja pracy robota w tym środowisku jest sama w sobie wymagająca obliczeniowo, wszelkie uproszczenia są zatem pożądane.

5. Implementacja systemu

Pewnego uproszczenia dokonano również przy opisie powierzchni kolizyjnych. Są to elementy, które są „widzialne” przez sensorykę robota w symulatorze. Możliwe jest wykorzystanie w tym celu powierzchni pochodzących z samych modeli wykonanych w programie *Blender*. Są to jednak powierzchnie nieregularne, zbudowane z siatek. Postanowiono, że powierzchnie kolizyjne zostaną opisane za pomocą trywialnych brył (prostopadłościów), co również powinno w znacznym stopniu wpływać na prędkość pracy symulatora. Na listingu 19 przedstawiono przykład opisu powierzchni kolizyjnej jednej z nóg stołu. Jak widać opis taki jest dość przejrzysty, wymagane jest tylko podanie wymiarów bryły i jej położenia w przestrzeni. W podobny sposób zamodelowano powierzchnie pozostałych elementów wchodzących w skład modelu pomieszczenia (rysunek 42).

Listing 19. Opis powierzchni kolizyjnej jednej z nóg biurka w postaci prostopadłościanu

```
<collision name="left_back_leg">
  <pose>-0.3283 0.6266 0.3300 0 0 0</pose>
  <geometry>
    <box>
      <size>0.0434 0.0467 0.6600</size>
    </box>
  </geometry>
</collision>
```



(a) Zrzut ekranu z programu Gazebo



(b) Zdjęcie przedstawiające otoczenie robota

Rysunek 42. Model pomieszczenia, w którym testowany będzie system oraz jego rzeczywisty odpowiednik

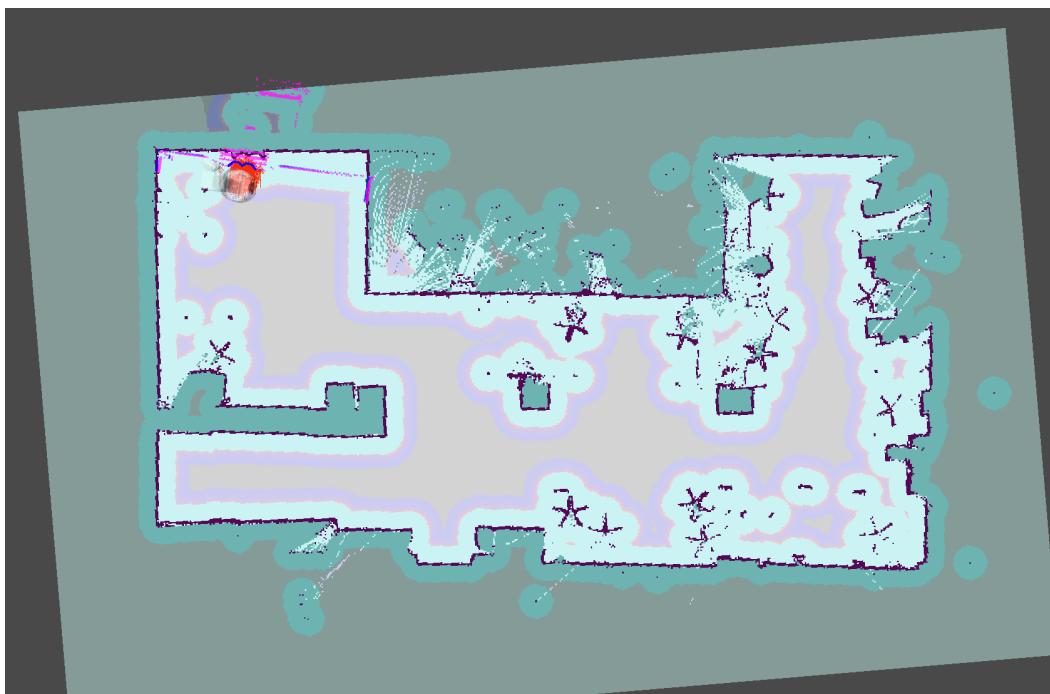
Przygotowanie środowiska pracy robota

Środowisko pracy robota, czyli pomieszczenie laboratoryjne, w którym testowany będzie implementowany system pozostawiono bez zmian. Uznamo, że spełnia ono wszelkie założenia dotyczące projektu, znajdują się tu stoły, biurka i krzesła, które spotkać można także w domach i mieszkaniach starszych osób - skutecznie utrudniają również nawigację robota. W pomieszczeniu laboratoryjnym obecna jest również wąska, dobrze wyposażona kuchnia oddzielona od reszty przestrzeni listwami podłogowymi, co także utrudni poruszanie się platformy jezdnej. Obecne są okna oraz drzwi wejściowe, co pozwoli na

potencjalne badanie ich stanu za pomocą widzenia maszynowego. Możliwe jest również zainstalowanie pewnych czujników, które w przyszłości mogłyby służyć jako część zdalnego systemu inteligentnego budynku. Istniejące zatem otoczenie robota nie zostało w żaden sposób zmodyfikowane.

Konfiguracja robota

Robot TIAGo po wstępnych badaniach i rozpoznaniu jego systemu przez użytkowników (w tym autora pracy) okazał się poprawnie działać bez potrzeby żadnej dodatkowej, nieplanowanej konfiguracji. Należało jedynie przeprowadzić krótką konfigurację komputerów, które miałyby się z nim w ogólności komunikować. Polegała ona na podłączeniu się do sieci bezprzewodowej generowanej przez komputer pokładowy robota. Aby możliwe było uruchamianie programów systemu ROS (w tym elementów implementowanego systemu) z poziomu komputera użytkownika należało dodatkowo ustawić adres IP mastera ROS. Polega to na przypisaniu adresu IP komputera pokładowego robota do zmiennej środowiskowej *ROS_MASTER_URI* na komputerze użytkownika. Aby jednak możliwe było rozpoczęcie pracy z wykorzystaniem nawigacji robota należało przeprowadzić jednorazowe budowanie mapy pomieszczenia (rysunek 43). Wykorzystano w tym celu gotowe oprogramowanie robota TIAGo. Od użytkowników procedura ta wymagała jedynie wykonania kilku przejazdów robotem po zakamarkach całego pomieszczenia.



Rysunek 43. Mapa zbudowana przez robota na podstawie rzeczywistego pomieszczenia laboratoryjnego

5.2. Implementacja komponentów systemu

5.2.1. Instalacja i konfiguracja wymaganych paczek ROS

Czynności wstępne

Rozwijanie systemu opisywanego w tej pracy wymagało przede wszystkim zainstalowania całego oprogramowania robota TIAGo wraz z symulatorem na komputerze użytkownika. Instrukcja opisująca potrzebne czynności znajduje się na stronie internetowej simulatora TIAGo [54]. Instalacja opiera się na pobraniu wielu paczek z różnych repozytoriów i ich późniejszej komplikacji. Na stronie jednak znaleźć można gotowy skrypt, który wykonuje potrzebne kroki automatycznie. Sam proces nie sprawił dzięki temu żadnych kłopotów.

Instalacja paczki *SMACH*

Paczka ta nie była domyślnie zainstalowana w systemie ROS. Z uwagi na to, że jest ona oficjalnie dostępna w repozytoriach tego systemu [46] wystarczyło ją doinstalować w sposób zgodny z jego standardami.

Instalacja szkieletu aplikacyjnego *ROSPlan*

Paczki wchodzące w skład szkieletu aplikacyjnego ROSPlan nie są oficjalnie dostępne w repozytoriach systemu ROS. Jedynym sposobem na wykorzystanie ich w projekcie jest pobranie ich plików źródłowych z udostępnionego przez autorów repozytorium [55] oraz skompilowanie ich. Strona z odpowiednim repozytorium zawiera również listę zależności, które muszą być uprzednio zainstalowane. Omawiane paczki zostały więc dodane do przestrzeni roboczej implementowanego systemu, dzięki czemu są razem z nim komplikowane. Okazało się to pomocne w sytuacjach, w których wprowadzane były drobne zmiany lub poprawki do elementów szkieletu ROSPlan.

Instalacja planera TFD

Ostatnią podstawową operacją jest instalacja planera TFD (Temporal Fast Downward, został on pokrótkę omówiony w paragrafie 2.3). Tak jak wcześniej wspomniano jest to planer domyślnie wspierany przez szkielet aplikacyjny ROSPlan (tabela 4). Oznacza to, że w systemie zaimplementowany jest już program interfejsujący się z nim. Jedyną wymaganą czynnością potrzebną do rozpoczęcia pracy z planerem jest jego pobranie z odpowiedniej strony internetowej [36] (jest to wolne oprogramowanie udostępnione na licencji GNU) oraz komplikacja. Najnowsza udostępniona wersja nosi oznaczenie 0.4, pobrana paczka jest więc oznaczona nazwą **tfd-src-0.4**. Paczkę należy rozpakować do odpowiedniego katalogu znajdującego się w paczce *rosplan_planning_system* szkieletu ROSPlan, a dokładnie do:

rosplan/rosplan_planning_system/common/bin/[tfd-src-0.4]

Wypakowane źródła należy dodatkowo skompilować przechodząc do wskazanego wyżej katalogu i wywołując polecenie:

./build

Planer TFD jest budowany domyślnie na systemach 32-bitowych, aby zatem został zbudowany poprawnie na systemie 64-bitowym należy doinstalować przedstawione niżej zależności:

libc6-i386, g++-multilib

Poza tym wymagany jest również zainstalowany Python w wersji 2.5 lub 2.6 [36].

5.2.2. Implementacja warstwy sterującej robotem

Każdy scenariusz pracy robota zaimplementowany został w postaci oddzielnej paczki (rozdział 3.3) systemu ROS. Dodatkowo utworzono dwie dodatkowe paczki związane ze wspomaganiem pracy systemu oraz dwie paczki zawierające definicje wszelkich wiadomości, serwisów czy akcji. Krótki opis wszystkich utworzonych paczek przedstawiono poniżej:

- ***rosplan_tiago_hazard_detection*** - implementacja scenariusza *Hazard Detection*.
- ***rosplan_tiago_transportation_attendant*** - implementacja scenariusza *Transportation Attendant*.
- ***rosplan_tiago_active_human_fall_prevention*** - implementacja scenariusza *Active Human Fall Prevention*.
- ***rosplan_tiago_params*** - zawiera programy zarządzające i udostępniające parametry używane przez pozostałe komponenty systemu (przykładowo zapisane dokładne współrzędne punktów powiązane z ich nazwami lub podstawowe dane użytkowników).
- ***rosplan_tiago_common*** - biblioteki i skrypty wykorzystywane w pozostałych częściach systemu. Między innymi implementacja komponentu *ROSPlan Controller* oraz skrypty ułatwiające sterowanie robotem TIAGO.
- ***rosplan_tiago_core_msgs*** - definicje wiadomości, serwisów, akcji wykorzystywanych w paczkach *rosplan_tiago_params* i *rosplan_tiago_common*.
- ***rosplan_tiago_scenarios_msgs*** - definicje wiadomości, serwisów, akcji wykorzystywanych w paczkach zawierających implementację scenariuszy.

Warto omówić szerzej zawartość paczek zawierających implementację poszczególnych scenariuszy. Przykład struktury plików takiej paczki pokazano na rysunku 44. Poszczególne katalogi zawierają:

- ***/include*** oraz ***/src***
implementację części komponentu *Plan Dispatch* wraz z klientami akcji *actionlib* poszczególnych akcji PDDL.
- ***/nodes***
implementację serwerów akcji *actionlib*, a co za tym idzie każdy plik znajdujący się tam zawiera implementację automatu skońzonego *SMACH* opakowanego we wspomniany serwer.

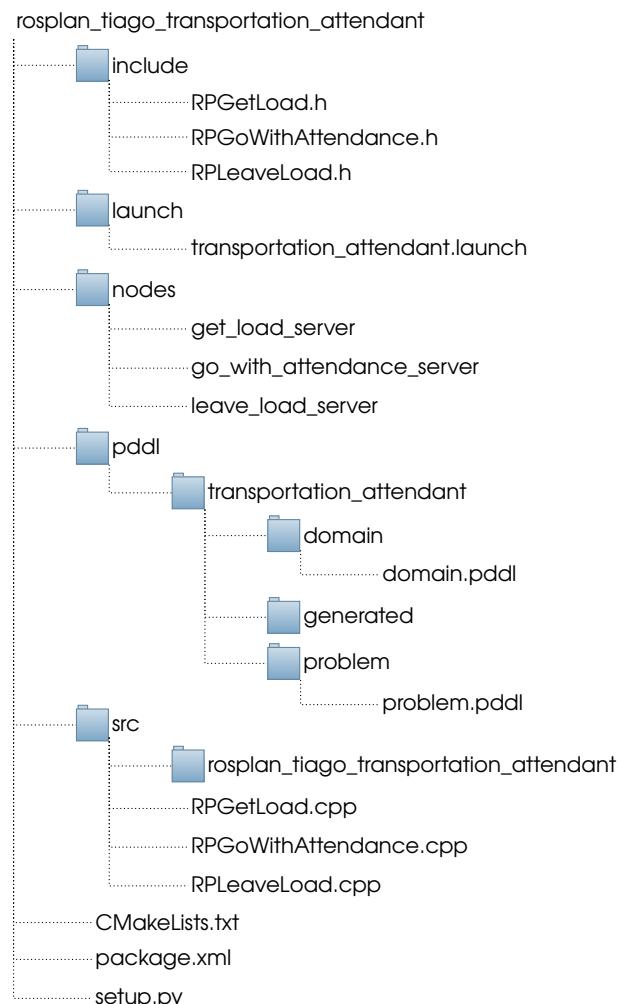
5. Implementacja systemu

— */pddl/[nazwa_scenariusza]*

model PDDL scenariusza podzielony na dziedzinę i problem. Obecny jest tam również katalog, do którego trafia opis problemu generowany przez komponent *Knowledge Base* na podstawie stanu aktualnej wiedzy.

— */launch*

plik *.launch uruchamiający cały system - zarówno komponenty szkieletu ROSPlan jak i komponenty implementowane.



Rysunek 44. Struktura plików paczki `rosplan_tiago_transportation_attendant`

5.2.3. Uruchamianie systemu

Włączenie konkretnego scenariusza

Rozpoczęcie pracy z systemem i robotem TIA Go należy rozpocząć od włączenia wybranego scenariusza. Należy to zrobić wykorzystując odpowiedni plik `*.launch`. Przykładowo włączenie scenariusza *Transportation Attendant* polega na uruchomieniu poniższej komendy.

`roslaunch rosplan_tiago_transportation_attendant transportation_attendant`

Każdy taki plik `*.launch` zawarty w paczce z implementacją scenariusza uruchamia wszystkie potrzebne komponenty szkieletu ROSPlan oraz związane ze scenariuszem akcje (automaty *SMACH* z aplikacjami klienckimi). Jest to zatem podstawowy i jedyny sposób na łatwe przygotowanie systemu do pracy.

Sekwencja startu pracy robota

Po przedstawionym wyżej włączeniu systemu dalsze sterowanie nim odbywa się głównie przez sterowanie komponentami szkieletu ROSPlan. Należy bowiem przekazać do nich rozkazy związane z generacją problemu PDDL, uruchomieniem planera, sparsowaniem planu do formatu wiadomości systemu ROS oraz rozpoczęciem jego wykonywania. Podobnie jak w pozostałych przypadkach związanych z komunikacją z komponentami ROSPlan tu również odbywa się to za pomocą serwisów. Polega to dokładnie na wywołaniu serwisów przedstawionych niżej w kolejności odpowiadającej opisany wcześniej zadaniom i zapewniającej prawidłowe rozpoczęcie pracy.

- `/rosplan_problem_interface/problem_generation_server`
- `/rosplan_planner_interface/planning_server`
- `/rosplan_parsing_interface/parse_plan`
- `/rosplan_plan_dispatcher/dispatch_plan`

Aby umożliwić łatwy start aplikacji wykorzystany będzie skrypt zapisany jako `/rosplan_tiago_common/scripts/rosplan_run.bash`

którego zadaniem jest wywołanie po kolejnych przedstawionych powyżej serwisów. Jego zawartość przedstawiono na listingu 20.

Listing 20. Zawartość skryptu `rosplan_run.bash`

```
#!/usr/bin/env bash
echo "Generating_a_Problem"
rosservice call /rosplan_problem_interface/problem_generation_server

echo "Planning"
rosservice call /rosplan_planner_interface/planning_server

echo "Executing_the_Plan"
rosservice call /rosplan_parsing_interface/parse_plan
rosservice call /rosplan_plan_dispatcher/dispatch_plan
```

5.3. Opis w języku PDDL

5.3.1. Omówienie dziedzin PDDL związanych z przygotowanymi scenariuszami

Każdy scenariusz pracy robota został przedstawiony w języku PDDL za pomocą odzielnej dziedziny. Z uwagi na dużą objętość plików je zawierających odpowiednie listingi

5. Implementacja systemu

zostały umieszczone w załączniku omawianej pracy. Są to zatem trzy dziedziny zawierające opisy akcji (omówionych wcześniej w rozdziale 4.2.2 i nazywanych akcjami realizowalnymi przez robota), funkcji i typów, które mogą być wykorzystane w ramach danego scenariusza. Cały ten opis zgodny jest z wprowadzeniem teoretycznym przedstawionym w rozdziale 2. Warto jednak wspomnieć, że akcje definiowane wykorzystując dyrektywę (*:durative-action*) (w przeciwieństwie do przedstawianej na przykładach dyrektywy (*:action*)). Jest to rozszerzenie wprowadzone do języka PDDL w wersji 2.1 (rozdział 2.2.2), które umożliwia definiowanie akcji o niezerowym czasie trwania. Dzięki temu możliwa będzie taka manipulacja definicji akcji, która pozwoli na jednocześnie wykonywanie przykładowo dwóch akcji już na etapie planowania - możliwa będzie więc między innymi realizacja scenariusza *Active Human Fall Prevention*, gdzie z założenia wykonywana jest w tle jedna z akcji. Wykorzystanie akcji o niezerowym czasie trwania wymusza także zdefiniowanie dyrektywy (*:duration*), która opisuje czas trwania akcji. Jest to także domyślna wartość, która brana jest pod uwagę podczas optymalizacji planu przez planer - jeśli nie zostanie wskazane inaczej plan optymalizowany jest pod względem jego całkowitego czasu trwania.

5.3.2. Przykładowe problemy PDDL

Poniżej przedstawione zostaną także przykładowe problemy PDDL każdego scenariusza. Sekcja zawierająca cel (*:goal*) problemu jest opisywana dla każdego wykorzystania systemu indywidualnie, w zależności od tego co robot ma w istocie wykonać. W poniższym zestawieniu sekcja ta pozostanie zatem pusta, a uwaga zwrócona zostanie na charakterystyczne dla każdego scenariusza elementy warte wyjaśnienia. Przykłady realizacji konkretnych celów przedstawione zostaną natomiast w rozdziale 6.

Scenariusz Hazard Detection

Listing 21. Skrócony problem modelu scenariusza *Hazard Detection*

```
(define (problem task)
  (:domain hazarddetection)
  (:objects
    door_kitchen door_room lamp_room dishwasher_kitchen - hazard
    rico - robot
    initial - robot-location
    door_kitchen_location door_room_location - hazard-location
    lamp_room_location - hazard-location
    dishwasher_kitchen_location wp0 wp1 wp2 wp3 - hazard-location
    lidar rgbd ultrasonic - robot-sensor
    door_sensor window_sensor - home-system-sensor
  ))
```

```

(:init
  (at rico initial)
  (not_checking)

  (linked lamp_room wp1)
  (linked door_room wp2)
  (linked door_kitchen wp3)
  (linked dishwasher_kitchen wp0)

  (sensor_type door_kitchen lidar)
  (sensor_type door_room lidar)
  (sensor_type lamp_room rgbd)
  (sensor_type dishwasher_kitchen rgbd)

; point which sensors are on robot's platform (1 or 0)
  (= (is_robot_sensor lidar) 1)
  (= (is_robot_sensor rgbd) 1)
  (= (is_robot_sensor ultrasonic) 1)
  (= (is_robot_sensor door_sensor) 0)
  (= (is_robot_sensor window_sensor) 0)

)

(:goal (
  ...
))

```

W opisie tego problemu warto zwrócić uwagę na przypisanie szeregu wartości do funkcji (*is_robot_sensor*). Funkcja ta opisuje przynależność zdefiniowanego wcześniej czujnika do robota lub do instalacji inteligentnego budynku. Jak wiadomo scenariusz uwzględnia detekcję hazardów przy użyciu sensoryki obu rodzajów, a odpowiednie jej opisanie przy użyciu tej funkcji pozwala na uwzględnienie tego podziału już na etapie planowania. Jest to kluczowe biorąc pod uwagę fakt, że czas potrzebny na wykonanie akcji detekcji hazardu przy użyciu sensora zdalnego jest znacznie krótszy niż przy wykorzystaniu czujnika obecnego na robocie. Szczególnym przypadkiem może być detekcja potencjalnie niebezpiecznej sytuacji w pewnej odległości od robota - przy użyciu instalacji budynku ogranicza się to jedynie do komunikacji z systemem zdalnym. Wykorzystanie sensora robota wymusza w pierwszej kolejności przejście do odpowiedniego miejsca. Dodatkowo dzięki odpowiedniemu wykorzystaniu informacji o rodzaju czujnika w dyrektywie

5. Implementacja systemu

(*:duration*) akcji, czas przewidziany przez planer na wykonanie tej akcji będzie znacznie krótszy. Dziedzinę tego scenariusza przedstawiono na listingu 34.

Scenariusz Transportation Attendant

Listing 22. Skrócony problem modelu scenariusza *Transportation Attendant*

```
(define (problem task)
  (:domain transportationattendant)
  (:objects
    rico - robot
    luke john - human
    glass sandwich - item
    initial - robot-location
    luke-location john-location wp0 wp1 - human-location
    glass-location sandwich-location spaghetti-location - item-location
  )

  (:init
    (at rico initial)
    (at john john-location)
    (at luke luke-location)

    (empty_robot)
    (not (not_empty_robot))

    (linked_to_location glass glass-location)
    (linked_to_location sandwich sandwich-location)
  )

  (:goal (
    ...
  )))
)
```

Należy tu zwrócić uwagę na redundancję w opisie faktu, że na robocie nie ma żadnego ładunku: (*empty_robot*) oraz (*not (not_empty_robot)*). Jest to związane z wykorzystanym planerem (opisanym szerzej w rozdziale 2.3), który nie ma wsparcia dla wprowadzania negacji w efektach akcji PDDL. Wymusza to zatem wprowadzanie do bazy wiedzy faktów domyślnie zanegowanych, których przykładem jest właśnie opis (*not_empty_robot*). Dziedzinę tego scenariusza przedstawiono na listingu 35.

Scenariusz Active Human Fall Prevention

Listing 23. Skrócony problem modelu scenariusza *Active Human Fall Prevention*

```
(define (problem task)
(:domain activehumanfallprevention)
(:objects
  luke john - human
  rico - robot
  initial - robot-location
  luke-location - human-location
  wp0 wp1 wp2 wp3 wp4 wp5 wp6 - waypoint
)

(:init
  (at rico initial)
  (at luke luke-location)
  (not_human_coming)
  (not (human_coming))

  (linked_to_location luke luke-location)
)

(:goal (
  ...
))
)
```

Dziedzinę tego scenariusza przedstawiono na listingu 36.

Omówione zostały już wszelkie kluczowe elementy implementowanych problemów i dziedzin PDDL. Jak wcześniej wspomniano, w ostatnim rozdziale pracy (dotyczącym testów systemu) przytoczone będą jeszcze przykładowe cele problemów PDDL (sekcje *(:goal)*) oraz wygenerowane na ich podstawie plany w celu ich weryfikacji.

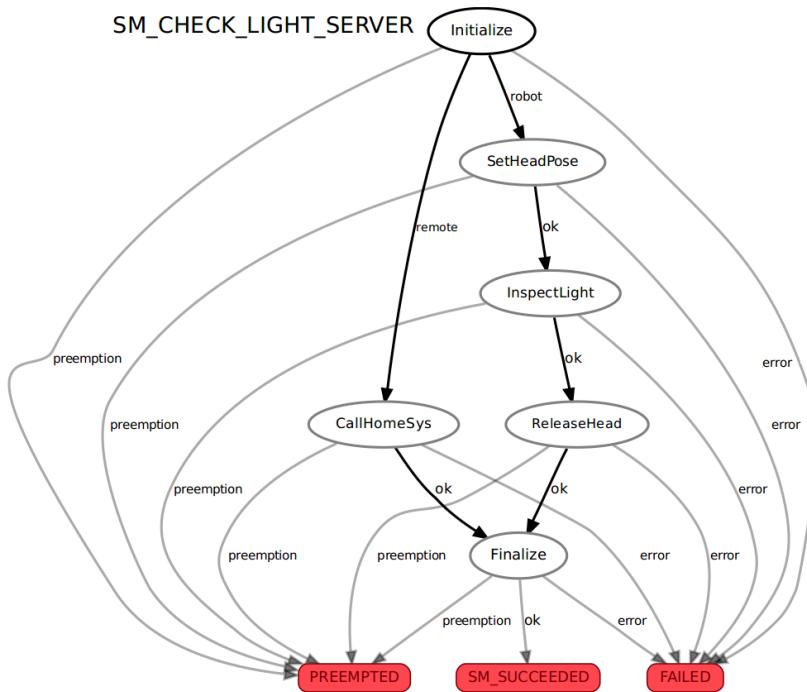
5.4. Automaty realizujące zaplanowane akcje

Wykorzystując paczkę *SMACH* zaimplementowano automaty skończone fizycznie realizujące akcje PDDL według projektu przedstawionego w rozdziale 4.2.4. Dzięki wcześniejszej znajomości możliwości tej paczki udało się w całości zaimplementować automaty według wszelkich założeń. Nie zostaną tu zatem przedstawione wszystkie utworzone automaty, podane zostaną jedynie wizualizacje niektórych z nich (jako przykład) wykonanych za pomocą programu *smach_viewer* (rysunki 45 oraz 46).

5. Implementacja systemu



Rysunek 45. Wizualizacja zaimplementowanego automatu akcji *Get Load* za pomocą programu *smach_viewer*



Rysunek 46. Wizualizacja zaimplementowanego automatu akcji *Check Light* za pomocą programu *smach_viewer*

5.5. Uzupełnianie wiedzy o świecie

Jak wspomniano w rozdziale 3.6.3 istnieją dwa sposoby na wprowadzanie zmian w stanie wiedzy przechowywanej w komponencie *Knowledge Base*. Pierwszy sposób wynika z samej budowy szkieletu ROSPlan, po pozytywnym zakończeniu danej akcji odpowiednia informacja o tym zostaje umieszczona w bazie danych. Aby wykorzystać sposób drugi polegający na modyfikacji wiedzy z poziomu pracy systemu, niezależnie od stopnia wykonania akcji należy korzystać z serwisów udostępnionych przez komponent *Knowledge Base*:

- ***rosplan_knowledge_base/update*** - umożliwia modyfikację (dodanie lub usunięcie) pojedynczego elementu w bazie danych.
- ***rosplan_knowledge_base/update_array*** - umożliwia modyfikację (dodanie lub usunięcie) wielu elementów jednocześnie.
- ***rosplan_knowledge_base/query_state*** - pozwala na wysłanie zapytania do bazy danych w celu uzyskania informacji o istnieniu danego elementu.

Wiadomości przesyłane za pomocą tych serwisów są uniwersalnymi obiektami typu *KnowledgeItem* [48]. Jest to bardzo rozbudowany i na tyle skomplikowany typ, by bezpośrednie wywoływanie serwisów z poziomu automatu akcji było zbyt uciążliwe. Zaimplementowano zatem w ramach komponentu *Knowledge Base Interface* (rysunek 25) prosty interfejs upraszczający tę komunikację. W jego skład wchodzą dwie klasy:

— ***TiagoKBUpdate***

Zaimplementowano tu metody pozwalające dodawanie lub usuwanie faktów, celów lub innych elementów które budują kompletny stan wiedzy szkieletu ROSPlan.

— ***TiagoKBQuery***

Zawarto tu metody umożliwiające odpytywanie bazy danych w celu sprawdzenia, czy podany atrybut w niej istnieje.

Obie klasy zaimplementowano w ramach paczki

rosplan_tiago_common, w pliku */src/rosplan_tiago_common/tiago_kb.py*.

Na poziomie implementowanego systemu (warstwa automatu skończonego akcji) obiekt dowolnej z omawianych klas otrzymuje dostatecznie łatwo zapisany atrybut PDDL. Jest on następnie przetwarzany i parsowany do wspomnianego typu *KnowledgeItem*. Dopiero wiadomość tak sparsowana może być wysłana przez serwis do komponentu *Knowledge Base* i odpowiednio przez niego zinterpretowana. Przykład użycia jednej z tych klas przedstawiono na listingu 24. Pierwsze wywołanie przedstawia usunięcie z bazy danych celu *human_detection_ongoing*. Kolejne natomiast przedstawia dodanie do bazy odpowiednio sparametryzowanego nowego celu *human_informed*. Odpowiednie dla danych predykatów parametry powinny być zdefiniowane wcześniej w postaci obiektu klasy *OrderedDict*.

Listing 24. Przykład użycia klasy *TiagoKBUpdate*

```
tiago_kb = TiagoKBUpdate()  
# remove human_detection_ongoing  
temp_ordered_dict = OrderedDict([( "human" , "luke" )])  
tiago_kb.add_remove_goals( "human_detection_ongoing" ,  
                           [ temp_ordered_dict ] ,  
                           should_add=False)  
  
# add human_informed goal  
temp_ordered_dict = OrderedDict([( "human" , "luke" )])  
tiago_kb.add_remove_goals( "human_informed" ,  
                           [ temp_ordered_dict ] ,  
                           should_add=True)
```

Wykorzystanie klasy *TiagoKBQuery* odbywa się w podobny sposób z tą różnicą, że zwracana jest do programu informacja czy podany w parametrze element istnieje w bazie danych komponentu *Knowledge Base*.

5.6. Ponowne planowanie na podstawie nowych faktów

Jak już wcześniej wspomniano obecne są podczas pracy systemu sytuacje, w których wymagane jest ponowne uruchomienie generatora problemu PDDL (na podstawie aktualnej wiedzy zgromadzonej w bazie danych). Otrzymany nowy problem wykorzystany musi być następnie przez planer do zaplanowania nowego ciągu akcji. W rozdziale 3.6 opisano, że wszelka komunikacja ze szkieletem ROSPlan odbywa się przez wykorzystanie serwisów systemu ROS. Również w tym przypadku możliwe jest przerwanie wykonywania aktualnego planu i wykorzystanie serii serwisów (między innymi tych użytych do rozpoczęcia pracy robota - paragraf 5.2.3) w celu ponownego zaplanowania i wykonania planu. Korzystanie z nich jednak bezpośrednio z poziomu automatu akcji może być dość uciążliwe między innymi ze względu na fakt, że niektóre z nich działają w trybie blokującym, co znacznie i niepotrzebnie opóźniłoby całą procedurę. Utworzono zatem w ramach komponentu *ROSPlan Controller* (rysunek 25) interfejs służący do uruchomienia procedury przeplanowania i wykonania nowego planu z poziomu bieżącego działania dowolnej akcji. Składa się on z pary działającej w trybie klient-serwer:

- Węzeł ***rosplan_service_control_server*** - serwer.

Subskrybuje temat

rosplan_sys_control/services

i w zależności od wiadomości, która się na nim pojawi wywołuje konkretny serwis szkieletu ROSPlan, który bezpośrednio steruje działaniem pożądanych programów (m. in. generatorem problemu lub interfejsem planera). Znajduje się w paczce

rosplan_tiago_common w pliku ***/nodes/rosplan_sys_control***.

- Klasa ***ROSPlanSysControlClient*** - klient.

Zawiera metody, których wywołanie skutkuje opublikowaniem na wspomnianym wyżej temacie wiadomości specyfikującej serwis, który ma zostać uruchomiony z poziomu serwera. Znajduje się w tej samej paczce w pliku
/src/rospлан_tiago_common/rospлан_control.py

6. Badania i wyniki oparte na przedstawionych scenariuszach

6.1. Projekt badań i środowisko testowe

Dla każdego scenariusza zostanie zaproponowany problem PDDL (w szczególności część zawierająca jego cel). Zostanie przetestowane działanie planera - wygenerowany zostanie więc plan, a następnie sprawdzana będzie poprawność realizacji tego planu. Badania przeprowadzane będą z udziałem autora pracy, z naciskiem na udział w akcjach robota wymagających interakcji z człowiekiem. Określenie poprawności działania systemu będzie złożone z dwóch czynników:

- **sprawdzenie aspektów technicznych**, czyli działania systemu od strony czysto inżynierskiej. Innymi słowy potwierdzenie braku błędów oprogramowania (lub udokumentowanie błędów wykrytych) oraz odporność systemu na nieprzewidziane sytuacje. Zostanie tu także zawarte potwierdzenie poprawności planera i realizacji planu.
- **sprawdzenie aspektów percepcyjnych**, czyli działania systemu na podstawie odczuć związanych ze współpracą człowieka z robotem.

Środowisko testowe robota odpowiada temu przedstawionemu w rozdziale zawierającym implementację systemu (rozdział 5) - żaden jego element nie uległ modyfikacji. Zostanie porównana praca robota w rzeczywistości i w symulacji. Podczas testów położony zostanie jednak nacisk na pracę robota w otoczeniu docelowym czyli rzeczywistym.

6.2. Działanie interfejsu od strony technicznej - planowanie i realizacja

6.2.1. Hazard Detection

Po uruchomieniu systemu z podanym na listingu 25 problemem wygenerowany został przedstawiony na listingu 26 plan.

Listing 25. Testowy problem modelu scenariusza *Hazard Detection*

```
(define (problem task)
  (:domain hazarddetection)
  (:objects
    door_kitchen door_room door_main - hazard
    lamp_room light_kitchen dishwasher_kitchen - hazard
    rico - robot
    initial - robot-location
    door_kitchen_location door_room_location door_main_location - hazard-location
    lamp_room_location light_kitchen_location - hazard-location
    dishwasher_kitchen_location wp0 wp1 wp2 wp3 - hazard-location
    lidar rgbd ultrasonic - robot-sensor
    door_sensor window_sensor light_sensor - home-system-sensor)
```

6. Badania i wyniki oparte na przedstawionych scenariuszach

```
)  
  
(:init  
  (at rico initial)  
  (not_checking)  
  
  (linked lamp_room wp1)  
  (linked light_kitchen light_kitchen_location)  
  (linked door_room wp2)  
  (linked door_kitchen wp3)  
  (linked door_main door_main_location)  
  (linked dishwasher_kitchen wp0)  
  
  (sensor_type door_kitchen lidar)  
  (sensor_type door_room lidar)  
  (sensor_type door_main door_sensor)  
  (sensor_type lamp_room rgbd)  
  (sensor_type light_kitchen light_sensor)  
  (sensor_type dishwasher_kitchen rgbd)  
  
  ; point which sensors are on robot's platform (1 or 0)  
  (= (is_robot_sensor lidar) 1)  
  (= (is_robot_sensor rgbd) 1)  
  (= (is_robot_sensor ultrasonic) 1)  
  (= (is_robot_sensor door_sensor) 0)  
  (= (is_robot_sensor window_sensor) 0)  
  (= (is_robot_sensor light_sensor) 0)  
  
)  
  
(:goal (and  
  (checked_light lamp_room)  
  (checked_light light_kitchen)  
  (checked_door door_kitchen)  
  (checked_door door_room)  
  (checked_door door_main)  
  (checked_dishwasher dishwasher_kitchen)  
  ))  
)  
)
```

W sekcji *goal* przedstawionego problemu zapisano predykaty *checked_light*, *checked_door* oraz *checked_dishwasher* wraz z odpowiednimi parametrami. Pośrednio oznacza to, że planer powinien wygenerować plan, którego efektem wykonania ma być sprawdzenie opisanych hazardów: włączone światło, otwarte drzwi oraz otwarta zmywarka.

Jak widać planer prawidłowo wykonał swoją pracę. Robot również prawidłowo zrealizował cały plan wykonując jedna po drugiej każdą akcję. Cały plan polegał na wykonaniu

Listing 26. Plan wygenerowany na podstawie testowego celu scenariusza *Hazard Detection*

```
0.001: (go rico initial wp0) [60.000]
60.011: (check_dishwasher rico dishwasher_kitchen wp0 rgbd) [20.000]
80.021: (go rico wp0 wp3) [60.000]
140.031: (check_door rico door_kitchen wp3 lidar) [20.000]
160.041: (check_door rico door_main door_main_location door_sensor) [5.000]
165.051: (go rico wp3 wp2) [60.000]
225.061: (check_door rico door_room wp2 lidar) [20.000]
245.071: (go rico wp2 wp1) [60.000]
305.081: (check_light rico lamp_room wp1 rgbd) [20.000]
325.091: (check_light rico light_kitchen light_kitchen_location light_sensor) [5.000]
```

kilku przejazdów z punktu do punktu i w określonych miejscach przeprowadzenie sprawdzenia hazardu (według odpowiedniej akcji **CHECK**, rysunek 47).



(a) Sprawdzanie otwartych drzwi (akcja **CHECK_DOOR**)



(b) Sprawdzanie stanu światła (akcja **CHECK_LIGHT**)

Rysunek 47. Robot TIAGo podczas realizacji zadań ze scenariusza *Hazard Detection*

6.2.2. Transportation Attendant

Po uruchomieniu systemu z podanym na listingu 27 problemem wygenerowany został przedstawiony na listingu 28 plan.

Listing 27. Testowy problem modelu scenariusza *Transportation Attendant*

```
(define (problem task)
(:domain transportationattendant)
(:objects
  rico - robot
  luke john mark - human
  tea coffee sandwich plate glass - item
  initial - robot-location
```

6. Badania i wyniki oparte na przedstawionych scenariuszach

```
luke–location john–location mark–location – human–location
wp0 wp1 wp2 wp3 wp4 wp5 – human–location
glass–location sandwich–location – item–location
spaghetti–location luke–greet–location kitchen – item–location
)

(:init
  (at rico initial)
  (at john john–location)
  (at luke luke–location)
  (at mark mark–location)

  (empty_robot)
  (not (not_empty_robot))

  (linked_to_location glass glass–location)
  (linked_to_location sandwich sandwich–location)
  (linked_to_location tea kitchen)
  (linked_to_location coffee kitchen)
  (linked_to_location spoon kitchen)
  (linked_to_location plate kitchen)
)

(:goal (and
  (load_left sandwich john john–location)
  (load_left glass luke luke–location)
  (load_left plate mark luke–location)
  (load_left coffee mark luke–location)
  )
)
)
```

Sekcja *goal* przedstawionego problemu zawiera powtórzony czterokrotnie predykat *load_left* z odpowiednimi parametrami. Predykat ten oznacza, że przedmiot został przekazany odpowiedniemu człowiekowi w odpowiednim miejscu i jest to bezpośredni efekt wykonania akcji **LEAVE_LOAD** (według dziedziny zawartej na listingu 35). Oznacza to, że zapisanie takiego celu oznacza chęć przekazania kanapki, szklanki, talerza oraz kawy pewnym osobom w dwóch miejscach o nazwach *john-location* i *luke-location*.

Listing 28. Plan wygenerowany na podstawie testowego celu scenariusza *Transportation Attendant*

```
0.001: (go rico initial glass–location) [60.000]
60.011: (go rico glass–location mark–location) [60.000]
120.021: (go_with_attendance rico mark mark–location kitchen) [200.000]
320.031: (get_load rico coffee mark kitchen) [30.000]
350.041: (go_with_attendance rico mark kitchen luke–location) [200.000]
550.051: (leave_load rico coffee mark luke–location) [30.000]
580.061: (go_with_attendance rico luke luke–location kitchen) [200.000]
```

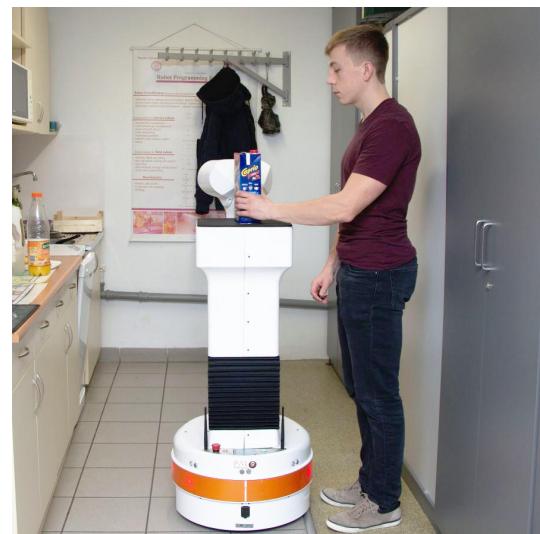
```

780.071: (go_with_attendance rico luke kitchen glass-location) [200.000]
980.081: (get_load rico glass luke glass-location) [30.000]
1010.091: (go_with_attendance rico luke glass-location luke-location) [200.000]
1210.101: (leave_load rico glass luke luke-location) [30.000]
1240.111: (go_with_attendance rico mark luke-location kitchen) [200.000]
1440.121: (get_load rico plate mark kitchen) [30.000]
1470.131: (go_with_attendance rico mark kitchen luke-location) [200.000]
1670.141: (leave_load rico plate mark luke-location) [30.000]
1700.151: (go rico luke-location john-location) [60.000]
1760.161: (go_with_attendance rico john john-location sandwich-location) [200.000]
1960.171: (get_load rico sandwich john sandwich-location) [30.000]
1990.181: (go_with_attendance rico john sandwich-location john-location) [200.000]
2190.191: (leave_load rico sandwich john john-location) [30.000]

```



(a) Prowadzenie człowieka (akcja **GO_WITH_ATTENDANCE**)



(b) Odbiór ładunku (akcja **GET_LOAD**)

Rysunek 48. Robot TIAGO podczas realizacji zadań ze scenariusza *Transportation Attendant*

Plan tego scenariusza jest znacznie dłuższy od planu przykładowego scenariusza *Hazard Detection*. Pomimo tego robot prawidłowo zrealizował cały plan wykonując po kolei wszystkie akcje przejeżdżając z punktu do punktu zachowując między innymi odpowiednie parametry jazdy podczas akcji **GO_WITH_ATTENDANCE** (rysunek 48). Podczas wykonywania akcji **GET_LOAD** i **LEAVE_LOAD** system prawidłowo sterował wysokością korpusu, położeniem głowy oraz mową. Badanie to, a w szczególności sam plan ujawnia jednak pewien błąd. Na listingu 28 powtarza się jednokrotnie akcja **GO** oraz akcja **GO_WITH_ATTENDANCE** - jest to niedoskonałość pracy wykorzystanego planera TFD. Prawidłowy optymalny plan powinien zawierać jedynie pojedyncze wystąpienia obu akcji, ponieważ każdy inny przypadek to jedynie sztuczne wydłużanie drogi robota i planu. Możliwe są dwie przyczyny przedstawionego problemu. Pierwszą z nich jest ograniczony

6. Badania i wyniki oparte na przedstawionych scenariuszach

czas planowania - jest on domyślnie ustawiony na 10 sekund, zatem po upływie tego czasu praca planera jest zatrzymywana i uznawany jest najbardziej optymalny uzyskany plan. W przypadku złożonych problemów czas ten może być zbyt krótki. Możliwe również, że w opisie modelu PDDL tego scenariusza obecny jest pewien błąd, który powoduje niepotrzebnie długą pracę planera.

6.2.3. Active Human Fall Prevention

Po uruchomieniu systemu z podanym na listingu 29 problemem wygenerowany został przedstawiony na listingu 30 plan.

Listing 29. Testowy problem modelu scenariusza *Active Human Fall Prevention*

```
(define (problem task)
  (:domain activehumanfallprevention)
  (:objects
    luke john - human
    rico - robot
    initial - robot-location
    luke-location - human-location
    wp0 wp1 wp2 wp3 wp4 wp5 wp6 - waypoint
  )

  (:init
    (at rico initial)
    (at luke luke-location)
    (not_human_coming)
    (not (human_coming))

    (linked_to_location luke luke-location)
  )

  (:goal (and
    (scanned_area wp0 wp1)
    (scanned_area wp1 wp2)
    (scanned_area wp2 wp3)
    (scanned_area wp3 wp4)
    (scanned_area wp4 wp5)
    (scanned_area wp5 wp6)
    (human_detection_ongoing luke)
  )))
)
```

Sekcja *goal* w tym przypadku zawiera predykat *scanned_area*. Jest to bezpośredni efekt wykonania akcji **GO_SCANNING**, opisującą cel przeskanowania określonych w parametrach obszarów (w poszukiwaniu przedmiotów leżących na podłodze). W sekcji tej zapisano także predykat *human_detection_ongoing*, co jest efektem wykonania akcji **HUMAN_APPROACH_DETECT**.

Listing 30. Plan wygenerowany na podstawie testowego celu scenariusza *Active Human Fall Prevention*

```

0.001: (human_approach_detect rico luke) [1000.000]
0.001: (go rico initial wp0) [60.000]
60.011: (go_scanning rico wp0 wp1) [60.000]
120.021: (go_scanning rico wp1 wp2) [60.000]
180.031: (go_scanning rico wp2 wp3) [60.000]
240.041: (go_scanning rico wp3 wp4) [60.000]
300.051: (go_scanning rico wp4 wp5) [60.000]
360.061: (go_scanning rico wp5 wp6) [60.000]

```

Plan został wygenerowany prawidłowo, robot także poprawnie wykonywał kolejne przejazdy w odpowiedniej konfiguracji. Były to między innymi opuszczona głowa (rysunek 49) oraz odpowiednio zmniejszona prędkość i przyspieszenie platformy jazdnej. Po przeskakowaniu czterech obszarów (aż do punktu o nazwie wp4) zasymulowano zbliżającego się człowieka. Działająca w tle akcja **HUMAN_APPROACH_DETECT** spowodowała natychmiastowe zaprzestanie wykonywania planu, odpowiednią modyfikację wiedzy o świecie oraz ponowne uruchomienie całej procedury związanej z wykonaniem scenariusza. Zmienioną zawartość wiedzy o świecie zauważać można na wygenerowanym ponownie problemie PDDL przedstawionym na listingu 31 (w sekcji *init*).

(a) Skanowanie obszaru (akcja **GO_SCANNING**)(b) Skanowanie obszaru (akcja **GO_SCANNING**)**Rysunek 49.** Robot TIAGo podczas realizacji zadań ze scenariusza *Active Human Fall Prevention***Listing 31.** Testowy problem modelu scenariusza *Active Human Fall Prevention* po przeplanowaniu (po wykryciu zbliżającego się człowieka)

```

(define (problem task)
(:domain activehumanfallprevention)

```

6. Badania i wyniki oparte na przedstawionych scenariuszach

```
(:objects
    luke john - human
    rico - robot
    initial - robot-location
    luke-location - human-location
    wp0 wp1 wp2 wp3 wp4 wp5 wp6 - waypoint
)

(:init
    (at rico initial)
    (at luke luke-location)
    (not (not_human_coming))
    (human_coming)

    (linked_to_location luke luke-location)

    (scanned_area wp0 wp1)
    (scanned_area wp1 wp2)
    (scanned_area wp2 wp3)
    (scanned_area wp3 wp4)
)

(:goal (and
    (human_informed luke)
    )
)
)
```

Sekcja *goal* zmodyfikowanego problemu zawiera jedynie predykat *human_informed*, który jest bezpośrednim efektem realizacji akcji **HUMAN_INTERACT**. Ponownie wygenerowany plan przedstawiono na listingu 32 i zgodnie z oczekiwaniem zawiera on jedynie akcję **HUMAN_INTERACT** poprzedzoną akcją **GO**.

Listing 32. Plan wygenerowany na podstawie testowego celu scenariusza *Active Human Fall Prevention* po przeplanowaniu

```
0.001: (go rico initial luke-location) [60.000]
60.011: (human_interact rico luke-location luke) [30.000]
```

Ponowna realizacja planu przebiegła prawidłowo. Robot wykonał podjazd do pozycji człowieka i nawiązał z nim prostą interakcję polegającą na przekazaniu informacji o wykrytych podczas skanowania pomieszczenia przedmiotach.

6.2.4. Wyniki badań

W odpowiednich warunkach wszystkie scenariusze realizowane są prawidłowo. Nadal dość nietrywialnym zadaniem jest opisywanie problemów w języku PDDL. Dużą wartość jednak wnosi doświadczenie i obycie z nim - każdy prawidłowo opisany cel jest bezbłędnie przetwarzany przez planer i generowany jest odpowiedni plan. Szkielet ROSPlan po nie-

wielkich kosmetycznych poprawkach prawidłowo wywołuje kolejne akcje wykonywalne przez robota. Stwierdzono poprawne działanie wszystkich komponentów systemu oraz robota TIAGO.

Wspomniano wyżej o odpowiednich warunkach pracy - w obecnym stanie systemu możliwe jest nieoczekiwane przerwanie działania pracy scenariusza. Jest to powodowane przez niepowodzenie ukończenia dowolnej z akcji, przerywane jest wtedy wykonywanie reszty planu. Podczas badań zauważono, że najłatwiej jest zaburzyć pracę robota przez zablokowanie mu możliwości przejazdu do zadanego punktu. Po upłynięciu odpowiednio długiego czasu lub przy braku możliwości wyznaczenia przez planer lokalny ścieżki akcja **GO** kończy się błędem. Cały plan zostaje wtedy zrealizowany tylko częściowo (do momentu przerwania). Dobrym rozwiązaniem byłoby uruchamianie w takich przypadkach procedury ponownego przeplanowania (z wykorzystaniem aktualnego stanu wiedzy) i próby realizacji nowego planu (tak jak to odbywa się w ramach scenariusza *Active Human Fall Prevention*). Taka procedura jednak nie została jeszcze zaimplementowana.

6.3. Porównanie pracy systemu z rzeczywistym robotem i symulatorem

Pierwsze próby uruchamiania oprogramowania na symulatorze, które w pełni działało na rzeczywistym robocie kończyły się niepowodzeniem. Przyczyną tego były pewne fragmenty oprogramowania zaimplementowane przez producenta na robocie prawdziwym, a niedostępne w symulatorze. Pierwszymi fragmentami implementowanego systemu, które były źródłem pewnych problemów były te związane ze sterowaniem głową i korpusem robota. Należało uwzględnić, że operacje te powinny być realizowane w symulacji w nieco inny sposób niż na robocie prawdziwym. Podobne problemy odkryto przy próbach modyfikacji parametrów planera lokalnego platformy mobilnej. Okazało się, że planer o nazwie *PALLocalPlaner*, który z powodzeniem działa na robocie nie został zaimplementowany w symulacji. Efektem tego były pewne różnice w nazwach parametrów odpowiedzialnych między innymi za maksymalną prędkość i przyspieszenia robota. System należało zatem uodpornić na kilka drobnych różnic istniejących w oprogramowaniu robota i symulatora. Poza tym jednak możliwe okazało się uruchamianie i badanie dokładnie tego samego oprogramowania korzystając zarówno z symulatora *Gazebo* jak i z prawdziwego robota.

6.4. Działanie systemu od strony percepcyjnej

Podczas przedstawionych powyżej testów każdego scenariusza robotowi towarzyszył autor niniejszej pracy. Dołożono wszelkich starań, by w trakcie badań wykonywać działania podobne do tych, które mogłyby wykonywać osoba trzecia (nieznająca struktury systemu). Polegało to głównie na wykonywaniu tylko poleceń robota.

Pierwszym wrażeniem jakie można odnieść podczas pracy systemu jest jego prostota. Robot może wykonać podczas realizacji scenariusza niedużą liczbę konkretnych akcji,

6. Badania i wyniki oparte na przedstawionych scenariuszach

których suma składa się na bardziej złożone działania. Podążanie za instrukcjami głosowymi robota (szczególnie w przypadku scenariusza *Transportation Attendant*) pozwala na ich zrozumienie i współpracę z nim. Dzięki sterowaniu pozycją głowy i symulowaniu mowy robot może z łatwością przykuć do siebie uwagę i nawiązywać nieskomplikowane interakcje z człowiekiem. Między innymi mogą to być prośby o postawienie lub zdjęcie przedmiotu z robota czy też głosowe zezwolenie na wykorzystanie robota jako podparcia podczas przemieszczania się. Nieodczuwalny jest także brak dodatkowego efektora robota. Wszystko co mogłoby być wykonane przy jego pomocy może zostać zrealizowane przez towarzyszącego robotowi człowieka. Pozwala to rozwijanie współpracy z robotem, poznawanie jego możliwości. Wszelkie skutki takiego podejścia mogą zatem być tylko pozytywne.

Przy niektórych przejściach do kolejnych akcji realizowanych przez robota można jedynie odczuć niewielkie opóźnienia. Nie są to jednak sytuacje nieuniknione, dalsze rozwijanie systemu będzie wprowadzać optymalizacje również na poziomie współpracy z człowiekiem.

7. Podsumowanie

7.1. Zalety i ograniczenia zaimplementowanego systemu

W tej części podsumowania warto zwrócić uwagę na zalety i wady (ograniczenia) zaimplementowanego systemu. Zaczynając od pozytywnych cech:

- Struktura oprogramowania jest przejrzysta i łatwa do zrozumienia. Implementacja zgodna jest z dobrymi praktykami oprogramowania opartego na systemie ROS (tak jak to zostało wskazane w części implementacyjnej tej pracy - rozdział 5).
- Złożone zadania robota podzielone są na pojedyncze czynności, których realizacja podlega dalszemu podziałowi na konkretne stany automatów skończonych. Pozwala to na łatwą rozbudowę i całkowitą kontrolę pracy robota. Dodatkowo możliwe jest dość szybkie lokalizowanie błędów oprogramowania.
- Zaimplementowany system jest w dużej mierze niezależny od platformy sprzętowej. Całość prac przeprowadzano wykorzystując robota TIAGO, jednak możliwe jest dość sprawne przeniesienie na wiele innych platform mobilnych tego typu.
- Zaprogramowane scenariusze mogą zostać bez większych modyfikacji uruchamiane zarówno na platformie rzeczywistej jak i w symulatorze.

Jeśli chodzi o wady omawianej implementacji należy wspomnieć o wielu nadal brakujących elementach, które w dużym stopniu rozwinięłyby jej możliwości.

- Brak jest implementacji wykrywania ludzi z wykorzystaniem wizji. Robot nie ma zatem możliwości „spoglądania” na niego podczas nawiązywania interakcji.
- Nie zostało zaimplementowane wykrywanie mowy człowieka. Jakkolwiek kontakt z nim jest jednostronny - jedynie robot może przekazywać informacje człowiekowi.
- Nie zostały zaimplementowane niektóre kluczowe dla samych scenariuszy funkcje. Przykładowo robot pozbawiony jest możliwości rzeczywistego wykrywania hazardów - istnieje półki co „szkielet” oprogramowania, który należałoby wypełnić odpowiednią implementacją. Prace nad takimi elementami nie leżały jednak ścisłe w zakresie tej pracy.

Należy także przypomnieć o ograniczeniach związanych z samym planowaniem symbolicznym. Z uwagi na charakter planowania symbolicznego nie da się jawnie zaproponować kolejności akcji, co oznacza też, że zrealizowanie tak trywialnego zadania jak powrót do stacji ładowania po zakończeniu scenariusza jest problemowe. Może to być oczywiście wykonane poprzez odpowiednią manipulację predykatami zapisanymi jako efekt głównych celów danego problemu. Predykaty takie mogą być zapisane następnie wśród wymagań akcji zadokowania, co spowoduje odpowiednie umieszczenie tej akcji przez planer w planie dopiero po akcjach realizujących cele główne. Kłopotliwe jest także opisywanie modeli planowania symbolicznego (problemy i dziedziny) z uwagi na brak

7. Podsumowanie

narzędzi wspomagających ten proces. W przypadku wystąpienia błędów zarówno logicznych jak i w składni planer kończy pracę błędem - często bez żadnych informacji o jego przyczynie. Rozbudowa istniejących modeli scenariuszy lub prototypowanie kolejnych jest zatem czynnością czasochłonną i sprawiającą problemy.

7.2. Możliwości rozwoju

Przedstawione w tej pracy planowanie symboliczne zakłada, że środowisko, w którym realizowany jest plan jest statyczne. Wiadomo oczywiście, że omawiane środowisko robota nie spełnia tego założenia. Objawia się to często brakiem możliwości zrealizowania planu w zmieniających się okolicznościach. Ponadto w przypadku niepowodzenia wykonania pojedynczej akcji często nie jest możliwa realizacja celu aktualnego scenariusza. Należy zatem przede wszystkim rozszerzać omawiany system o możliwości związane z ciągłą weryfikacją stanu otoczenia i wykorzystywanie tej informacji w celu aktualizacji planu. Takie funkcje umożliwią także odpowiednią reakcję na zakończone niepowodzeniem akcje. Jedną z podstawowych reakcji mogłyby być uwzględnienie nowego stanu otoczenia robota, ponowne uruchomienie planera i rozpoczęcie wykonywania planu uwzględniającego nową wiedzę.

Prace, jakie należałyby wykonać w przyszłości powinny być również związane z dalszą implementacją funkcji realizujących konkretne czynności robota. Byłoby to w szczególności wykorzystanie widzenia maszynowego w celu realizacji akcji związanych z wykrywaniem hazardów oraz wykrywaniem dokładnej pozycji człowieka (między innymi, by nawiązać z nim „kontakt wzrokowy”). Należałyby także zaimplementować (wykorzystując tę samą sensorykę) wykrywanie potencjalnie niebezpiecznych przedmiotów leżących na podłodze (w ramach scenariusza *Active Human Fall Prevention*).

Wśród kolejnych prac związanych z rozwojem projektu należy wyróżnić implementację przetwarzania głosu człowieka. Jest to potrzebne, by możliwe było uruchamianie scenariuszy i celów z nimi związanych z poziomu interfejsu głosowego, a w przyszłości także wymiana informacji i faktów pomiędzy człowiekiem, a robotem.

Należy także rozwijać bazę wiedzy tak, by poszerzyć możliwości pozyskiwania informacji o ludziach będących w otoczeniu robota. Powinny to być takie dane jak wysokość ludzi, ich wiek, poziom sprawności fizycznej - powinny one wpływać na zachowania robota (na przykład jego wysokość podczas niektórych akcji lub prędkość podczas prowadzenia człowieka).

Pojawił się także pomysł związany z umieszczeniem prostego fizycznego przycisku w okolicach górnej części robota. Jego wcisnięcie miałoby stanowić jednoznaczny sygnał potwierdzający zakończenie wykonywania pewnych zadań przez człowieka. Przykładem jego wykorzystania może być akcja pobrania ładunku przez robota w ramach scenariusza *Transportation Attendant*. Człowiek po jego umieszczeniu musiałby wcisnąć taki przycisk w celu potwierdzenia wszelkich działań. Przycisk mógłby być także przydatny podczas

prowadzenia człowieka w ramach akcji **GO_WITH_ATTENDANCE**. Podczas jej realizacji przycisk musiałby być nieustannie wciśnięty, a jego puszczenie oznaczałoby pozostawienie człowieka w tyle.

7.3. Wnioski

Celem pracy była realizacja systemu wykorzystującego planowanie symboliczne w robotyce przy wspieraniu osób starszych. Pracę zaimplementowano na robocie mobilnym TIAGO, a wynikiem jej realizacji miało być wyciągnięcie wniosków na temat możliwości wykorzystania planowania symbolicznego w dziedzinie opieki nad starszymi ludźmi.

Biorąc pod uwagę charakterystykę zagadnień związanych z planowaniem symbolicznym oraz doświadczenie nabyte przy realizacji niniejszej pracy udało się dojść do pewnych wniosków. Pierwszym z nich jest to, że planowanie symboliczne nie jest odpowiednim rozwiązaniem, by wykorzystywać je przy wspieraniu osób starszych (przynajmniej w kontekście przedstawionym w niniejszej pracy). Zagadnienie opieki nad starszymi ludźmi charakteryzuje się dużą zmiennością w czasie i wieloma nieprzewidzianymi zdarzeniami. Robot narażony jest na ciągłe interakcje z ludźmi i przerwania aktualnie wykonywanych zadań. System i robot muszą być nieustannie responsywne, by zachować pewną naturalność i być łatwo zaakceptowane przez ludzi. Planowanie symboliczne natomiast wymusza nieprzerwaną realizację pewnego ciągu zadań (planu), którego przerywanie i modyfikowanie jest czynnością kosztowną w czasie. Dodatkowo każda modyfikacja aktualnie wykonywanego planu (czyli jego przerwanie, ponowne uruchomienie planera i rozpoczęcie realizacji kolejnego planu) powoduje utratę sensu wykorzystania tej dziedziny w ogóle. Jej główną cechą jest gwarancja, że podążanie za planem pozwala na realizację celu w optymalny sposób. Potrzeba wielokrotnych ingerencji w taki plan oznacza, że optymalność ta jest utracona, a przede wszystkim oznacza to, że brak jest jednego, określonego celu.

Oczywiście nie należy przekreślać wykorzystania planowania symbolicznego w omanianym środowisku. Charakter tej dziedziny narzucił dokładny podział każdego scenariusza (zespołu zachowań robota) na niewielkie, trywialne akcje. Dzięki odgórnej potrzebie takiego podziału możliwe było niezmierne uproszczenie programowania zachowań robota. Możliwe zatem, że dzięki temu w ogóle udało się w krótkim czasie uruchamiać pierwsze fragmenty systemu.

Dokonano zatem podejścia do wykorzystania planowania symbolicznego w robotyce usługowej w zagadnieniu wspierania osób starszych. Próbę tę można zdecydowanie uznać za udaną, zrealizowano wszelkie założenia pracy oraz przeprowadzono niezbędne badania. Ostatecznie nabrano pewnych wątpliwości jeśli chodzi o samo wykorzystanie planowania symbolicznego w przedstawionym zagadnieniu. Pomimo to, jest ono warte uwagi, szczególnie w obliczu nadchodzących zmian społecznych i demograficznych oraz związanych z nimi nowych potrzeb.

Bibliografia

- [1] M. Gałuszka, *Siwiejąca populacja: Ekonomiczna, społeczna i etyczna waloryzacja starości*. Morgan Kaufmann Publishers Inc., 2007.
- [2] *World population ageing - highlights*. United Nations, 2019.
- [3] „Family support in graying societies”, Dostępne pod adresem <https://www.pewsocialtrends.org/2015/05/21/family-support-in-graying-societies/>.
- [4] J. Broekens, M. Heerink i H. Rosendal, „Assistive social robots in elderly care: A review”, *Gerontechnology*, t. 8, s. 94–103, kw. 2009. DOI: 10.4017/gt.2009.08.02.002.00.
- [5] „Strona internetowa robota paro”, Dostępne pod adresem <http://www.parorobots.com/>.
- [6] S. Paxton, „Changing the culture for dementia care changing the culture for dementia care”, *Nursing Older People*, t. 25, s. 9–9, lip. 2013. DOI: 10.7748/nop2013.07.25.6.9.s16.
- [7] „Strona internetowa robota stevie”, Dostępne pod adresem <https://stevietherobot.com/>.
- [8] W. Dudek, M. Węgierek, J. Karwowski, W. Szynkiewicz i T. Winiarski, „Task harmonisation for a single-task robot controller”, w *12th International Workshop on Robot Motion and Control (RoMoCo)*, K. Kozłowski, red., IEEE, 2019, s. 86–91. DOI: 10.1109/RoMoCo.2019.8787385. adr.: http://robotyka.ia.pw.edu.pl/svn/rcprg_pubs/2019/multitasking.pdf.
- [9] D. Seredyński, T. Winiarski i C. Zieliński, „Fabric: framework for agent-based robot control systems”, w *12th International Workshop on Robot Motion and Control (RoMoCo)*, K. Kozłowski, red., IEEE, 2019, s. 215–222. DOI: 10.1109/RoMoCo.2019.8787370. adr.: http://robotyka.ia.pw.edu.pl/svn/rcprg_pubs/2019/fabric.pdf.
- [10] C. Zieliński, T. Winiarski i T. Kornuta, „Agent-based structures of robot systems”, w *Trends in Advanced Intelligent Control, Optimization and Automation*, W. Mitkowski, J. Kacprzyk, K. Oprzedkiewicz i P. Skruch, red., Cham: Springer International Publishing, 2017, s. 493–502, ISBN: 978-3-319-60699-6.
- [11] „International conference on automated planning and scheduling”, 2019, Dostępne pod adresem <http://www.icaps-conference.org/>.
- [12] J. L. Bresina, A. K. Jónsson, P. H. Morris i K. Rajan, „Activity planning for the mars exploration rovers.”.
- [13] L. Larsen, J. Kim, M. Kupke i A. Schuster, „Automatic path planning of industrial robots comparing sampling-based and computational intelligence methods”, *Procedia Manufacturing*, t. 11, s. 241 –248, 2017, 27th International Conference on Flexible Automation and Intelligent Manufacturing, FAIM2017, 27-30 June 2017, Modena, Italy, ISSN: 2351-9789. DOI: <https://doi.org/10.1016/j.promfg.2017.06.030>.

7. Bibliografia

- 2017.07.237. adr.: <http://www.sciencedirect.com/science/article/pii/S2351978917304456>.
- [14] V. Sanelli, M. Cashmore, D. Magazzeni i L. Iocchi, „Short-term human-robot interaction through conditional planning and execution”, w *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, czer. 2017, s. 540–548.
 - [15] „Strona internetowa systemu ros”, Dostępne pod adresem <http://wiki.ros.org/>.
 - [16] S. Friedenthal, A. Moore i R. Steiner, *A Practical Guide to SysML: Systems Modeling Language*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, ISBN: 9780080558363, 9780123743794.
 - [17] J. Hoffmann, „Everything you always wanted to know about planning (but were afraid to ask)”, paź. 2011, s. 1–13.
 - [18] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun i D. Weld, „Pddl - the planning domain definition language”, sierp. 1998.
 - [19] „International planning competition 2018”, w, Dostępne pod adresem <https://ipc2018.bitbucket.io/>, ICAPS, 2019.
 - [20] R. Fikes i N. J. Nilsson, „Strips: A new approach to the application of theorem proving to problem solving”, w *IJCAI*, 1971.
 - [21] E. P. D. Pednault, „Adl: Exploring the middle ground between strips and the situation calculus”, w *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, s. 324–332, ISBN: 1-55860-032-9. adr.: <http://dl.acm.org/citation.cfm?id=112922.112954>.
 - [22] O. Lima, „Task planning, execution and monitoring for mobile manipulators in industrial domains”, kw. 2016.
 - [23] I. Georgievski i M. Aiello, „An overview of hierarchical task network planning”, mar. 2014.
 - [24] D. McDermott, „A heuristic estimator for means-ends analysis in planning”, kw. 1999.
 - [25] J. Penberthy i D. Weld, „Ucpop: A sound, complete, partial order planner for adl”, mar. 2002.
 - [26] M. Cox i M. Veloso, „Supporting combined human and machine planning: The prodigy 4.0 user interface version 2.0*”, paź. 1997.
 - [27] „Jędrzej potoniec, strona internetowa”, Dostępne pod adresem <http://www.cs.put.poznan.pl/jpotoniec/>.
 - [28] „Strona internetowa przedstawiająca problem układania wieży hanoi”, Dostępne pod adresem <https://pl.khanacademy.org/computing/computer-science/algorithms/towers-of-hanoi/a/towers-of-hanoi>.
 - [29] A. Moreno, „High level task planning with inference for the tiago robot”, paź. 2016.

- [30] „Strona internetowa przedstawiająca problem małpy”, Dostępne pod adresem <https://pythonawesome.com/python-framework-for-ai-planning-and-automated-programming/>.
- [31] „Strona internetowa planera optic”, Dostępne pod adresem <https://nms.kcl.ac.uk/planning/software/optic.html>.
- [32] „Strona internetowa planera popf”, Dostępne pod adresem <https://nms.kcl.ac.uk/planning/software/popf.html>.
- [33] „Strona internetowa planera ff”, Dostępne pod adresem <https://fai.cs.uni-saarland.de/hoffmann/ff.html>.
- [34] „Strona internetowa planera lpg”, Dostępne pod adresem <http://zeus.ing.unibs.it/lpg/>.
- [35] „Strona internetowa planera smtplan+”, Dostępne pod adresem <http://kcl-planning.github.io/SMTPlan/>.
- [36] „Strona internetowa planera tfd”, Dostępne pod adresem <http://gki.informatik.uni-freiburg.de/tools/tfd/downloads.html>.
- [37] P. Eyerich, R. Mattmüller i G. Röger, „Using the context-enhanced additive heuristic for temporal and numeric planning”, w *In Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 2009.
- [38] „Strona internetowa oprogramowania gazebo”, Dostępne pod adresem <http://gazebosim.org/>.
- [39] E. Drumwright, J. Hsu, N. Koenig i D. Shell, „Extending open dynamics engine for robotics simulation”, list. 2010, s. 38–50. DOI: 10.1007/978-3-642-17319-6_7.
- [40] S. Maerivoet, „Advanced computer graphics using opengl”, sty. 2001.
- [41] „Strona internetowa projektu blender”, Dostępne pod adresem <https://www.blender.org/>.
- [42] T. Dovramadjiev, *1 introduction to blender software. interface*, wrz. 2018. DOI: 10.13140/RG.2.2.18577.07524.
- [43] „Strona internetowa przedstawiająca robota tiago”, Dostępne pod adresem <https://tiago.pal-robotics.com/>.
- [44] „Strona internetowa firmy pal robotics”, Dostępne pod adresem <https://pal-robotics.com/>.
- [45] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler i A. Ng, „Ros: An open-source robot operating system”, t. 3, sty. 2009.
- [46] J. Bohren, „Smach”, Dostępne pod adresem <http://wiki.ros.org/smach>, 2018.
- [47] „Dokumentacja narzędzia smach_viewer”, Dostępne pod adresem http://wiki.ros.org/smach_viewer.
- [48] „Dokumentacja szkieletu aplikacyjnego rosplan”, Dostępne pod adresem <https://kcl-planning.github.io/ROSPlan/documentation/>, KCL-Planning, 2018.
- [49] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtós i M. Carreras, „Rosplan: Planning in the robot operating system”, *Proceedings*

7. Bibliografia

- International Conference on Automated Planning and Scheduling, ICAPS*, t. 2015, s. 333–341, sty. 2015.
- [50] „Dokumentacja komponentu parsing interface”, Dostępne pod adresem https://kcl-planning.github.io/ROSPlan//documentation/interfaces/03_parsing_interface.html.
 - [51] „Dokumentacja komponentu action interface”, Dostępne pod adresem https://kcl-planning.github.io/ROSPlan//tutorials/tutorial_10.
 - [52] „Dokumentacja komponentu knowledge interface”, Dostępne pod adresem http://kcl-planning.github.io/ROSPlan//documentation/knowledge/03_KnowledgeItem.html.
 - [53] „Strona internetowa projektu incare”, Dostępne pod adresem <http://www.aal-europe.eu/projects/incare/>.
 - [54] „Strona internetowa z dokumentacją robota tiago i symulatora”, Dostępne pod adresem <http://wiki.ros.org/Robots/TIAGo>.
 - [55] „Repozytorium zawierające źródła szkieletu rosplan”, Dostępne pod adresem <https://github.com/KCL-Planning/ROSPlan>.

Spis rysunków

1	Roboty Paro i Stevie	16
2	Strywializowane planowanie (przebieg i efekt) [17]	19
3	PDDL wyewoluował z różnych obecnych rozwiązań [22]	20
4	Problem układania wieży Hanoi [28]	24
5	Problem układania wieży Hanoi przedstawiony na grafie [29]	26
6	Problem małpy i banana [30]	27
7	Okno główne programu Gazebo [38]	31
8	Okno główne programu Blender [41]	32
9	Robot TIAGo wykorzystywany w omawianym projekcie	33
10	Elementy składające się na oprogramowanie robota TIAGo	35
11	Struktura oprogramowania sterującego robotem TIAGo	35
12	Simulator robota TIAGo w oprogramowaniu Gazebo [38]	36
13	Struktura przykładowej paczki ROS	37
14	Schemat działania biblioteki <i>actionlib</i> [15]	40
15	Efekt wykonania przedstawionego na listingach 11 oraz 12 kodu [46]	44
16	Okno programu <i>smach_viewer</i> [47] z uruchomionym przykładowym automatem (ze stanami reprezentującymi zagnieżdżone automaty)	45
17	Struktura szkieletu ROSPlan [48]	47
18	Struktura komponentu Knowledge Base szkieletu ROSPlan [48]	48
19	Struktura komponentu Problem Interface szkieletu ROSPlan [48]	48
20	Struktura komponentu Planner Interface szkieletu ROSPlan [48]	49
21	Struktura komponentu Parsing Interface szkieletu ROSPlan [48]	49
22	Graficzne przedstawienie planu Esterel.	50
23	Struktura komponentu Plan Dispatch szkieletu ROSPlan [48]	51
24	Diagram przewidywanych przypadków użycia projektowanego systemu	57
25	Schemat blokowy przedstawiający projekt całego systemu wraz z przepływem informacji	60
26	Diagram sekwencji systemu dla generycznego scenariusza	61
27	Przedstawienie komponentów systemu ROSPlan jako jeden blok na potrzeby uproszczenia diagramów sekwencji	64
28	Przykładowy diagram sekwencji po zastąpieniu komponentów szkieletu ROSPlan jednym blokiem o nazwie <i>ROSPlan system</i>	64
29	Diagram sekwencji pracy scenariusza Hazard Detection dla przykładowego planu	65
30	Diagram sekwencji pracy scenariusza Transportation Attendant dla przykładowego planu	67
31	Diagram sekwencji pracy scenariusza Active Human Fall Prevention dla przykładowego planu	69
32	Projekt automatu skońzonego akcji Go	71
33	Projekt automatu skońzonego akcji CheckDoor	72
34	Projekt automatu skońzonego akcji HumanApproachDetection	72
35	Projekt automatu skońzonego akcji CheckLight	73
36	Projekt automatu skońzonego akcji CheckWindow	74
37	Projekt automatu skońzonego akcji GoWithAttendance	75
38	Projekt automatu skońzonego akcji GetLoad	76

39 Projekt automatu skońzonego akcji LeaveLoad	77
40 Projekt automatu skońzonego akcji GoScanning	78
41 Projekt automatu skońzonego akcji HumanInteract	79
42 Model pomieszczenia, w którym testowany będzie system oraz jego rzeczywisty odpowiednik	82
43 Mapa zbudowana przez robota na podstawie rzeczywistego pomieszczenia laboratoryjnego	83
44 Struktura plików paczki <i>rosplan_tiago_transportation_attendant</i>	86
45 Wizualizacja zaimplementowanego automatu akcji <i>Get Load</i> za pomocą programu <i>smach_viewer</i>	92
46 Wizualizacja zaimplementowanego automatu akcji <i>Check Light</i> za pomocą programu <i>smach_viewer</i>	92
47 Robot TIAGo podczas realizacji zadań ze scenariusza <i>Hazard Detection</i>	99
48 Robot TIAGo podczas realizacji zadań ze scenariusza <i>Transportation Attendant</i>	101
49 Robot TIAGo podczas realizacji zadań ze scenariusza <i>Active Human Fall Prevention</i>	103

Spis tabel

1 Podstawowe parametry konstrukcyjne robota TIAGo	33
2 Podstawowe parametry komputera pokładowego robota	34
3 Lista czujników dostępnych na platformie robota	34
4 Wspierane przez szkielet ROSPlan planery	49

Spis załączników

1. Dziedziny problemów PDDL.

A. Dziedziny problemów PDDL

A.1. Dziedzina problemu małpy i banana

Listing 33. Dziedzina problemu małpy i banana.

```
(define (domain monkeyproblem)
(:requirements :adl)
(:constants monkey box knife bananas waterfountain glass)
(:predicates
  (goto ?x ?y)
  (climp ?x)
  (push-box ?x ?y)
  (get-knife ?y)
  (grab-bananas ?y)
  (getwater ?y)
  (on-floor)
  (at ?x ?y)
  (hasbananas)
  (hasknife)
  (onbox ?x)
  )

(:action goto
  :parameters (?x ?y)
  :precondition (and
    (on-floor)
    (at monkey ?y)
    )
  :effect (and
    (at monkey ?x)
    (not (at monkey ?y))
    )
  )

(:action climp
  :parameters (?x)
  :precondition (and
    (at box ?x)
    (at monkey ?x)
    )
  :effect (and
    (onbox ?x)
    (not (on-floor))
    )
  )
```

```
(:action push-box
  :parameters (?x ?y)
  :precondition (and
    (at box ?y)
    (at monkey ?y)
    (on-floor)
    )
  :effect (and
    (at monkey ?x)
    (at box ?x)
    (not (at monkey ?y))
    (not (at box ?y))
    )
  )
)

(:action get-knife
  :parameters (?y)
  :precondition (and
    (at knife ?y)
    (at monkey ?y)
    )
  :effect (and
    (hasknife)
    (not (at knife ?y))
    )
  )
)

(:action grab-bananas
  :parameters (?y)
  :precondition (and
    (hasknife)
    (at bananas ?y)
    (onbox ?y)
    )
  :effect (and
    (hasbananas)
    )
  )
)
```

A.2. Dziedzina scenariusza *Hazard Detection*

Listing 34. Dziedzina scenariusza *Hazard Detection*.

```
(define (domain hazarddetection)

(:requirements :typing :adl :durative-actions :numeric-fluents)

(:types hazard - linkable
        robot - locatable
        hazard-location robot-location - location
        robot-sensor home-system-sensor - sensor)

(:predicates (at ?obj - locatable ?loc - location)
             (checked_door ?hazard - hazard)
             (checked_light ?hazard - hazard)
             (checked_dishwasher ?hazard - hazard)
             (linked ?hazard - linkable ?loc - location)
             (sensor_type ?haz - hazard ?sen - sensor)
             (checking)
             (not_checking))

(:functions (is_robot_sensor ?sen - sensor))

(:durative-action GO
  :parameters (?obj - robot ?start - location ?destination - location)
  :duration (= ?duration 30)
  :condition (and
    (at start (at ?obj ?start))
    (over all (not_checking)))
  :effect (and
    (at end (at ?obj ?destination))
    (at start (not (at ?obj ?start)))))
)

(:durative-action CHECK_DOOR
  :parameters (?obj - robot ?hazard - hazard
              ?hazloc - hazard-location ?sen - sensor)
  :duration (= ?duration (+ (* (is_robot_sensor ?sen) 30) 5))
  :condition
  (and
    (at start (not_checking))
    (over all (sensor_type ?hazard ?sen))
    (over all (or
      (at ?obj ?hazloc)
      (= (is_robot_sensor ?sen) 0)))
    (over all (linked ?hazard ?hazloc)))
  )
  :effect (and
    (at start (not (not_checking))))
```

A. Dziedziny problemów PDDL

```
(at end (not_checking))
(at end (checked_door ?hazard)))
)

(:durative-action CHECK_LIGHT
:parameters (?obj - robot ?hazard - hazard
?hazloc - hazard-location ?sen - sensor)
:duration (= ?duration (+ (* (is_robot_sensor ?sen) 30) 5))
:condition (and
(at start (not_checking))
(over all (sensor_type ?hazard ?sen))
(over all (or
(at ?obj ?hazloc)
(= (is_robot_sensor ?sen) 0)))
(over all (linked ?hazard ?hazloc)))
:effect (and
(at start (not (not_checking)))
(at end (not_checking)))
(at end (checked_light ?hazard)))
)

(:durative-action CHECK_DISHWASHER
:parameters (?obj - robot ?hazard - hazard
?hazloc - hazard-location ?sen - sensor)
:duration (= ?duration (+ (* (is_robot_sensor ?sen) 30) 5))
:condition (and
(at start (not_checking))
(over all (sensor_type ?hazard ?sen))
(over all (or
(at ?obj ?hazloc)
(= (is_robot_sensor ?sen) 0)))
(over all (linked ?hazard ?hazloc)))
:effect (and
(at start (not (not_checking)))
(at end (not_checking)))
(at end (checked_dishwasher ?hazard)))
)
)
```

A.3. Dziedzina scenariusza *Transportation Attendant*

Listing 35. Dziedzina scenariusza *Transportation Attendant*.

```
(define (domain transportationattendant)

(:requirements :typing :adl :durative-actions :numeric-fluents)

(:types robot human - attendable
       attendable item - locatable
       human-location item-location robot-location waypoint - location)

(:predicates (empty_robot)
             (not_empty_robot)
             (item_on_robot ?item - item)
             (gave ?item - item ?human - human)
             (at ?obj - locatable ?loc - location)
             (attended ?obj - attendable ?from - location ?to - location)
             (linked_to_location ?lcl - locatable ?loc - location)
             (load_left ?item - item ?human - human ?loc - location))

(:durative-action GO
  :parameters (?obj - robot ?start - location ?destination - location)
  :duration (= ?duration 60)
  :condition (and
               (at start (at ?obj ?start))
               (over all (empty_robot)))
  :effect (and
            (at end (at ?obj ?destination))
            (at start (not (at ?obj ?start)))))
  )

(:durative-action GO_WITH_ATTENDANCE
  :parameters (?obj - robot ?human - human
               ?start - location ?destination - location)
  :duration (= ?duration 200)
  :condition (and
               (at start (at ?obj ?start))
               (at start (at ?human ?start)))
  :effect (and
            (at end (at ?obj ?destination))
            (at end (at ?human ?destination))
            (at end (attended ?human ?start ?destination)))
            (at start (not (at ?obj ?start)))
            (at start (not (at ?human ?start)))))
  )

(:durative-action GET_LOAD
  :parameters (?obj - robot ?item - item
               ?human - human ?itemloc - item-location)
```

A. Dziedziny problemów PDDL

```
:duration ( = ?duration 30)
:condition (and
            (at start (empty_robot))
            (over all (at ?obj ?itemloc))
            (over all (at ?human ?itemloc))
            (over all (linked_to_location ?item ?itemloc)))
:effect (and
          (at start (not_empty_robot))
          (at start (not (empty_robot))))
          (at end (item_on_robot ?item)))
        )
(:durative-action LEAVE_LOAD
  :parameters (?obj - robot ?item - item
               ?human - human ?destination - location)
  :duration ( = ?duration 30)
  :condition (and
              (at start (item_on_robot ?item))
              (at start (not_empty_robot))
              (over all (at ?obj ?destination))
              (over all (at ?human ?destination)))
  :effect (and
            (at end (not (not_empty_robot)))
            (at end (empty_robot))
            (at end (not (item_on_robot ?item)))
            (at end (load_left ?item ?human ?destination)))
        )
    )
)
```

A.4. Dziedzina scenariusza *Active Human Fall Prevention*

Listing 36. Dziedzina scenariusza *Active Human Fall Prevention*.

```
(define (domain activehumanfallprevention)

(:requirements :typing :adl :durative-actions :numeric-fluents)

(:types robot human - locatable
       human-location robot-location waypoint - location)

(:predicates (at ?obj - locatable ?loc - location)
             (linked_to_location ?lcl - locatable ?loc - location)
             (scanned_area ?loc_from - location ?loc_to - location)
             (human_coming)
             (not_human_coming)
             (human_detection_ongoing ?human - human)
             (human_informed ?human - human))

(:durative-action GO
  :parameters (?obj - robot ?start - location ?destination - location)
  :duration (= ?duration 60)
  :condition (and
    (at start (at ?obj ?start)))
  :effect (and
    (at end (at ?obj ?destination))
    (at end (not (at ?obj ?start))))
  )

;return false when human is coming - to do that there should b
;during this action there is new knowledge
added to KB - every time some new this is detected

(:durative-action GO_SCANNING
  :parameters (?obj - robot ?start - location ?destination - location)
  :duration (= ?duration 60)
  :condition (and
    (over all (not_human_coming))
    (at start (at ?obj ?start)))
  :effect (and
    ;return that this area has been checked
    (at end (at ?obj ?destination))
    (at end (scanned_area ?start ?destination))
    (at end (not (at ?obj ?start))))
  )

;at the end of this action fact is added to KB that human is coming
;then replanning should be executed (at the end of smach server
;so it should be
;stop dispatching -> [generate problem -> run planner -> run parser -> dispatch]
```

7. Dziedziny problemów PDDL

```
;when human detected returns true (at end (is_human_coming))
```

```
(:durative-action HUMAN_APPROACH_DETECT
  :parameters (?obj - robot ?human - human)
  :duration (= ?duration 1000)
  :condition (and
    (over all (not_human_coming)))
  :effect (and
    (at start (human_detection_ongoing ?human))
    (at end (human_coming))
    (at end (not (not_human_coming))))
  )
```

;this should read from KB about hazardous items

robot found **and** pass the info to the human after approach

```
(:durative-action HUMAN_INTERACT
  :parameters (?obj - robot ?humanloc - human-location ?human - human)
  :duration (= ?duration 30)
  :condition (and
    (at start (human_coming))
    (over all (at ?obj ?humanloc))
    (over all (linked_to_location ?human ?humanloc)))
  :effect (and
    (at end (human_informed ?human)))
  )
)
```