Praca dyplomowa inżynierska

Adam Skubis

Graficzny edytor interpretowanego języka specyfikującego zadania robotów

Graphical editor for interpratable language specifying robotic tasks

Praca dyplomowa inżynierska pod kierunkiem
dr inż. Tomasza Winiarskiego

Instytut Automatyki i Informatyki Stosowanej
Politechniki Warszawskiej

Ocena .....

.....

Podpis Przewodniczącego

Komisji Egzaminu Dyplomowego

Warszawa, Wrzesień 2011

Podziękowania

Chciałbym podziękować wszystkim osobom, które pomogły mi w stworzeniu tej pracy. W szczególności podziękowania należą się mojemu promotorowi, Tomaszowi Winiarskiemu za wszystkie uwagi i propozycje dotyczące tej pracy, jakie od niego otrzymałem. Dziękuję również innym osobom, które pomagały mi podczas pisania tej pracy - Konradowi Banachowiczowi, Maciejowi Stefańczykowi, Piotrowi Trojankowi, Rafałowi Tulwinowi i Mariuszowi Żbikowskiemu.

Streszczenie

Celem niniejszej pracy inżynierskiej było stworzenie graficznego edytora pozwalającego na tworzenie zadań robotycznych zapisywanych w języku XML zgodnych z systemem MRROC++. Wymagane było, aby istniała możliwość zapisu zadań do pliku XML zgodnego ze specyfikacją w pliku DTD, możliwość odczytu z tych plików oraz żeby poprawność zadań była sprawdzana w programie. W celu umożliwienia działania aplikacji na wielu platformach do stworzenia tego projektu został wybrany język C++ oraz platforma Qt. Stworzona aplikacja spełnia wszystkie wymagania, które zostały przed nią postawione i została zaprojektowana z myślą o wygodzie użytkowników. Testowanie aplikacji odbyło się w laboratorium zarówno na sprzęcie rzeczywistym jak i na symulatorach.

Abstract

The aim of this thesis was to create a graphical editor, which allows to create multi-robot tasks described in XML language and compatible with MRROC++ system. It was required to allow users to save tasks to XML files, load tasks from these files and to check the corectness in the program. In order to allow multi-platform usage C++ language and Qt platform were chosen. The created application meets all the requirements and was created to be user-friendly. It was tested in the laboratory both on real robots and simulator.

Spis treści

1 Wstęp

1.1 Geneza pracy

W ramach struktury ramowej MRROC++ tworzonej w Instytucie Automatyki i Informatyki Stosowanej Politechniki Warszawskiej w 2008 roku została stworzona przez Marka Kisiela praca inżynierska umożliwiająca specyfikowanie zadań wielorobotowych w plikach XML[1] nazywana w dalszej części tej pracy zadaniem FSautomat. Z czasem okazało się, że pisanie zadań robotycznych w plikach XML jest niewygodne i umożliwia powstawanie ewentualnych błędów. W celu uniknięcia tych problemów zdecydowano się stworzyć system, który będzie pozwalał na automatyczne generowanie zadań robotycznych w edytorze graficznym, w którym będą one przedstawione w postaci automatu skończonego.

1.2 Cel pracy

Celem tej pracy jest stworzenie edytora graficznego, który będzie umożliwiał tworzenie zadań wielorobotowych dla struktury MRROC++. Aplikacja ma być zgodna z ustaleniami poczynionymi przez Marka Kisiela w programie, który pozwala na uruchamianie ich w tej strukturze. Nowostworzony program ma na celu ułatwienie tworzenia i edycji plików zawierających zadania robotyczne, a także sprawdzanie poprawności tych zadań. Dzięki temu użytkownicy nieznający języka XML oraz struktury ramowej MRROC++ będą mogli tworzyć zadania robotyczne. Pozwoli to na ułatwienie wdrażania nowych użytkowników do działania z systemem. W celu umożliwienia pracy użytkownikom różnych platform pracy do stworzenia aplikacji wybrane zostało środowisko Qt.

1.3 Układ pracy

Rozdział 2. zawiera opis Technologii użytych w tej pracy i podstawy teoretyczne automatu skończonego. W rozdziale 3. opisane zostały wszystkie elementy składowe języka definiującego zadania robotyczne stworzonego na potrzeby zadania FSautomat. Rozdział 4. opisuje założenia projektu, rozdział 5. projekt aplikacji. W rozdziale 6. opisane są szczegóły implementacji aplikacji zaś rozdział 7. podsumowuje pracę.

2 Opis wybranych technologii

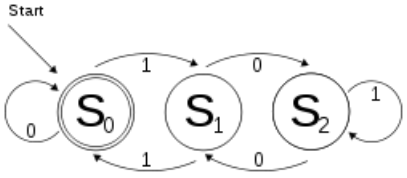
Projekt został stworzony w języku C++ przy użyciu platformy Qt do współpracy ze strukturą MRROC++. W tym rozdziale przedstawię pokrótce technologie użyte w tym projekcie.

2.1 Automat skończony

Automat skończony (ang. Finite State Automaton) jest teoretycznym tworem, służącym do opisu działań systemów. Składa się on ze stanów, przejść pomiędzy nimi (tranzycji), wejść, oraz akcji związanych ze stanami (wyjść). Dokładny matematyczny opis przedstawia krotka $A = \{I, S, O, T, W, S_0\}$, gdzie:

- **I** - zbiór sygnałów wejściowych automatu
- **S** - zbiór stanów automatu
- **O** - zbiór sygnałów wyjściowych automatu
- **T** - funkcja przejścia $T : S, I- > S$
- **W** - funkcja wyjść $W : S- > O$
- S_0 - stan początkowy automatu

W dalszej części używać będę pojęć: **Stan** - element zbioru S łączący się z pewnym działaniem systemu. **Tranzycja** - funkcja przejścia definiująca warunek przejśćia pomiędzy stanami. Poniżej przedstawiony jest graf reprezentujący automat skończony:



Rysunek 1: Przykład przedstawienia automatu jako grafu

2.1.1 Automat Moore'a

Automat Moore'a[2] jest automatem skończonym, w którym wartości na wyjściu automatu są zdefiniowane jedynie przez stan w którym znajduje się automat, nie zależą one od wejść (w przeciwieństwie do Automatu Mealy'ego, w którym funkcja wyjść przyjmuję postać $W : S, I- > O$). Poniżej przedstawiam przykładową tabelę reprezentującą automat skończony Moore'a:

State	$I = 0$	$I = 1$	Out
S_0	S_0	S_1	O_0
S_1	S_2	S_1	O_1
S_2	S_1	S_2	O_2

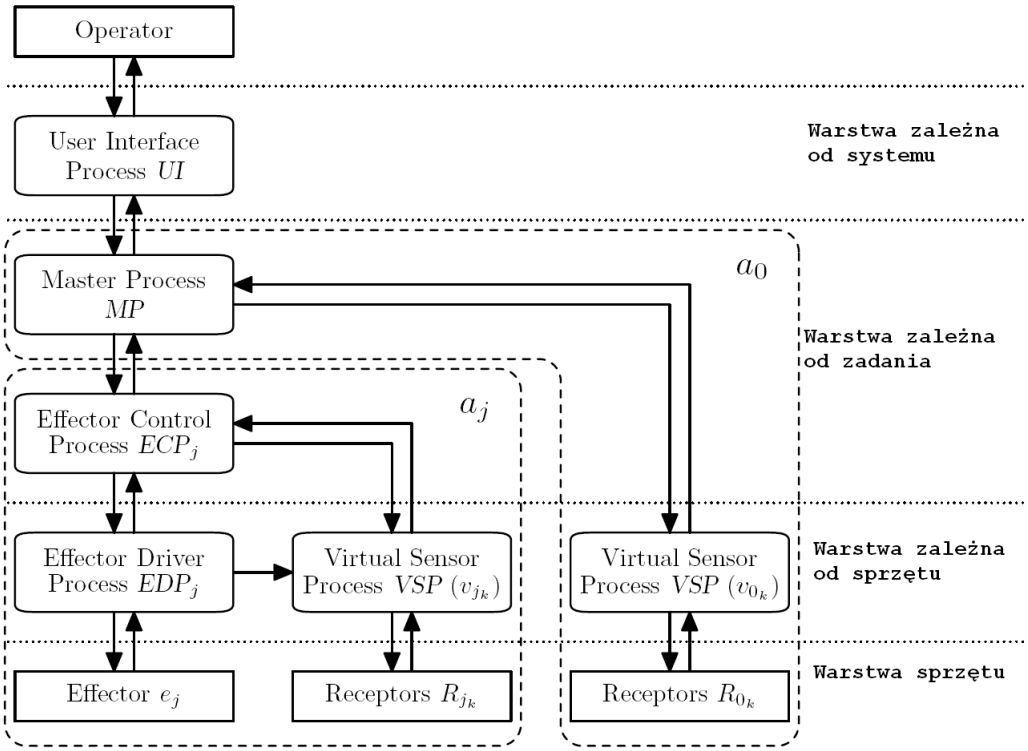
Tablica 1: Automat Moore'a przedstawiony w postaci tabeli

2.2 MRROC++

Struktura ramowa MRROC++ (Multi-Robot Research Oriented Controller)[3] jest platformą zapewniającą biblioteki modułów oraz wzorce programistyczne do tworzenia sterowników dla systemów wielorobotowych. Pozwala ona na tworzenie zadań wielorobotowych w oparciu o istniejące już sterowniki i moduły.

2.2.1 Struktura ramowa

System MRROC++ posiada hierarchiczną strukturę, dzięki czemu możliwe jest tworzenie sterowników składających się z wielu modułów. Każdemu modułowi mogą odpowiadać procesy, które uruchamiane są w systemie Linux. Dzięki takiej strukturze rozszerzanie o nowe zadanie nie wymaga pisania całości programu, jedynie potrzebnych (nowych) jego części.



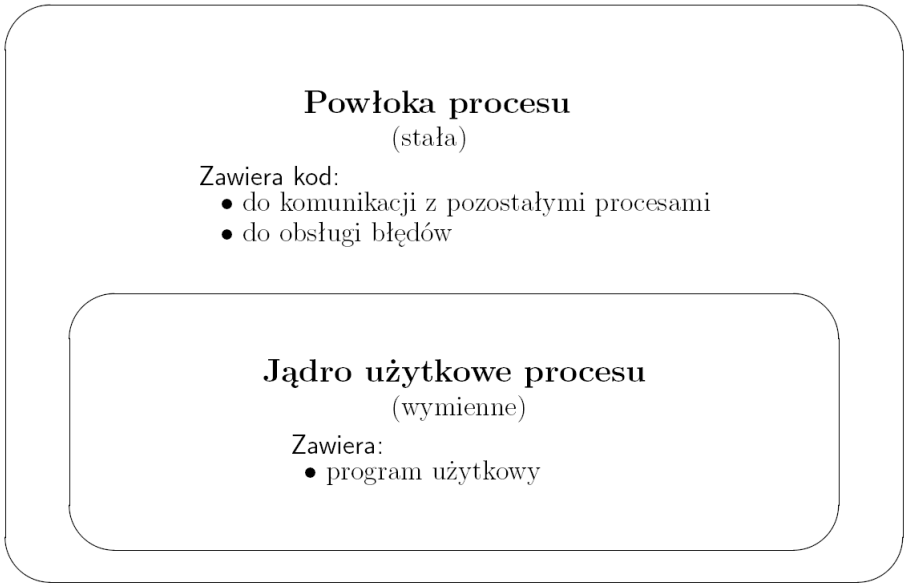
Rysunek 2: Struktura systemu MRROC++

- **UI**(User Interface) - Interfejs, przez który użytkownik komunikuje się z programem

- **MP**(Master Process) - Proces nadzorczy, kontrolujący pozostałe procesy
- **ECP**(Effector Control Process) - Proces kontrolera efektora, zależny od zadania
- **EDP**(Effector Driver Process) - Proces sterownika efektora, zależny od użytego efektora
- **VSP**(Virtual Sensor Process) - Proces czujnika wirtualnego, odpowiedzialny za odczyt i przetwarzanie danych z czujników rzeczywistych

Procesy MP, ECP oraz VSP mają wspólną strukturę:

- Jądro procesu odpowiada za realizację zadania, jest zależne od zadania.
- Powłoka procesu odpowiada za komunikację z innymi procesami i obsługę błędów.



Rysunek 3: Struktura procesu w MRROC++

W dalszym opisie będę się często odnosił również do generatorów - są to obiekty w strukturze MRROC++ które generują sterowanie dla robotów na podstawie danych z ECP i MP (np. generują kolejne kroki ruchu na podstawie zadanej trajektorii).

2.2.2 Zadanie FSautomat

W 2008 roku Marek Kisiel stworzył w ramach struktury MRROC++ zadanie FSautomat[1], które ma na celu przetwarzanie plików XML na zadania robotyczne. Zdefiniowana została struktura takiego pliku opisującego zadanie oraz sposób odczytu przez program. Miało to na celu ułatwienie tworzenia prostych zadań jak i zmniejszenie czasu testowania zadań poprzez zniesienie konieczności kompilacji po każdej zmianie. Stworzony na cele tego zadania przez Marka Kisiela opis działania systemu wielorobotowego jest automatem Moore'a

(Opis znajduje się w podrozdziale 2.1.1). Poniżej prezentuję proste zadanie, które przedstawia inicjalizacje jednego generatora ruchu dla robota IRP-6 oraz użycie tego generatora do odtworzenia trajektorii zapisanej w pliku.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE TaskDescription SYSTEM "fsautomat.dtd" >
<TaskDescription>
    <State id="INIT" type="systemInitialization">
        <taskInit>
            <ecp name="irp6p_m">
                <ecp_newsmooth_gen>6</ecp_newsmooth_gen>
            </ecp>
        </taskInit>
        <transition condition="true" target="Move"/>
    </State>
    <State id="Move" type="set_next_ecp_state">
        <ROBOT>irp6p_m</ROBOT>
        <ECPGeneratorType>ECP_GEN_NEWSMOOTH</ECPGeneratorType>
        <TrajectoryFilePath>/home/adam/workspace/mrrocpp/src
  /application/rcsc/test/irp6p_move.trj</TrajectoryFilePath>
        <transition condition="true" target="_STOP_"/>
    </State>
</TaskDescription>
```

Wydruk 1: Prosty plik XML z zadaniem robotycznym

2.3 XML

XML (E**x**tensible **M**arkup **L**anguage) jest uniwersalnym językiem znaczników służącym do reprezentowania różnych struktur danych. Jest to uproszczona wersja SGML-a (Standard Generalized Markup Language), którego zbyt skompikowany charakter nie był potrzebny dla potrzeb zadania robotycznego. Podstawowym elementem plików XML są znaczniki, które służą do opisu danych. Wydruk 1 pokazuje przykładowy znacznik XML. Każdy znacznik w języku XML musi zawierać rozpoczynający i kończący znacznik (chociaż mogą być one połączone do jednego znacznika). Znaczniki można też zagnieżdżać w sobie, tworząc strukturę rodzic-dzieci.

```
<Imie>Jan</Imie>
```

Wydruk 2: Przykładowy znacznik XML

XML definiuje reguły tworzenia dokumentów, takie jak:

- Plik na początku musi zawierać deklarację XML, posiadająca atrybut version z wartością 1.0 lub 1.1.
- Plik musi zawierać jeden element główny, wszystkie pozostałe elementy (jeśli występują) muszą być jego potomkami (poniżej tego elementu w hierarchii rodzic-dzieci).

- Każdy znacznik otwierający musi mieć odpowiadający mu znacznik kończący.
- Elementy będące dziećmi muszą być prawidłowo zagnieżdżone w elementach będącymi ich rodzicami. Oznacza to, że oba znaczniki (początkowy i końcowy) muszą być pomiędzy znacznikami elementu będącego rodzicem.
- Znaki '<', '&' '"' oraz '>' są zastrzeżone, w związku z tym w celu ich zastąpienia używa się odpowiednio "&lt", "&amp", "&quot", "&gt".
- Dokument oprócz znaczników może zawierać komentarze, które są otoczone poprzez ciągi "<!--" i "-->", np. <!--Komentarz do danych-->.

```
<Ksiegozbior>
  <Ksiazka ISBN="1-234-567">
    <Tytul>Wielka ksiega zagadek</Tytul>
    <Autor>Antoni Zagadka</Autor>
  </Ksiazka>
  <Ksiazka ISBN="9-876-543">
    <Tytul>Jak pisac ksiazki</Tytul>
    <Autor>Tomasz Ksiazkowski</Autor>
    <Autor>Aneta Ksiazkowska</Autor>
    <Typ>Edukacyjne</Typ>
  </Ksiazka>
</Ksiegozbior>
```

Wydruk 3: Przykład pliku XML - księgozbiór

Na powyższym przykładzie widać, że element księgozbiór jest elementem głównym pliku. Posiada on dwa podelementy - Książki. Każda książka ma atrybut ISBN, oraz podelementy Tytul oraz Autor. Element Typ jest nieobowiązkowym podelementem elementu Ksiazka.

2.4 DTD

DTD (Document Type Definition) jest językiem opisującym typ dokumentu SGML. Zawiera on opis składni plików danego typu. W ramach pracy Marka Kisiela do zadania FSautomat został stworzony plik zawierający definicję składni zadania robotycznego zapisywanego w plikach XML. Plik DTD składa się głównie z elementów:

- **ELEMENT** zawiera definicję elementu i jego składowych, wraz z ich dopuszczalną liczebnością.
- **ATTLIST** definiuje atrybut dla zdefiniowanego elementu.

Na poniższym przykładzie można zobaczyć, jak wygląda plik z definicją dokumentu. Jest to prosta definicja Ksiegozbioru przedstawionego na wydruku 3.

```
<!ELEMENT Ksiegozbior (Ksiazka+)>
<!ELEMENT Ksiazka (Tytul, Autor+, Typ?)
<!ATTLIST Ksiazka ISBN CDATA #REQUIRED>
```

```
    <!ELEMENT Tytul (#PCDATA)>
5   <!ELEMENT Autor (#PCDATA)>
    <!ELEMENT Typ (#PCDATA)>
```

Wydruk 4: Definicja języka XML dla Księgozbioru

Linia pierwsza powyższego wydruku definiuje, że element Ksiegozbior składa się z przynajmniej jednego elementu typu Ksiazka. W linii drugiej znajduje się definicja elementu Ksiazka - zawiera ona Tytul, przynajmniej jednego autora i może zawierać (0 lub 1) typ. W linii trzeciej zapisane jest, że Ksiazka posiada atrybut ISBN, który jest wymagany. Linie 4-6 definiują elementy Tytul, Autor oraz Typ, nie pozwalając na żadne ich podelementy oraz zapewniając im możliwość posiadania zawartości tekstowej(PCDATA).

2.5 Platforma Qt

Aplikacja napisana w ramach tej pracy inżynierskiej została stworzona z wykorzystaniem platformy Qt[4]. Qt jest tworzone jako zestaw bibliotek i narzędzi programistycznych służących głównie do tworzenia aplikacji z interfejsem graficznym. Programy napisane z wykorzystaniem tej platformy są przenośne pomiędzy systemami operacyjnymi, co było jednym z podstawowych wymagań dotyczących tej aplikacji.

2.5.1 Mechanizm sygnałów i slotów

Głównym sposobem komunikacji pomiędzy obiektami wykorzystywanymi w aplikacji jest mechanizm sygnałów i slotów. Polega on na połączeniu ze sobą dwóch funkcji o takiej samej liście parametrów (jedna z tych funkcji musi być zdefiniowana jako sygnał, druga jako slot). Uruchomienie jednej funkcji (wyemitowanie sygnału) powoduje uruchomienie drugiej funkcji (slotu). Warunkiem takiego połączenia jest istnienie takiej samej listy argumentów dla obu funkcji (sygnału i slotu). Przykład połączenia sygnału i slotu, oraz użycia go znajduje się na poniższym wydruku:

```
1  ClassA * ObjectA = new OBjectA();//creating one class object
   ClassB * ObjectB = new OBjectB();//creating second class object
   connect(OBjectA, SIGNAL(TestSignal(int num)),
       OBjectB, SLOT(TestSlot(int num)));
5  //connecting signal with a slot
   OBjectA->emit(TestSignal(1));//signal, that starts
   //the "TestSlot" function in B class.
```

Wydruk 5: Przykład sygnału i slotu

Powyższy wydruk prezentuje stworzenie dwóch elementów i połączenie sygnału jednego obiektu z obiektem drugiego. Po takim połączeniu wywołanie sygnału z klasy ClassA w linii 5 spowoduje wykonanie funkcji TestSlot() z klasy ClassB z argumentem 1.

2.5.2 Graphics View Framework

W celu stworzenia widoku elementów znajdujących się w zadaniu zostały wykorzystane narzędzia oferowane przez Graphics View Framework. Pozwalają one na intuicyjne tworzenie sceny diagramu, na której znajdują się pewne obiekty, oraz widoku (widoków) na tę scenę. Wykorzystane przeze mnie elementy tej klasy to:

- **Klasa QGraphicsScene** reprezentująca scenę, na której znajdują się obiekty. Jest ona jedynie kontenerem przechowującym obiekty, nie odpowiada w żaden sposób za ich wygląd. Elementy znajdujące się na scenie lokalizowane są poprzez kartezjański układ współrzędnych.
- **Klasa QGraphicsItem** jest bazową klasą reprezentującą element na scenie. Każdy element posiada swoje współrzędne względem sceny, musi też posiadać funkcję go rysującą. Używając tej klasy można dowolnie tworzyć elementy o złożonym wyglądzie.
- **Klasa QGraphicsView** jest widokiem na scenę. Odpowiada ona za przesuwanie i skalowanie sceny, wyświetlanie elementów i przekazywanie sygnałów do sceny.

3 Definicja języka specyfikującego zadania robotów

W ramach swojej pracy inżynierskiej Marek Kisiel stworzył język znaczników, przy pomocy którego można było tworzyć zadania robotyczne w systemie MRROC++. Język ten pozwala na opisanie zadania robotycznego jako automatu Moore'a, czyli na definiowanie działań automatu jedynie w stanach, a nie w tranzycjach.

3.1 Opis konstrukcji zadań

Zadanie robotyczne zdefiniowane zgodnie z przyjętymi w zadaniu FSautomat zasadami składa się z:

- **Stanów** reprezentujących działania systemu robotycznego.
- **Tranzycji** stanowiących przejścia między stanami i zawierających warunki przejścia.
- **Zadania głównego** zawierającego stan początkowy i końcowy całego podzadania, a także odniesienia do podzadań.
- **Podzadań** zawierających stan końcowy i reprezentujących mniejsze części projektu, wstawianych w cykl działania robota jako sekwencje stanów wykonywaną w trakcie tranzycji. Po zakończeniu podzadania system kontynuuje zadanie, z którego został wywołany, od stanu wskazywanego jako cel przez tranzycję, która uruchomiła podzadanie.

W dalszej części tej pracy będę używał również nazwy **zadanie** do opisu jednego z podzadań lub zadania głównego.

3.2 DTD zadania robotycznego

W każdym pliku zadania robotycznego dla systemu MRROC++ powinien się znajdować element definiujący składnię zadania w celu możliwości weryfikacji poprawności składniowej pliku.

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE TaskDescription SYSTEM "fsautomat.dtd" >
```

Wydruk 6: Nagłówek pliku definiującego zadanie robotyczne

W powyższym nagłówku w 1. linii widzimy typową dla XML definicję wersji i kodowania. Linia druga przedstawia definicję, mówiącą, że korzeniem (elementem nadrzędnym) całego pliku jest element TaskDescription, zaś opis składni znajduje się w pliku fsautomat.dtd.

3.3 Element TaskDescription

Element TaskDescription jest elementem głównym pliku z definicją zadania robotycznego, zawiera definicję całego zadania. Jego opis z pliku DTD znajduje się na wydruku 8. Element ten musi zawierać przynajmniej 1 stan, może zawierać element Graphics oraz dowolną liczbę elementów Subtask oraz xi:include.

3.4 Element SubTask

Element SubTask jest elementem definiującym podzadanie. Ze względu na przejrzystość zapisu podzadanie definiowane jest jako element główny pliku definiującego podzadanie, który jest załączany do zadania głównego poprzez mechanizm xinclude.

```
1 <!ELEMENT SubTask (State+,Graphics?)>
```

Wydruk 7: Opis elementu Subtask w pliku DTD

Powyższy wydruk definiuje, że element Subtask musi zawierać przynajmniej 1 element typu State oraz może posiadać element Graphics.

3.5 Element State

Element State reprezentuje stan systemu robotycznego, a także związane z nim działanie. Jest to podstawowy element budowy zadań robotycznych dla systemu MRROC++. Poniżej przedstawiony jest zapis dotyczący stanu z pliku DTD.

```
1 <!ELEMENT TaskDescription (Graphics?State+,SubTask*,xi:include*)>
  <!ELEMENT SubTask (State+,Graphics?)>
    <!ELEMENT State ((ROBOT|SetOfRobots)?,ECPGeneratorType?,PosX?,PosY?,
    (AddArg|(TrajectoryFilePath,Trajectory))?,taskInit?,
5   Speech?,Sensor?,TimeSpan?,Parameters?,transition*)>
      <!ATTLIST State id CDATA #REQUIRED>
      <!ATTLIST State type CDATA #REQUIRED>
```

Wydruk 8: Opis elementu State w pliku DTD

Linia 1 i 2 opisuje, że element State może znajdować się wewnątrz elementu TaskDescription oraz Subtask. Linie 3-5 definiują jakie elementy mogą być dziećmi elementu State. Linie 6-7 definiują, że ten element musi mieć atrybuty id oraz type.

3.5.1 Dzieci elementu State

Jak widać na wydruku 8 element stanu może mieć wielu potomków. Postaram się pokrótce przybliżyć ich sens w tym podrozdziale, dokładny opis wraz z przykładem użycia znajduje się w dalszej części tego tego rozdziału.

- **transition** jest opcjonalnym elementem przechowującym tranzycję z danego stanu (zobacz 3.6)
- **ROBOT** jest opcjonalnym elementem przechowującym nazwę robota. Występuje zamiennie z elementem SetOfRobots (zobacz 3.7)
- **SetOfRobots** jest opcjonalnym elementem zawierającym zestaw robotów. Występuje zamiennie z elementem Robot (zobacz 3.8)
- **ECPGeneratorType** jest opcjonalnym elementem przechowującym nazwę generatora wywoływanego w danym stanie (zobacz 3.9)

- **TrajectoryFilePath** jest opcjonalnym elementem przechowującym ścieżkę do pliku z trajektorią ruchu dla generatora. Może występować z elementem Trajectory, wyklucza istnienie elementu AddArg (zobacz 3.10)
- **Trajectory** jest opcjonalnym elementem przechowującym Trajektorię ruchu robota. Może występować z elementem TrajectoryFilePath, wyklucza istnienie elementu AddArg (zobacz 3.11)
- **Speech** jest opcjonalnym elementem przechowującym wartość dla generatora mowy (zobacz 3.12)
- **AddArg** jest opcjonalnym elementem przechowującym wartość dodatkową związaną z akcją w danym stanie. Jego wystąpienie wyklucza wystąpienie elementów TrajectoryFilePath i Trajectory (zobacz 3.13)
- **Sensor** jest opcjonalnym elementem przechowującym nazwę czujnika (zobacz 3.14)
- **TimeSpan** jest opcjonalnym elementem przechowującym wartość czasu (w ms) dla danego stanu (zobacz 3.15)
- **taskInit** jest opcjonalnym elementem przechowującym dane do inicjalizacji systemu (zobacz 3.16)
- **Parameters** jest opcjonalnym elementem przechowującym dodatkowe parametry nie uwzględniony w innych elementach (zobacz 3.17)
- **PosX** jest opcjonalnym elementem zawierającym wartość składowej X pozycji graficznej stanu (zobacz 3.19)
- **PosY** jest opcjonalnym elementem zawierającym wartość składowej Y pozycji graficznej stanu (zobacz 3.20)

3.5.2 Opis atrybutów

Atrybut ID definiuje nazwę stanu. Nazwa jednoznacznie definiuje stan dla zadania robotycznego, nie może się powtarzać. Na jej podstawie stany są znajdowane w trakcie wykonywania zadania. Trzy nazwy stanów mają specjalne znaczenie:

- **INIT** Oznacza początek całego zadania, a zarazem inicjalizację całego sprzętu i podprogramów potrzebnych do działania systemu. Może on być tylko typu systemInitialization.
- **_STOP_** Oznacza koniec zadania i zakończenie wykonywania programu.
- **_END_** Oznacza koniec wykonywanego podzadania. W związku z graficzną reprezentacją stanów jest on zapisywany dla każdego zadania i jest jedynym wyjątkiem od unikalności nazw stanów. Nie reprezentuje on żadnej czynności, jedynie zdjęcie ze stosu nazwy następnego stanu zapisanego tam przy rozpoczynaniu wykonania podzadania.

Atrybut Type opisuje typ stanu, a co za tym idzie - typ akcji wykonywanej w danym stanie. Nazwa typu powiązana jest z nazwą funkcji MRROC++, która jest dla danego typu stanu uruchamiana. Obecnie w systemie używane są następujące typy stanów:

- **systemInitialization** reprezentujący inicjację generatorów dla robotów, czujników i innych potrzebnych aplikacji w systemie. Przykład:

```
1  <State id="INIT" type="systemInitialization">
     <PosX>844.000000</PosX>
     <PosY>2232.000000</PosY>
     <taskInit>
5        <mp/>
         <ecp name="irp6ot_m">
       <ecp_smooth_gen>7</ecp_smooth_gen>
       <bias_edp_force_st>0</bias_edp_force_st>
       <ecp_tff_gripper_approach_gen>8</ecp_tff_gripper_approach_gen>
10       </ecp>
         <ecp name="irp6p_m">
       <bias_edp_force_st>0</bias_edp_force_st>
       <ecp_tff_gripper_approach_gen>8</ecp_tff_gripper_approach_gen>
       <ecp_smooth_gen>6</ecp_smooth_gen>
15       </ecp>
     </taskInit>
     <transition condition="true" target="approach_1"/>
   </State>
```

Wydruk 9: Przykład stanu systemInitialization

Powyższy przykład prezentuje stan, o pozycji (844,2232) na scenie edytora, nie zawierający żadnych sensorów ani transmitterów do inicjacji. W liniach 6 i 11 zdefiniowane są dwa roboty do inicjacji (irp6ot_m i irp6p_m), dla każdego z nich używane będą trzy generatory, każdy z nich zawiera wartość numeryczną, które będzie przekazana do jego konstruktora.

- **set_next_ecp_state** reprezentujący wykonanie akcji związanej z uruchomieniem generatora dla robota. Przykład:

```
1  <State id="approach_1" type="set_next_ecp_state">
           <PosX>1184.000000</PosX>
           <PosY>2187.000000</PosY>
     <ROBOT>irp6p_m</ROBOT>
5    <ECPGeneratorType>ECP_GEN_NEWSMOOTH</ECPGeneratorType>
     <TrajectoryFilePath>/home/adam/workspace/mrrocpp/src/application
       /swarm_demo/trajectory_postument_joint.trj</TrajectoryFilePath>
     <Trajectory coordinateType="JOINT" motionType="Absolute">
       <Pose>
10       <Velocity>0.1 0.1 0.1 0.1 0.1 0.1</Velocity>
         <Accelerations>0.07 0.07 0.07 0.07 0.07 0.07</Accelerations>
         <Coordinates>-1.576715 -1.910680 0.299621 0.039053 1.570008
          -1.563285</Coordinates>
```

```
    </Pose>
  </Trajectory>
  <transition condition="true" target="approach_2"/>
</State>
```

Wydruk 10: Przykład stanu set_next_ecp_state

Powyższy przykład opisuje stan ruchu dla robota irp6p_m przy pomocy generatora ECP_GEN_NEWSMOOTH. Dla ruchu zdefiniowana jest zarówno ścieżka do pliku z trajektorią, jak i trajektoria zapisana elementami XML.

- **wait_for_task_termination** reprezentujący oczekiwanie grupy robotów na zakończenie zadania przez wszystkie roboty z tej grupy. Przykład:

```
    <State id="approach_3" type="wait_for_task_termination">
        <PosX>1284.000000</PosX>
        <PosY>2237.000000</PosY>
        <SetOfRobots>
            <FirstSet>
                <ROBOT>irp6ot_m</ROBOT>
                <ROBOT>irp6p_m</ROBOT>
            </FirstSet>
        </SetOfRobots>
        <transition condition="true" target="bias_post"/>
    </State>
```

Wydruk 11: Przykład stanu wait_for_task_termination

Na powyższym wydruku stanu wait_for_task_termination zawarty jest zestaw robotów zwierający roboty irp6ot_m i irp6p_m. Roboty te po zakończeniu wykonywania swojego obecnego zadania będą oczekiwać na siebie nawzajem i dopiero gdy oba skończą dotychczasowe akcje system przejdzie do następnego stanu.

- **emptyGen** reprezentujący uruchomienie pustego generatora dla robota. Przykład:

```
<State id="approach_7" type="emptyGen">
        <PosX>784.000000</PosX>
        <PosY>2437.000000</PosY>
  <ROBOT>irp6p_m</ROBOT>
  <transition condition="true" target="approach_8"/>
</State>
```

Wydruk 12: Przykład stanu emptyGen

Powyższy przykład prezentuje stan typu emptyGen, w którym dla robota irp6p_m zostanie uruchomiony pusty generator.

- **wait_ms** reprezentujący oczekiwanie całego systemu przez określony czas. Przykład:

```
<State id="wait2s" type="wait_ms">
  <PosX>962.000000</PosX>
  <PosY>2721.000000</PosY>
  <TimeSpan>2000</TimeSpan>
  <transition condition="true" target="track_end_aa"/>
</State>
```

Wydruk 13: Przykład stanu wait_ms

Powyższy przykład prezentuje stan typu wait_ms, w którym działanie zostanie wstrzymane na 2000 ms.

- **send_end_motion_to_ecps** reprezentujący zatrzymanie działających generatorów dla zestawu robotów. Przykład:

```
<State id="approach_7" type="send_end_motion_to_ecps">
        <PosX>1784.000000</PosX>
        <PosY>1437.000000</PosY>
        <SetOfRobots>
            <FirstSet>
                <ROBOT>irp6ot_m</ROBOT>
                <ROBOT>irp6p_m</ROBOT>
            </FirstSet>
        </SetOfRobots>
  <transition condition="true" target="approach_8"/>
</State>
```

Wydruk 14: Przykład stanu send_end_motion_to_ecps

Powyższy przykład prezentuje stan, w którym działanie robotów irp6p_m i irp6ot_m zostanie zatrzymane.

- **initiateSensorReading** reprezentujący akcję inicjowania odczytu z sensora. Przykład:

```
<State id="init\_camera\_track" type="initiateSensorReading">
        <PosX>1434.000000</PosX>
        <PosY>1437.000000</PosY>
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <transition condition="true" target="get_reading_camera_track"/>
</State>
```

Wydruk 15: Przykład stanu initiateSensorReading

Powyższy przykład prezentuje inicjację pobrania danych z sensora SENSOR_CAMERA_ON_TRACK.

- **getSensorReading** reprezentujący pobranie danych z czujnika. Przykład:

```
<State id="get\_reading\_camera\_track" type="getSensorReading">
        <PosX>1334.000000</PosX>
```

```
        <PosY>1437.000000</PosY>
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <transition condition="true" target="wait1s"/>
</State>
```

Wydruk 16: Przykład stanu getSensorReading

Wydruk 16. prezentuje pobranie danych z sensora SENSOR_CAMERA_ON_TRACK.

3.6 Element transition

Element transition reprezentuje tranzycję, czyli przejście pomiędzy stanami. Jest on potomkiem elementu State, który to definiuje stan początkowy tej tranzycji. Występuje dowolną liczbę razy, aczkolwiek dla każdego stanu poza końcowym powinien występować przynajmniej raz. Poniżej przedstawiony jest zapis dotyczący elementu transition z pliku DTD.

```
<!ELEMENT transition (#PCDATA)>
  <!ATTLIST transition condition CDATA #REQUIRED>
  <!ATTLIST transition target CDATA #REQUIRED>
```

Wydruk 17: Opis elementu transition w pliku DTD

Powyższy wydruk definiuje, że element transition nie ma żadnych podelementów, posiada dwa atrybuty: condition i target.

Tranzycje są przetwarzane przez system w kolejności, w której są zapisane w pliku XML, więc ich kolejność ma znaczenie. Każdy element transition w ramach nadrzędnego elementu State powinien mieć warunek różny od innych elementów transition. Ostatnia tranzycja dla każdego stanu powinna mieć warunek true, żeby system nie pozostał w zawieszeniu na jakimś stanie, gdy wszystkie warunki są niespełnione. Ten element nie posiada podelementów, za to ma 2 obowiązkowe atrybuty: condition oraz target. Poniżej znajduje się przykład użycia elementu transition:rzykład:

```
...
<transition condition="stateOperationResult" target="bias_post"/>
<transition condition="iniFile.irp6p_uruchomiony" target="approach_6"/>
<transition condition="true" target="move_subtask>>approach_2"/>
...
```

Wydruk 18: Przykład elementu transition

Powyższy przykład prezentuje różne użycia tranzycji. W przypadku gdy warunek stateOperationResult jest spełniony automat przejdzie do stanu bias_post W przeciwnym przypadku sprawdzany będzie warunek drugiej tranzycji: gdy będzie on spełniony automat przejdzie do stanu approach_6 - w przeciwnym przypadku (warunek true) automat wykona podzadanie zaczynając od stanu move_subtask a potem przejdzie do stanu approach_2.

3.6.1 Opis atrybutów

Atrybut condition opisuje warunek przejścia pomiędzy stanami. W systemie MRROC++ dopuszczalne jest użycie jednego z warunków:

- **true** oznaczający bezwarunkowe przejście pomiędzy stanami.
- **stateOperationResult** oznaczający wynik zakończenia działania ze stanu początkowego tranzycji.
- **iniFile.VALUE** oznaczający sprawdzenie wartości logicznej atrybutu VALUE (zdefiniowanego przez użytkownika) występującego w pliku ini zadania.

Atrybut target definiuje stan docelowy tranzycji. W najprostszym przypadku jest to nazwa stanu docelowego. Konstrukcja SubTask»State oznacza, że najpierw zostanie wykonane podzadanie od stanu SubTask aż do stanu końcowego podzadania, a następnie zadanie będzie dalej wykonywane od stanu State.

3.7 Element ROBOT

Występuje jako dziecko dwóch elementów:

- **State**, w tym przypadku występuje 0 lub 1 raz. Definiuje on robota, dla którego wykonywana jest akcja w danym stanie. Używany przy stanach typu emptyGen i set_next_ecp_state. Przykład:

```
<State id="approach_1" type="runGenerator">
  <ROBOT>irp6p_m</ROBOT>
  <ECPGeneratorType>ECP_GEN_NEWSMOOTH</ECPGeneratorType>
  <TrajectoryFilePath>/home/adam/workspace/mrrocpp/src/application
  /swarm_demo/trajectory_postument_joint.trj</TrajectoryFilePath>
  <Trajectory coordinateType="JOINT" motionType="Absolute">
    <Pose>
      <Velocity>0.1 0.1 0.1 0.1 0.1 0.1</Velocity>
      <Accelerations>0.07 0.07 0.07 0.07 0.07 0.07</Accelerations>
      <Coordinates>-1.576715 -1.910680 0.299621 0.039053 1.570008
      -1.563285</Coordinates>
    </Pose>
  </Trajectory>
  <transition condition="true" target="approach_2"/>
</State>
```

Wydruk 19: Przykład elementu Robot jako dziecka elementu State

Ruch w powyższym przykładzie zostanie wykonany dla robota irp6p_m.

- **FirstSet**, w tym przypadku występuje przynajmniej 1 raz. Jest to składowa zestawu robotów. Przykład:

```
<SetOfRobots>
            <FirstSet>
                <ROBOT>irp6ot_m</ROBOT>
                <ROBOT>irp6p_m</ROBOT>
            </FirstSet>
</SetOfRobots>
```

Wydruk 20: Przykład elementu Robot jako dziecka elementu FirstSet

W powyższym przykładzie w skład zestawu robotów wchodzą roboty irp6p_m i irp6ot_m.

Obecnie jako zawartość elementu ROBOT dopuszczalne są nazwy robotów obecnych w systemie MRROC++:

- irp6ot_m
- irp6p_m
- festival
- conveyor
- bird_hand

3.8 Element SetOfRobots

Jest dzieckiem elementu State, występuje w nim 0 lub 1 raz. Określa zestaw robotów używany w danym stanie. w obecnej wersji systemu MRROC++ wszystkie funkcje zadania FSautomat używają tylko jednego zestawu robotów, czego konsekwencją jest istnienie tylko jednego podelementu - FirstSet. Poniżej zamieszczony jest zapis z pliku DTD dotyczący elementu SetOfRobots:

```
<!ELEMENT SetOfRobots (FirstSet)>
```

Wydruk 21: Opis elementu SetOfRobots z pliku fsautomat.dtd

Powyższy wydruk opisuje, że element SetOfRobots zawiera zawsze 1 element FirstSet. Element SetOfRobots jest używany dla stanów send_end_motion_to_ecps oraz wait_for_task_termination. Przykład:

```
<State id="approach_3" type="wait_for_task_termination">
  <PosX>1284.000000</PosX>
  <PosY>2237.000000</PosY>
  <SetOfRobots>
      <FirstSet>
    <ROBOT>irp6ot_m</ROBOT>
    <ROBOT>irp6p_m</ROBOT>
      </FirstSet>
```

```
    </SetOfRobots>
10  <transition condition="true" target="bias_post"/>
</State>
```

Wydruk 22: Przykład użycia elementów SetOfRobots i FirstSet

Powyższy przykład pokazuje użycie elementu SetOfRobots w stanie wait_for_task_termination, w skład zestawu robotów wchodzą roboty irp6p_m i irp6ot_m.

3.8.1 Element FirstSet

Element FirstSet jest dzieckiem elementu SetOfRobot, występuje w nim 0 lub 1 raz. Zawiera on przynajmniej 1 podelement typu Robot, co pokazuje poniższy wyciąg z pliku DTD:

```
1 <!ELEMENT FirstSet (ROBOT+)>
```

Wydruk 23: Opis elementu FirstSet z pliku fsautomat.dtd

Powyższy wydruk definiuje, że element FirstSet składa się z jednego lub więcej elementów typu ROBOT. Przykład użycia tego elementu można zobaczyć na wydruku 22

3.9 Element ECPGeneratorType

Jest dzieckiem elementu State, w którym występuje 0 lub 1 raz. Zawiera on nazwę generatora użytego w danym stanie, występuje tylko w ramach stanu o typie set_next_ecp_state. Poniżej przedstawiony jest zapis dotyczący elementu ECPGeneratorType znajdujący się w pliku fsautomat.dtd:

```
1   <!ELEMENT ECPGeneratorType (#PCDATA)>
```

Wydruk 24: Opis elementu ECPGeneratorType z pliku fsautomat.dtd

Powyższy wydruk, definiuje, że element ECPGeneratorType nie ma podelementów, ale zawiera wartość tekstową. Wartość ta odpowiadać musi jednemu z ciągów znakowych (nazw) reprezentujących generatory dostępne w systemie MRROC++. Obecny zbiór tych nazw to:

- ECP_GEN_TEACH_IN
- ECP_GEN_NEWSMOOTH
- ECP_GEN_WEIGHT_MEASURE
- ECP_GEN_TRANSPARENT
- ECP_GEN_TFF_NOSE_RUN
- ECP_ST_BIAS_EDP_FORCE
- ECP_GEN_TFF_RUBIK_GRAB

- ECP_GEN_TFF_RUBIK_FACE_ROTATE
- ECP_ST_GRIPPER_OPENING
- ECP_GEN_TFF_GRIPPER_APPROACH
- ECP_GEN_FESTIVAL

Przykład:

```
<State id="bias_post" type="set_next_ecp_state">
        <PosX>1430.000000</PosX>
        <PosY>2239.000000</PosY>
        <ROBOT>irp6p_m</ROBOT>
        <ECPGeneratorType>ECP_ST_BIAS_EDP_FORCE</ECPGeneratorType>
        <AddArg>5</AddArg>
        <transition condition="true" target="bias_track"/>
</State>
```

Wydruk 25: Przykład użycia elementu ECPGeneratorType

Powyższy przykład prezentuje stan, którego wykonanie spowoduje uruchomienie generatora ECP_ST_BIAS_EDP_FORCE dla robota irp6p_m.

3.10 Element TrajectoryFilePath

Element TrajectorFilePath jest dzieckiem elementu State, występuje w nim 0 lub 1 raz. Definiuje on ścieżkę do pliku z trajektorią. Używany jest dla stanu o typie set_next_ecp_state dla generatora ECP_GEN_NEWSMOOTH. Jego użycie podczas wykonywania programu wymaga ustawienia zmiennej trajectory_from_xml w pliku INI na 0. Opis elementu TrajectoryFilePath z pliku DTD przedstawiony jest poniżej:

```
<!ELEMENT TrajectoryFilePath (#PCDATA)>
```

Wydruk 26: Opis elementu TrajectoryFilePath z pliku fsautomat.dtd

Zgodnie z powyższym wydrukiem element TrajectoryFilePath nie zawiera podelementów, ale posiada zawartość tekstową, która reprezentuje ścieżkę do pliku z trajektorią. Przykład:

```
<State id="post_end_aa" type="set_next_ecp_state">
        <PosX>1212.000000</PosX>
        <PosY>2721.000000</PosY>
        <ROBOT>irp6p_m</ROBOT>
        <ECPGeneratorType>ECP_GEN_NEWSMOOTH</ECPGeneratorType>
        <TrajectoryFilePath>/home/adam/workspace/mrrocpp/src/application
/swarm_demo/trajectory_postument_angle.trj</TrajectoryFilePath>
        <transition condition="true" target="empty10"/>
</State>
```

Wydruk 27: Przykład użycia elementu TrajectoryFilePath

W powyższym przykładzie element TrajectoryFilePath definiuje ścieżkę do pliku z trajektorią, która zostanie odtworzona w tym stanie przez robota irp6p_m.

3.11 Element Trajectory

Element Trajectory jest dzieckiem elementu State, występuje w nim co najwyżej raz. Opisuje on trajektorię ruchu dla robota, przedstawioną jako kolejne pozycje, do których robot ma dotrzeć. Używany jest dla stanu o typie set_next_ecp_state dla generatora ECP_GEN_NEWSMOOTH. Jego użycie podczas wykonywania programu wymaga ustawienia zmiennej trajectory_from_xml w pliku INI na "1". Opis elementu Trajectory z pliku DTD znajduje się poniżej:

```
1 <!ELEMENT Trajectory (Pose+)>
  <!ATTLIST Trajectory coordinateType CDATA #REQUIRED>
  <!ATTLIST Trajectory motionType CDATA #REQUIRED>
```

Wydruk 28: Opis elementu Trajectory z pliku fsautomat.dtd

Powyższy wydruk definiuje, że element Trajectory zawiera 1 lub więcej elementów Pose, oraz posiada 2 atrybuty: coordinateType i motionType. Przykład użycia elementu Trajectory przedstawiony jest na wydruku 30.

3.11.1 Opis atrybutów

Jak wynika z opisu elementu Trajectory z pliku fsautomat.dtd posiada on 2 obowiązkowe elementy:

- coordinateType
- motionType

Element motionType opisuje typ ruchu i ma tylko dwie dopuszczalne wartości:

- **ABSOLUTE** oznaczającą, że ruch odbywa się we współrzędnych bezwzględnych danego robota.
- **RELATIVE** oznaczającą, że ruch odbywa się względem obecnej pozycji robota.

Element coordinateType opisuje typ współrzędnych, w których definiowany jest ruch. Dopuszczalne wartości to:

- ecp_XYZ_ANGLE_AXIS
- ecp_XYZ_EULER_ZYZ
- JOINT
- MOTOR
- ECP_PF_VELOCITY

3.11.2 Element Pose

Element Pose jest podelementem Trajectory, występuje w nim przynajmniej 1 raz. Opisuje on jedną pozycję w ruchu robota. Poniżej znajduje się opis elementu Pose z pliku DTD:

```
<!ELEMENT Pose (Velocity, Accelerations, Coordinates)>
  <!ELEMENT Velocity (#PCDATA)>
  <!ELEMENT Accelerations (#PCDATA)>
  <!ELEMENT Coordinates (#PCDATA)>
```

Wydruk 29: Opis elementu Pose z pliku fsautomat.dtd

Jak wynika z powyższej definicji element Pose posiada następujące podelementy występujące zawsze tylko raz:

- **Velocity** określający maksymalne prędkości ruchu robota na wszystkich współrzędnych.
- **Accelerations** określający maksymalne przyspieszenia ruchu robota na wszystkich współrzędnych.
- **Coordinates** określający koordynaty ruchu robota.

Przykład:

```
<State id="Postument_angle_axis2" type="set_next_ecp_state">
  <ROBOT>irp6p_m</ROBOT>
  <ECPGeneratorType>ECP_GEN_NEWSMOOTH</ECPGeneratorType>
  <Trajectory coordinateType="ecp_XYZ_ANGLE_AXIS" motionType="Absolute">
    <Pose>
      <Velocity>0.1 0.1 0.1 0.1 0.1 0.1</Velocity>
      <Accelerations>0.07 0.07 0.07 0.07 0.07 0.07</Accelerations>
      <Coordinates> -0.509557 1.268231 0.676142 0 0 1.57</Coordinates>
    </Pose>
    <Pose>
      <Velocity>0.1 0.1 0.1 0.1 0.1 0.1</Velocity>
      <Accelerations>0.07 0.07 0.07 0.07 0.07 0.07</Accelerations>
      <Coordinates> -0.509527 1.508295 0.676157 0 0 1.57</Coordinates>
    </Pose>
    <Pose>
      <Velocity>0.1 0.1 0.1 0.1 0.1 0.1</Velocity>
      <Accelerations>0.07 0.07 0.07 0.07 0.07 0.07</Accelerations>
      <Coordinates> 0.010473 1.508295 0.676157 0 0 1.57</Coordinates>
    </Pose>
    <Pose>
      <Velocity>0.1 0.1 0.1 0.1 0.1 0.1</Velocity>
      <Accelerations>0.07 0.07 0.07 0.07 0.07 0.07</Accelerations>
      <Coordinates> 0.010473 1.268295 0.676157 0 0 1.57</Coordinates>
    </Pose>
  </Trajectory>
  <transition condition="true" target="wait_postgrip"/>
</State>
```

Wydruk 30: Przykład użycia elementu Trajectory

Na powyższym wydruku widać stan, w którym element robot irp6p_m będzie się poruszał przy użyciu generatora ECP_GEN_NEWSMOOTH. Ruch jest opisany względem bazy robota (motionType="Absolute"), zaś współrzędne są opisane w układzie XYZ angle axis (coordinateType="ecp_XYZ_ANGLE_AXIS"). Element Trajectory posiada 4 pozycje, każda z nich posiada elementy Velocity, Accelerations oraz Coordinates. Każdy z tych elementów posiada zawartość tekstową, którą jest 6 wartości numerycznych oddzielonych znakiem tabulacji.

3.12 Element Speech

Element Speech jest potomkiem elementu State. Zawiera on tekst wypowiadany przez robota festival. Definicja tego elementu z pliku DTD znajduje się poniżej:

```
1 <!ELEMENT Speech (#PCDATA)>
```

Wydruk 31: Opis elementu Speech z pliku fsautomat.dtd

Jak wynika z powyższego wydruku element ten nie posiada podelementów, tylko zawartość tekstową. Element ten jest przeznaczony dla stanu set_next_ecp_state dla robota festival dla generatora ECP_GEN_FESTIVAL. Przykład:

```
1 <State id="wypowiedz_1" type="set_next_ecp_state">
    <ROBOT>festival</ROBOT>
    <ECPGeneratorType>ECP_GEN_FESTIVAL</ ECPGeneratorType>
    <Speech>To jest zdanie testowe.</Speech>
5   <transition condition="true" target="approach_16"/>
  </State>
```

Wydruk 32: Przykład użycia elementu Speech

Powyższy przykład prezentuje użycie elementu Speech, w wyniku wykonania akcji związanej z tym stanem robot festival powinien wypowiedzieć zdanie "To jest zdanie testowe".

3.13 Element AddArg

Element AddArg jest podelementem elementu State. Definiuje on dodatkowy argument numeryczny związany ze stanem. Poniżej znajduje się jego definicja z pliku DTD:

```
1 <!ELEMENT AddArg (#PCDATA)>
```

Wydruk 33: Opis elementu AddArg z pliku fsautomat.dtd

Zgodnie z powyższą definicją element ten nie ma podelementów, ale zawiera wartość tekstową. Element ten jest używany obecnie jedynie dla stanu o typie set_next_ecp_state jako dodatkowy argument przekazywany dla generatora. Przykład:

```
<State id="bias_post" type="set_next_ecp_state">
  <PosX>1430.000000</PosX>
  <PosY>2239.000000</PosY>
  <ROBOT>irp6p_m</ROBOT>
  <ECPGeneratorType>ECP_ST_BIAS_EDP_FORCE</ECPGeneratorType>
  <AddArg>5</AddArg>
  <TrajectoryFilePath></TrajectoryFilePath>
  <transition condition="true" target="bias_track"/>
</State>
```

Wydruk 34: Przykład użycia elementu AddArg

Efektem wykonania akcji związanej z tym stanem w systemie MRROC++ będzie wywołanie funkcji ruchu dla generatora ECP_ST_BIAS_EDP_FORCE z argumentem 5.

3.14 Element Sensor

Element Sensor występuje w dwóch przypadkach:

- jako dziecko elementu State, definiując Sensor używany w danym Stanie, dla stanów o typach initiateSensorReading lub getSensorReading, występuje tylko raz. Przykład:

```
<State id="ica_5" type="initiateSensorReading">
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <transition condition="true" target="ica_6"/>
</State>
```

Wydruk 35: Przykład użycia elementu Sensor jako potomek elementu State

W powyższym przykładzie zainicjowany zostanie odczyt z sensora SENSOR_CAMERA_ON_TRACK.

- jako dziecko elementu MP, definiując Sensor inicjalizowany dla danego zadania, występując dowolną liczbę razy. Przykład:

```
<mp>
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <Sensor>SENSOR_CAMERA_SA</Sensor>
</mp>
```

Wydruk 36: Przykład użycia elementu Sensor jako potomek elementu mp

W powyższym przykładzie w ramach procesu MP zainicjowane zostaną sensory SENSOR_CAMERA_ON_TRACK i SENSOR_CAMERA_SA.

Wartość zapisana w elemencie Sensor może przyjmować wartości sensorów obecnych w systemie MRROC++:

- SENSOR_CAMERA_ON_TRACK

- SENSOR_CAMERA_SA

Poniżej znajduje się opis elementu sensor z pliku DTD:

```
<!ELEMENT Sensor (#PCDATA)>
```

Wydruk 37: Opis elementu Sensor z pliku fsautomat.dtd

Powyższy wydruk definiuje, że element Sensor nie ma podelementów, ale posiada zawartość tekstową.

3.15 Element TimeSpan

Element TimeSpan jest dzieckiem elementu State, występuje w nim 0 lub 1 raz. Opisuje on czas dotyczący akcji w stanie. Opis elementu znajduje się w poniższym wyciągu z pliku DTD:

```
<!ELEMENT TimeSpan (#PCDATA)>
```

Wydruk 38: Opis elementu TimeSpan z pliku fsautomat.dtd

Powyższy wydruk definiuje, że element TimeSpan nie ma podelementów, ale posiada zawartość tekstową. W obecnym działaniu systemu MRROC++ używany jest on do opisu czasu oczekiwania systemu dla stanu o typie wait_ms. Przykład:

```
<State id="wait2s" type="wait_ms">
        <PosX>962.000000</PosX>
        <PosY>2721.000000</PosY>
        <TimeSpan>2000</TimeSpan>
        <transition condition="true" target="track_end_aa"/>
</State>
```

Wydruk 39: Przykład użycia elementu TimeSpan

Wykonanie akcji związanej z powyższym stanem będzie skutkowało zatrzymaniem systemu robotycznego na 2000 ms.

3.16 Element taskInit

Element taskInit jest dzieckiem elementu State występującym 0 lub 1 raz, jest on zarezerwowany dla stanu o typie systemInitialization. Zawiera on informacje o używanych w programie robotach, generatorach, sensorach i transmitterach. Opis elementu taskInit z pliku DTD znajduje się poniżej:

```
<!ELEMENT taskInit (ecp+,mp?)>
  <!ELEMENT ecp (ecp_teach_in_gen?,ecp_newsmooth_gen?,weight_measure_gen?,
    ecp_gen_t?,ecp_tff_nose_run_gen?,
    bias_edp_force_st?,ecp_tff_rubik_grab_gen?,
    ecp_tff_rubik_face_rotate_gen?,gripper_opening?,
    ecp_tff_gripper_approach_gen?,ecp_gen_festival?)>
    <!ATTLIST ecp name CDATA #REQUIRED>
```

```
    <!ELEMENT ecp_gen_t (#PCDATA)>
    <!ELEMENT ecp_gen_festival (#PCDATA)>
    <!ELEMENT ecp_tff_nose_run_gen (#PCDATA)>
    <!ELEMENT ecp_tff_rubik_grab_gen (#PCDATA)>
    <!ELEMENT ecp_tff_gripper_approach_gen (#PCDATA)>
    <!ELEMENT ecp_tff_rubik_face_rotate_gen (#PCDATA)>
    <!ELEMENT ecp_teach_in_gen (#PCDATA)>
    <!ELEMENT bias_edp_force_st (#PCDATA)>
    <!ELEMENT ecp_newsmooth_gen (#PCDATA)>
    <!ELEMENT weight_meassure_gen (#PCDATA)>
    <!ELEMENT ecp_sub_task_gripper_opening (#PCDATA)>

  <!ELEMENT mp (Sensor*,Transmitter?)>
    <!ELEMENT Transmitter (#PCDATA)>
```

Wydruk 40: Opis elementu taskInit z pliku fsautomat.dtd

Jak wynika z powyższego wydruku element taskInit musi zawierać przynajmniej 1 element ecp i 0 lub 1 element mp. Elementy te są opisane w najbliższych podrozdziałach.

3.16.1 Element ecp

Element ecp definiuje używanego robota i generatory dla niego używane wraz z parametrami przekazywanymi do konstruktorów tych generatorów. Element ten posiada atrybut name, który określa nazwę robota, którego dana sekcja ecp dotyczy. Obecność w ramach sekcji ecp któregoś wymienionego na wydruku 40 podelementu oznacza inicjację danego generatora. Każdy element może wystąpić co najwyżej raz.

3.16.2 Element mp

Element mp definiuje jakie elementy systemu MRROC++ mają zostać stworzone w ramach procesu mp. W każdym elemencie mp może się znajdować:

- wiele elementów typu Sensor (opisanych w podrozdziale 3.14)
- 0 lub 1 element typu Transmitter, określający nazwę transmittera używanego przez program. Jego opis z pliku DTD znajduje się na wydruku 40.

Poniżej znajduje się przykład użycia elementów taskInit, mp i ecp:

```
<State id="INIT" type="systemInitialization">
        <PosX>844.000000</PosX>
        <PosY>2232.000000</PosY>m
        <taskInit>
        <mp>
      <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
      <Sensor>SENSOR_CAMERA_SA</Sensor>
    </mp>
    <ecp name="irp6ot_m">
            <ecp_smooth_gen>7</ecp_smooth_gen>
```

```
            <bias_edp_force_st>0</bias_edp_force_st>
            <ecp_tff_gripper_approach_gen>8</ecp_tff_gripper_approach_gen>
    </ecp>
    <ecp name="irp6p_m">
            <bias_edp_force_st>0</bias_edp_force_st>
            <ecp_tff_gripper_approach_gen>8</ecp_tff_gripper_approach_gen>
            <ecp_smooth_gen>6</ecp_smooth_gen>
    </ecp>
        </taskInit>
        <transition condition="true" target="approach_1"/>
</State>
```

Wydruk 41: Przykład użycia elementów taskInit mp i ecp

W powyższym przykładzie w ramach sekcji mp znajdują się dwa sensory: SENSOR_CAMERA_ON_TRACK i SENSOR_CAMERA_SA. Istnieją 2. sekcje ecp (dla robotów irp6p_m i irp6ot_m), w każdej z nich znajdują się 3 elementy reprezentujące generatory.

3.17 Element Parameters

Element Parameters jest używany do przechowywania dodatkowych parametrów dla Stanu. W obecnej formie systemu MRROC++ element ten nie jest wykorzystywany. Jego opis z pliku DTD znajduje się poniżej:

```
<!ELEMENT Parameters (#PCDATA)>
```

Wydruk 42: Opis elementu Parameters z pliku fsautomat.dtd

Jak wynika z powyższej definicji element Parameters nie ma potomków, ale posiada wartość tekstową.

3.18 Element Graphics

Element Graphics został stworzony na cele aplikacji graficznej jako podelement elementu taskDescription i Subtask. Jego celem jest opis parametrów sceny danego zadania (położenia środka sceny oraz używanej skali) w celu odtworzenia ich po wczytaniu zadania z pliku. Poniżej znajduje się opis elementu Graphics z pliku DTD:

```
<!ELEMENT Graphics (PosX, PosY, Scale?)>
```

Wydruk 43: Opis elementu Graphics z pliku fsautomat.dtd

Z powyższego wydruku wynika, że element ten zawiera elementy PosX oraz PosY określające położenie środka obszaru widocznego sceny, oraz może zawierać element Scale mówiący o skali używanej dla zadania. Przykład:

```
<Graphics>
        <PosX>-410</PosX>
        <PosY>-2058</PosY>
        <Scale>50</Scale>
```

```
5 </Graphics>
```

Wydruk 44: Przykład użycia elementu Graphics

Powyższy przykład prezentuje pozycję widocznego elementu sceny jako punkt (-410,-2058), oraz skalę 50%.

3.18.1 Element Scale

Element Scale jest dzieckiem elementu Graphics. Może w jego ramach występować 0 lub 1 raz. Określa on skalę (w %), w jakiej wyświetlana jest scena. Przykład jego użycia można znaleźć na wydruku 44. Opis elementu Scale z pliku DTD znajduje się na poniższym wydruku:

```
1 <!ELEMENT Scale (#PCDATA)>
```

Wydruk 45: Opis elementu Scale z pliku fsautomat.dtd

Powyższy wydruk definiuje, że element Scale nie posiada podelementów, ma za to zawartość tekstową.

3.19 Element PosX

Element PosX określa współrzędną X położenia punktu na scenie. Występuje on w dwóch przypadkach:

- jako dziecko elementu State, określa wtedy położenie środka figury reprezentującej stan. Przykład użycia znajduje się między innymi na wydruku 27.
- jako dziecko elementu Graphics, w tym przypadku określa położenie środka pola widocznego dla użytkownika. Przykład znajduje się na wydruku 44.

Opis elementu PosX z pliku DTD znajduje się na poniższym wydruku:

```
1 <!ELEMENT PosX (#PCDATA)>
```

Wydruk 46: Opis elementu PosX z pliku fsautomat.dtd

Powyższy wydruk definiuje, że element PosX nie posiada podelementów, ma za to zawartość tekstową.

3.20 Element PosY

Element PosY określa współrzędną Y położenia punktu na scenie. Występuje on w dwóch przypadkach:

- jako dziecko elementu State, określa wtedy położenie środka figury reprezentującej stan. Przykład użycia znajduje się między innymi na wydruku 27.
- jako dziecko elementu Graphics, w tym przypadku określa położenie środka pola widocznego dla użytkownika. Przykład znajduje się na wydruku 44.

Opis elementu PosX z pliku DTD znajduje się an poniższym wydruku:

```
1 <!ELEMENT PosY (#PCDATA)>
```

Wydruk 47: Opis elementu PosY z pliku fsautomat.dtd

Powyższy wydruk definiuje, że element PosY nie posiada podelementów, ma za to zawartość tekstową.

4 Założenia projektu

W tym rozdziale omawiam założenia projektu graficznego edytora interpretowanego języka specyfikującego zadania robotów. Założenia funkcjonalne dla aplikacji:

- **Umożliwienie graficznego tworzenia grafów skierowanych reprezentujących zadania zgodnie z wymaganiami podanymi w zadaniu FSautomat.** Powinno to być zrealizowane jako możliwość graficznego tworzenia grafów.
- **Umożliwienie edycji stanów i tranzycji.** Powinna istnieć możliwość edycji elementów grafów poprzez definiowanie atrybutów właściwych dla danych elementów.
- **Możliwość zapisywania zadań do plików XML zgodnych z definicją w pliku DTD.** Edytor musi umożliwiać opcję zapisu, w której poprawnie sformułowane zadanie powinno zostać zapisane do pliku, który może zostać wykonany przez zadanie FSautomat. W razie wykrycia błędów edytor musi poinformować o tym użytkownika i zapisać plik w celu umożliwienia użytkownikowi poprawy błędów w innej sesji działania programu.
- **Możliwość wczytania zadań zapisanych w pliku XML.** Powinna istnieć możliwość wczytania zapisanego zadania z pliku XML zapisanego przez program lub reprezentującego poprawnie zapisane zadanie.
- **Sprawdzanie poprawności automatu.** Edytor powinien sprawdzać poprawność wprowadzonych danych i ich zgodność z założeniami zadania FSautomat, oraz wyświetlać informacje obłędach i ostrzeżenia o możliwych pomyłkach użytkownikowi.

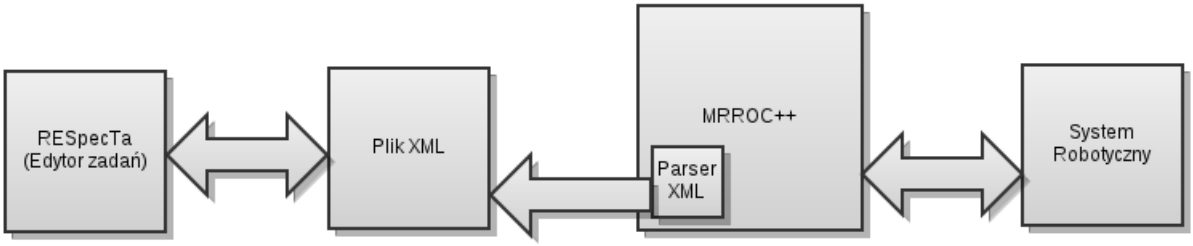
Założenia niefunkcjonalne:

1. **Wielosystemowość aplikacji.** Aplikacja powinna być dostępna dla użytkowników zarówno systemów operacyjnych typu Linux, Windows jak i MAC OS.
2. **Użycie znanej platformy graficznej w celu łatwiejszego ewentualnego rozwoju.** Platforma użyta do programowania powinna być powszechna, aby w razie potrzeby rozwoju aplikacji nie wymagało to dużych nakładów pracy.
3. **Użycie języka C++.** Jest to spowodowane użyciem tego języka w laboratorium robotyki IAiIS PW do innych celów, tj. stworzenia struktury MRROC++.
4. **Stworzenie intuicyjnego interfejsu graficznego.** Interfejs powinien być zbliżony w obsłudze do większości popularnych interfejsów pozwalających na tworzenie diagramów, tj. IBM Rational czy Altova UModel.

Konsekwencją założeń niefunkcjonalnych 1, 2 i 3 było wybranie platformy programistycznej Qt.

Dodatkowym założeniem projektu był sposób komunikacji pomiędzy edytorem i systemem MRROC++ - Uznałem, że wystarczające dla tej aplikacji będzie komunikowanie się jedynie

za pomocą plików XML zawierających zadania robotyczne. Schemat tej komunikacji można zobaczyć na poniższym diagramie:

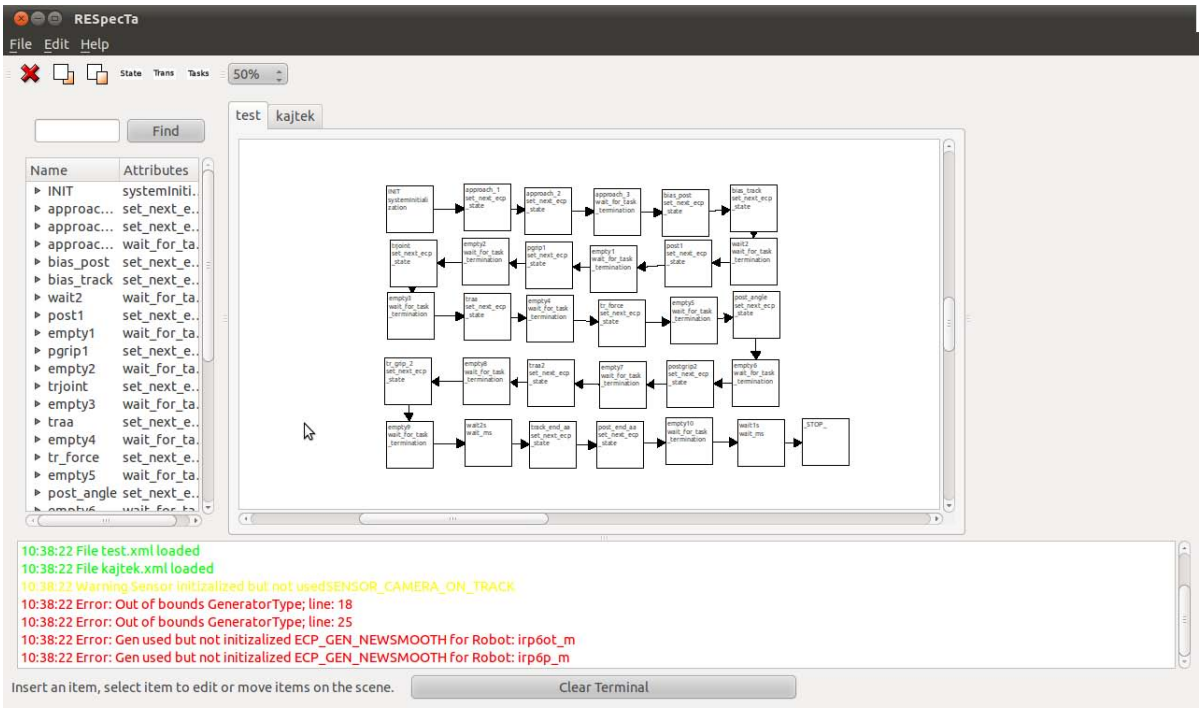


Rysunek 4: Diagram przepływu informacji pomiędzy edytorem a systemem robotycznym

Powyższy diagram prezentuje przepływ informacji od edytora (RESpecTa) do systemu robotycznego. W edytorze tworzone będą pliki, które zostaną wczytane przez system MRROC++ i następnie wykorzystane do uruchomienia zadania na systemie robotycznym.

5 Projekt aplikacji

W ramach tej pracy inżynierskiej powstała aplikacja służąca do graficznej edycji zadań robotycznych. Aplikacja ta nosi nazwę [R]obot dedicated [E]ditor for [Spec]ified [Ta]sks (RESpecTa) W tym rozdziale opisuję, w jaki sposób została zaprojektowana ta aplikacja i jakie funkcje zostały dla niej stworzone. Poniżej przedstawiam wygląd aplikacji, poszczególne jej składowe są opisane w dalszej części tego rozdziału.



Rysunek 5: Widok aplikacji RESpecTa

5.1 Wzorzec MVC

Wzorzec Model-Widok-Kontroler[5] jest wzorcem projektowym często używanym do tworzenia aplikacji GUI. Składa się on z:

- Modelu reprezentującego i operującego na danych.
- Widoku prezentującego dane.
- Kontrolera przetwarzające

W przypadku edytora będącego przedmiotem tej pracy inżynierskiej Widok został połączony z Kontrolerem, ponieważ część działań użytkownika, które w pełnym MVC powinny być przekazywane do kontrolera (np. przesuwanie elementów po scenie) jest obsługiwana przez mechanizmy natywne Qt zawarte w widoku. W związku z tym aplikacja składa się z części odpowiedzialnej za przechowywanie, zmienianie i udostępnianie danych oraz części odpowiedzialnej za wyświetlanie użytkownikowi GUI i obsługiwanie zdarzeń związanych z akcjami użytkownika.

5.2 Użycie klas boost::graph

Do opisu zadań reprezentowanych jako automat wybrałem szablon boost::graph[6]. W celu spełnienia założeń i zapewnienia pełnej funkcjonalności zadań wybrany graf musiał być skierowany, pozwalać na istnienie wielu krawędzi pomiędzy tymi samymi stanami, oraz na krawędź z danego stanu do tego samego stanu.

Konkretny opis użytego grafu przedstawia poniższy wydruk:

```
1  using namespace boost;

   struct state_t {
       typedef vertex_property_tag kind;
5    };
   struct transition_t {
       typedef edge_property_tag kind;
     };

10 typedef property<state_t, BaseState *> VertexProperty;
   typedef property<transition_t, Transition *> EdgeProperty;

   typedef adjacency_list<vecS, vecS, directedS, VertexProperty, EdgeProperty>
      MyGraphType;
```

Wydruk 48: Wyciąg z pliku Graph.h

Jak wynika z powyższego wydruku - zarówno stany dla grafu jak i krawędzie dla stanów składowane są jako obiekt typu vector. Graf jest skierowany: zarówno dla węzła jak i dla krawędzi grafu zostały zdefiniowane właściwości - wskaźniki odpowiednio na klasę BaseState i Transition.

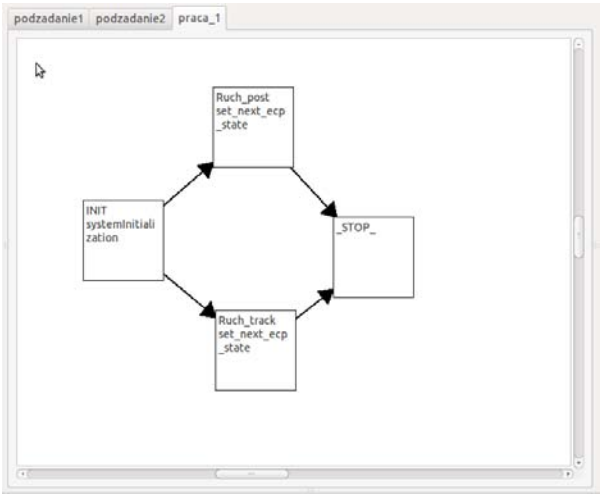
5.3 Widok zadania

W celu umożliwienia jak najbardziej intuicyjnego tworzenia modelu danych elementy w nim zawarte muszą być przedstawione w graficzny sposób.

5.3.1 Scena diagramu

Każde zadanie posiada swoją scenę (2.5.2) służącą do przedstawienia grafu. Sceny (a więc też zadania) mogą być dowolnie przełączane w trakcie działania edytora. Przedstawienie elementów na scenie pozwala na graficzne zobrazowanie tranzycji pomiędzy stanami co wydaje się najbardziej intuicyjnym sposobem ich przedstawiania. Jak widać na powyższym Rysunku 6 sceny są zakładkami, które są opisane nazwą zadania, którego dotyczą. Scena każdego zadania zawsze posiada stan końcowy (nie można go usunąć), a scena zadania początkowego zawiera również stan początkowy. Pozwala to użytkownikowi uniknąć błędu braku takiego stanu.

Scenę można przesuwać poprzez przeciąganie sceny lub użycie suwaków. Scena powiększa się, gdy elementy wysuwane są poza obszar zdefiniowany jako rozmiar sceny. Elementy znaj-

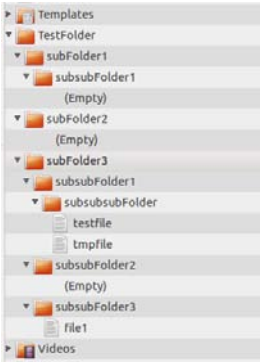


Rysunek 6: Przykładowy widok sceny

dujące się na scenie mogą być przesuwane pojedynczo lub grupami. Zaznaczenie elementu na scenie powoduje otwarcie odpowiedniego okna edycji 5.4 dla elementu. W przypadku odznaczenia elementu lub zaznaczenia więcej niż jednego elementu - okna edycji są zamykane. Elementy zaznaczone na scenie można usuwać, wysuwać na wierzch, lub chować pod spodem (gdy elementy się na siebie nakładają).

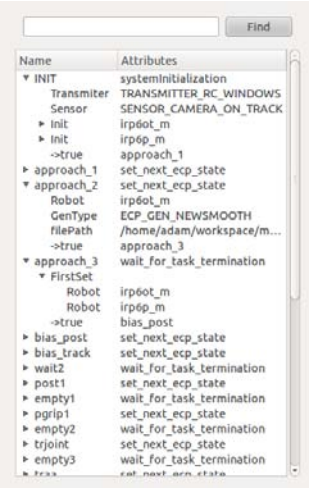
5.3.2 Lista elementów

W celu łatwiejszego znajdowania elementów w ramach danego zadania uznałem, że należy stworzyć inny model przeglądania danych. Użyłem do tego modelu widoku drzewiastego, typowego np. dla widoku folderów w aplikacjach do przeglądania danych. Cechą, która była



Rysunek 7: Przykładowy widok drzewiasty - foldery i pliki w systemie Ubuntu 10.10

decydująca dla wyboru takiego widoku jest możliwość oglądania podelementów zawartych w danym elemencie, oraz możliwość rozwijania i zwijania listy podelementów. Elementami, które są zawsze widoczne w tym widoku (podobnie jak nadrzędne foldery w widoku folderów) są elementy reprezentujące stany. Podelementami stanów są ich atrybuty - wartości charakterystyczne dla danego stanu i tranzycje. Tranzycje są wyróżniane poprzez poprzedzenie ich warunku ciągiem znaków "->". Zaznaczenie elementu na liście powoduje zaznaczenie tego elementu na scenie, oraz otworzenie go do edycji w oknie edycji. Elementy podrzędne dla stanów



Rysunek 8: Widok drzewiasty elementów przykładowego zadania w aplikacji RESpecTa

też mogą być rozwijalne, np. zestaw robotów może być rozwinięty aby pokazać znajdujące się w nim roboty.

Dla listy została też zaimplementowana funkcja znajdowania elementu po nazwie. Ma to ułatwić poszukiwanie danego stanu lub tranzycji, w przypadku długiej listy elementów. Możliwość ta jest uruchamiana przez wpisanie poszukiwanej frazy i wciśnięcie przycisku wyszukiwania.

5.4 Okna edycji

W edytorze znajdują się okna edycji, służące do zmian w istniejących elementach. W tym podrozdziale opisuję ich przeznaczenie i realizację.

5.4.1 Panel edycji stanu

Panel ten pozwala na edycję wszystkich stanów, z wyjątkiem stanów końcowych (te są nieedytowalne). Edycja rozpoczyna się, gdy do zadania (sceny) zostaje dodany nowy stan lub gdy stan został zaznaczony i odbywa się poprzez wprowadzanie nowych danych. Nie musi być ona zatwierdzana żadnym przyciskiem: w przypadku, gdy dane są poprawne są one automatycznie zapisywane gdy okno edycji jest zmieniane. Poniżej przedstawiam wygląd okna edycji dla stanu początkowego.

Panel edycji składa się z części wspólnych dla wszystkich stanów: paska edycji nazwy, opcji wybrania typu stanu, parametrów (cechy wspólne dla wszystkich stanów) i przycisku zatwierdzenia. Część środkowa panelu jest zmienna i zależy od wybranego typu stanu. Poniżej opisuję opisać panele edycji dla każdego typu stanu:

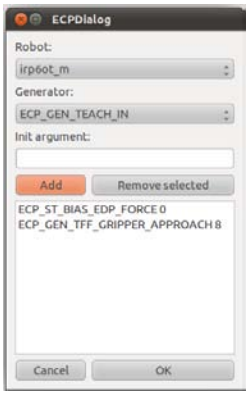
- Panel edycji stanu o typie systemInitialization, przedstawiony na rysunku 9 zawiera:
  - Listę robotów, dla których został zdefiniowany element ecp (czyli tych, których generatory zostały zdefiniowane)
  - Przycisk usuwania wybranego elementu ecp

Rysunek 9: Panel edycji stanu - edycja stanu o typie systemInitialization

– Przyciski pozwalające na otwarcie okien edycji ecp oraz mp

Poniżej przedstawiam opis i widok tych okien:

– Okno edycji ECP, które zawiera:



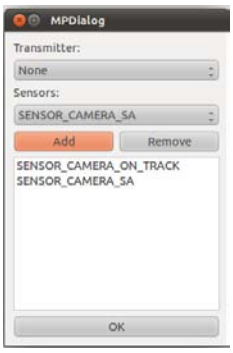
Rysunek 10: Okno edycji i tworzenia elementu ecp

  * Listę rozwijalną definiującą robota, dla którego ta sekcja ecp jest tworzona
  * Listę rozwijalną definiującą generator
  * Linię edycji do wpisywania inicjacyjnego argumentu generatora
  * Przyciski pozwalające na dodawanie wybranego generatora wraz z argumentem inicjacyjnym i usuwanie go z listy generatorów
  * Listę generatorów stworzonych dla tego robota
  * Przycisk akceptacji i odrzucenia zmian.

– Okno edycji MP, które zawiera:

  * Listę rozwijalną pozwalającą na wybór transmittera
  * Listę rozwijalną pozwalającą na wybór sensora
  * Przyciski pozwalającą na dodanie sensora do listy i usunięcie zaznaczonego sensora z listy.

* Listę wybranych sensorów
* Przycisk akceptacji zmian



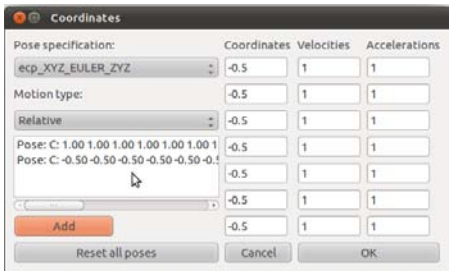
Rysunek 11: Okno edycji elementu mp

- Panel edycji stanu o typie set_next_ecp_state, przedstawiony na rysunku 12, zawiera:

Rysunek 12: Panel edycji stanu o typie set_next_ecp_state

  - Listy rozwijalne pozwalające na wybranie robota i generatora, które odpowiadają za ruch
  - Linię pozwalającą na edycję argumentu dla generatora
  - Linię pozwalającą na edycję elementu Speech
  - Linię oraz przycisk pozwalające na ustalenie ścieżki do pliku z trajektorią
  - Przycisk otwierający okno edycji trajektorii, które jest przedstawione na rysunku 13 i zawiera:
    * Listy rozwijalne pozwalające na wybór typu ruchu i współrzędnych
    * Listę zawierającą zdefiniowane pozycje
    * Okna pozwalające na specyfikację pozycji (koordynatów, prędkości max. i przyspieszeń max.)
    * Przyciski dodania pozycji i wyczyszczenia listy pozycji
    * Przyciski zatwierdzenia i odrzucenia zmian

- Identyczne panele edycji stanów o typie send_end_motion_to_ecps oraz wait_for_task_termination(te stany mają te same listy atrybutów). Widok panelu odpowiedzialnego za ich edycję przedstawiony jest na rysunku 14, zawiera:



Rysunek 13: Okno edycji trajektorii

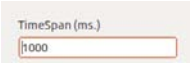
- Listę robotów wybranych do zestawu robotów
- Listę rozwijalną pozwalającą na wybór robota do dodania
- Przyciski odpowiedzialne za dodawanie Robota z listy rozwijalnej do zestawu i usuwania zaznaczonego elementu z zestawu

Rysunek 14: Panel edycji stanu wait_for_task_termination i send_end_motion_to_ecps

- Panel edycji stanu o typie emptyGen, przedstawiony na rysunku 15, zawiera:

Rysunek 15: Panel edycji stanu o typie emptyGen

  - Linię edycji argumentu
  - Listę rozwijalną pozwalającą na wybór robota
- Panel edycji stanu o typie wait_ms, przedstawiony na rysunku 16, zawiera:
  - Linię edycji elementu Timespan
- Panele edycji stanów o typie initiateSensorReading oraz getSensorReading wyglądają tak samo (te stany mają te same listy atrybutów). Widok panelu odpowiedzialnego za ich edycję przedstawiony jest na rysunku 17. Zawiera:
  - Listę rozwijalną pozwalającą na wybór sensora



Rysunek 16: Panel edycji stanu o typie wait

Rysunek 17: Panel edycji stanu o typie initiateSensorReading i getSensorReading

5.4.2 Panel edycji tranzycji

Panel edycji tranzycji pozwala na edycję właściwości tranzycji, a także na zmianę stanu początkowego i/lub końcowego tranzycji. Edycja rozpoczyna się po wstawieniu nowej tranzycji do zadania lub po zaznaczeniu istniejącej tranzycji. Panel składa się z:

- Rozwijalnej linii pozwalającej na wybór typu warunku i linii edycji warunku
- Linii rozwijalnych pozwalających na wybór podzadania i stanu, od którego podzadanie będzie wykonywane
- Linii rozwijalnych reprezentujących stan początkowy i końcowy tranzycji i pozwalające na ich zmianę

Wygląd panelu przedstawiony jest na rysunku 18.

Rysunek 18: Panel edycji tranzycji

5.4.3 Panel edycji podzadań

Panel edycji podzadań pozwala tworzyć nowe podzadania, usuwać je, zmieniać nazwy zadań i czyścić zadania. Panel składa się z:

- Listy istniejących zadań
- Linii edycji, do której wprowadza się nową nazwę zadania

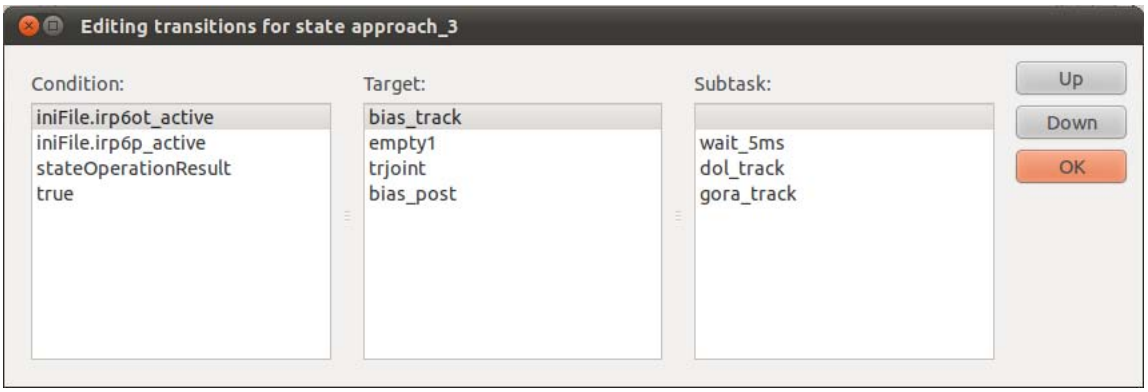
- Przycisków pozwalających na zmianę nazwy, wyczyszczenie i usunięcie zaznaczonego zadania (usunięcie zadania głównego jest niemożliwe)
- Przycisku dodania nowego podzadania.

Wygląd panelu przedstawiony jest na rysunku 19.

Rysunek 19: Panel edycji zadań

5.4.4 Okno edycji kolejności tranzycji

Okno edycji kolejności tranzycji pozwala na ustalenie kolejności tranzycji wychodzących z danego stanu. Okno to zawiera listę tranzycji, zawierającą ich warunek, stan docelowy i podzadanie oraz przyciski pozwalające na przesunięcie zaznaczonej tranzycji w górę lub w dół, a także przycisk akceptacji zmian. Wygląd tego okna jest zaprezentowany na rysunku 20.



Rysunek 20: Okno edycji kolejności tranzycji

5.5 Pozostałe elementy aplikacji

W tym podrozdziale omówię pozostałe elementy edytora wraz z ich przeznaczeniem.

5.5.1 Menu

Menu, które stworzyłem ma na celu ułatwienie pracy, przez co głównym celem stworzenia go było ułatwienie pracy użytkownikom. w związku z tym starałem się stworzyć je w sposób intuicyjny. Menu składa się z następujących podmenu:

- **File** zawierające opcje związane z aplikacją:
  - New - tworzy nowy projekt
  - Load - wczytuje projekt z pliku
  - Save - zapisuje projekt do wcześniej zdefiniowanego pliku
  - Save As - zapisuje projekt do pliku wraz z wyborem pliku
  - Quit - kończy działanie edytora.
- **Edit** zawierające opcje związane z elementami i scenami:
  - Delete - usuwa zaznaczone elementy
  - Bring to front - sprawia, że zaznaczone elementy są wysuwane przed pozostałe
  - Send to back - sprawia, że zaznaczone elementy są chowane za pozostałymi
  - Transitions - otwiera okno edycji tranzycji dla zaznaczonego stanu
  - Validate - sprawdza całość projektu pod kątem błędów i wyświetla je
  - Zoom in - Przybliża scenę
  - Zoom out - Oddala scenę
- **Help** zawierającą opcję About, otwierającą okno zawierające krótki opis działania aplikacji

5.5.2 Pasek narzędzi

Pasek narzędzi ma na celu stworzenie łatwego dostępu dla użytkownika do najczęściej używanych funkcji. Zadaniami elementów paska narzędzi aplikacji RESpecTa, pokazanego na rysunku 21 są (od lewej do prawej):

1. Usunięcie zaznaczonych elementów
2. Wysunięcie zaznaczonych elementów przed pozostałe
3. Schowanie zaznaczonych elementów za pozostałe
4. Rozpoczęcie dodania stanu (zakończone przez kliknięcie na scenie, co definiuje miejsce w którym stan powstanie)
5. Rozpoczęcie dodanie tranzycji (zakończone przez przeciągnięcie od stanu początkowego do końcowego)
6. Otwarcie okna edycji zadań
7. Zmiana skali sceny.

Rysunek 21: Pasek narzędzi

5.5.3 Terminal

Kolejnym elementem aplikacji jest terminal, pokazany na rysunku 22. Służy on do przekazywania użytkownikowi komunikatów. Wydzieliłem w nim dwie części:

- Terminal (na białym tle), w którym są na bieżąco wyświetlane wiadomości (kolor zielony czcionki), ostrzeżenia (kolor żółty) i błędy (kolor czerwony)
- Linię podpowiedzi (czarna czcionka na szarym tle), która sugeruje co użytkownik powinien w danym momencie zrobić

Rysunek 22: Terminal

5.6 Poprawność edytowanego zadania

Jednym z najważniejszych powodów powstania tego edytora jest wyeliminowanie błędów, które mogłyby powstawać w wyniku specyfikacji plików XML przez użytkowników. W tym celu realizuję możliwie kompleksowy system sprawdzania poprawności zadań.

5.6.1 Sprawdzanie przy wprowadzaniu nowych danych

W przypadku wprowadzania nowych danych należy zapewnić unikatowość nazwy stanu i warunku tranzycji w ramach stanu startowego. Dodatkowo sprawdzane jest, czy stany zawierają wszystkie elementy obowiązkowe (np. czy zestaw robotów nie jest pusty).

5.6.2 Sprawdzanie podczas zapisu

Jest to sprawdzanie, które można wywołać również funkcją Validate. w ramach tej funkcji sprawdzane jest:

- istnienie stanu początkowego i końcowego w zadaniu głównym
- istnienie stanu końcowego dla wszystkich podzadań
- użycie wszystkich zainicjalizowanych sensorów i generatorów dla robotów (sygnalizowane jako ostrzeżenie)
- zainicjalizowanie wszystkich użytych sensorów i generatorów dla robotów

- sprawdzenie istnienia w zadaniach stanów zdefiniowanych jako początki wykonania podzadań
- sprawdzenie, czy dla każdego stanu (poza końcowymi) istnieje przynajmniej jedna tranzycja wyjściowa
- sprawdzenie, czy dla każdego stanu ostatnia tranzycja ma warunek true
- sprawdzenie, czy tranzycja z warunkiem true nie wskazuje na ten sam stan (powodowałoby to zapętlenie programu przy wykonywaniu zadania w systemie MRROC++)

5.6.3 Sprawdzanie podczas odczytu

Przy odczycie sprawdzane jest istnienie wszystkich obowiązkowych elementów dla każdego stanu podczas odczytu oraz istnienie dodatkowych elementów niezdefiniowanych dla danego nadelementu (niepożądane dzieci elementu). Po wczytaniu uruchamiana jest funkcja Validate w celu sprawdzenia błędów w całości projektu.

6 Realizacja aplikacji

W tym rozdziale przedstawiam w skrócie zmiany jakich dokonałem, oraz opisuję dane przechowywane w programie.

6.1 Zmiany w strukturze MRROC++

W celu rozpoczęcia prac nad edytorem do zadań robotycznych musiałem upewnić się, że zadanie stworzone przez Marka Kisiela jest w pełni funkcjonalne. W związku ze zmianami, które nastąpiły w strukturze MRROC++ musiałem dostosować do nich system.

6.1.1 Zmiany w zadaniu FSautomat

Od czasu stworzenia przez Marka Kisiela w systemie MRROC++ zaszło wiele zmian. Większość z nich dotyczyła nazw funkcji i stałych i była bardzo prosta do rozszerzenia na zadanie FSautomat.

W przypadku funkcji odpowiadającej za oczekiwani na zakończenie ruchu robotów zmieniona została również lista argumentów (między innymi zredukowane zostały 2 listy robotów do jednej). W związku z tym razem z mgr Piotrem Trojankiem stworzyliśmy analogiczną funkcję, która używała zestawu robotów zawartego w klasie reprezentującej stan w systemie MRROC++.

Największą zmianą w systemie MRROC++ (która zaszła od czasu uruchomienia zadania FSautomat do momentu rozpoczęcia przeze mnie prac) było zastąpienie generatora smooth odpowiadającego za odtwarzanie trajektorii przez roboty IRP-6 w laboratorium robotyki na generator newsmooth[7] i związana z tym zmiana klasy trajektorii im odpowiadającej. Wiązało się to z utworzeniem wraz z inż. Rafałem Tulwinem funkcji pozwalających na wczytywanie do generatora gotowych trajektorii.

W celu umożliwienia w różnych momentach działania programu wczytywania trajektorii z plików zapisanych w różnych współrzędnych zmieniłem funkcję wczytującą z pliku tak, aby najpierw sprawdzała ile współrzędnych posiada wektor przyspieszeń, a następnie dostosowywała wczytywanie danych do tej liczby.

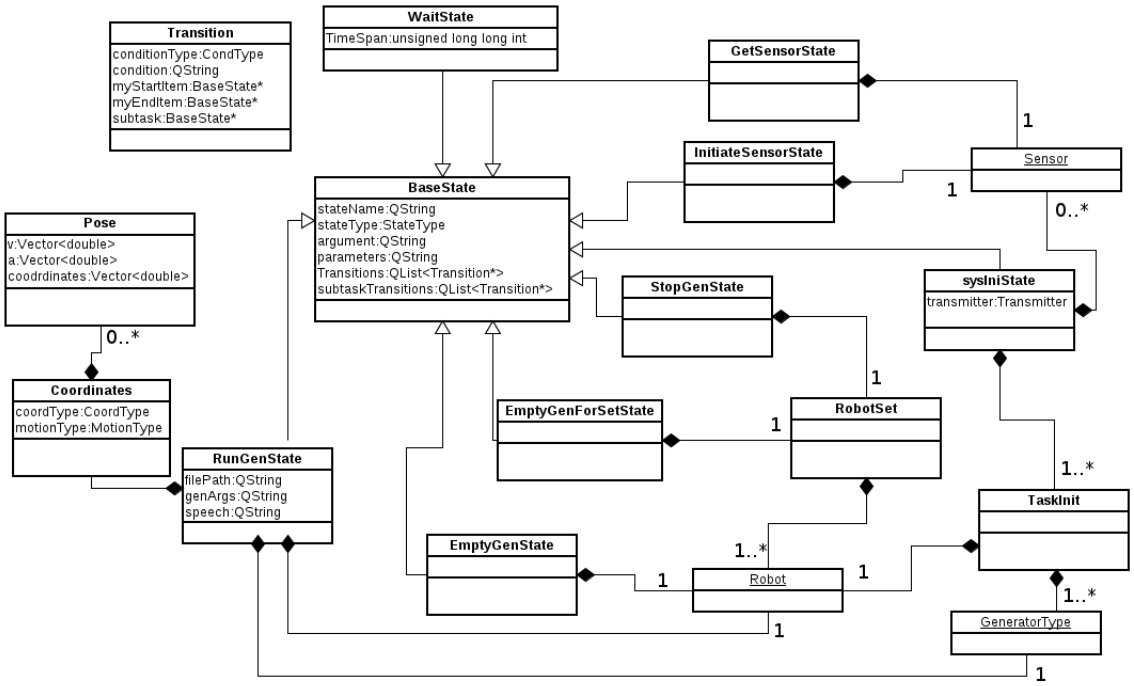
6.1.2 Zmiany w definicji języka XML

W związku ze zmianami w systemie MRROC++, a także w związku z usprawnianiem działania zadania FSautomat i dostosowywaniem go do graficznej reprezentacji wprowadziłem następujące zmiany w języku XML definiującym zadania robotów:

- w elemencie Trajectory usunąłem atrybut NumOfPoses, określający liczbę pozycji w trajektorii, który był redundantny. Obecnie liczba ta jest liczona podczas wczytywania elementów typu Pose.
- w elemencie Trajectory wprowadziłem atrybut motionType, który został szerzej opisany w podrozdziale 3.11.1

- Element Pose nie zawiera już elementów startVelocity ani endVelocity, ponieważ generator newsmooth nie dopuszcza innych prędkości krańcowych niż 0.
- Do opisu elementu State dodałem elementy PosX (3.19) oraz PosY (3.20) określające pozycję graficzną elementu na scenie w edytorze graficznym.
- Do elementów TaskDescription oraz Subtask został dodany element Graphics (3.18) opisujący położenie i skalę widoku sceny w edytorze graficznym.
- Konieczność występowania elementu transition w ramach elementu State została zniesiona, związane jest to z zapisywaniem końcowych stanów dla każdego zadania.
- Element SetOfRobots zawiera obecnie tylko jeden podelement: FirstSet. Jest to spowodowane zmianami w funkcjach wykorzystujących zestawy robotów - obecnie żadna funkcja nie wykorzystuje dwóch zestawów, więc okazało się to zbędne.

6.2 Schemat klas danych



Rysunek 23: Diagram klas danych dla edytora

Na diagramie 23 przedstawione zostały klasy oraz obiekty (typy wyliczeniowe), które reprezentują dane potrzebne do zapisu zadania robotycznego. Hierarchia klas stanów jest bardzo prosta: każdy typ stanu ma własną podklasę; wszystkie dziedziczą po klasie bazowej. Klasa tranzycji zawiera 3 wskaźniki na stany, zaś Stan zawiera 2 listy tranzycji; spowodowane jest to mechanizmem podzadań stworzonym w zadaniu FSautomat. Wszystkie przedstawione powyżej klasy posiadają funkcje pozwalające na zapisywanie i odczytywanie ich do/z plików XML przy użyciu klas Qt (QXmlStreamReader oraz QXmlStreamWriter).

6.3 Przedstawienie elementów XML w C++

W celu ułatwienia działania na danych, które nie są prostymi znacznikami XML, ale zawierają inne znaczniki, postanowiłem reprezentować je w postaci klas. W tym podrozdziale przedstawiam sposób zapisu poszczególnych znaczników XML w aplikacji. Elementy główne TaskDescription oraz SubTask nie są przedstawione, ich zawartość jest zapisywana do klasy reprezentującej graf. Pozostałe elementy zawierają elementy potrzebne do ich opisu według definicji zawartych w rozdziale 3.

- Klasa reprezentująca element State dziedziczy po elemencie QGraphicsItem, co zagwarantuje jej graficzną reprezentację. Stany o różnych typach (a więc zawierające różne podelementy) będą reprezentowane przez inne podklasy klasy State. Elementy wspólne dla wszystkich typów stanów będą zawarte w klasie bazowej. W celu zapewnienia widoczności stanów i tranzycji końcowych musi powstać klasa reprezentująca stan końcowy, nie posiadająca żadnego z typów.

```
class BaseState : public QGraphicsPolygonItem
{
...
    /**
    *   Name of the state.
    */
    QString stateName;

    /**
    *   Type of the State.
    */
    StateType stateType;

    /**
    *   Argument(optional) of the state.
    */
    QString argument;

    /**
    *   Parameters(optional) of the state.
    */
    QString parameters;

    /**
    *   List of transitions of this state.
    */
    QList<Transition *> Transitions;

    /**
    *   List of transitions, which point to this item as to a subtask.
    */
    QList<Transition *> subtaskTransitions;
```

```
};
```

Wydruk 49: Klasa bazowa reprezentująca stan - BaseState

Jak widać na powyższym wydruku klasa bazowa Stanu będzie zawierać wskaźniki na wszystkie tranzycje zaczynające się i/lub kończące się w tym stanie (również na Tranzycje, które wskazują na ten stan jako na początek podzadania). Typ stanu jest reprezentowany jako typ wyliczeniowy, a argument, parametr i nazwa będą ciągami znakowymi.

Poniżej przedstawiam projekty klas stanów konkretnych:

- **sysInitState** - stan o typie systemInitialization, dla którego stateType równa się wartości wyliczeniowej SYSTEM_INITIALIZATION.

```
class sysInitState:public BaseState
{
...
    /**
    *   Vector of robotInits, representing initializations of
    * generators for robots (1 robot only in each element).
    */
    std::vector<robotInit> inits;
    /**
    *   Object representing transmitter to be initialized.
    */
    Transmitter transmitter;
    /**
    *   Vector of Sensors to be initialized.
    */
    std::vector<Sensor> sensors;
};
```

Wydruk 50: Klasa reprezentująca stan systemInitialization - sysInitState

Jak wynika z powyższego kodu poza typami wyliczeniowymi - Sensor i Transmitter (6.3) użyty jest tu również typ robotInit, opisany w 6.3

- **RunGenState** - stan o typie set_next_ecp_state, dla którego stateType równa się wartości wyliczeniowej RUN_GENERATOR.

```
class RunGenState:public BaseState
{
...
    /**
    *   Path to the trajectory file.
    */
    QString filePath;
    /**
    *   Robot which executes the generator.
    */
    Robot robot;
```

```
    /**
    *   Generator, which is executed.
    */
    GeneratorType genType;
    /**
    *   Coordinates holding poses and pose specification.
    */
    Coordinates * coords;
    /**
    *   Additional argument for generator execution.
    */sysInitState

    QString genArgs;
    /**
    *   Speech which will be used by a speech generator.
    */
    QString speech;
};
```

Wydruk 51: Klasa reprezentująca stan set_next_ecp_state - RunGenState

Ta klasa, poza elementami prostymi (typy wyliczeniowe i ciągi znakowe), zawiera wskaźnik na element klasy Coordinates reprezentującej zapis trajektorii ruchu dla robota, opisany szerzej w 6.3.

– **EmptyGenForSetState** - stan o typie wait_for_task_termination, dla którego stateType równa się wartości wyliczeniowej EMPTY_GEN_FOR_SET.

```
class EmptyGenForSetState:public BaseState
{
...
    /**
    *   Set of robots.
    */
    RobotSet set;
};
```

Wydruk 52: Klasa reprezentująca stan wait_for_task_termination - EmptyGenForSetState

Klasa ta zawiera jedynie element reprezentujący zestaw robotów - RobotSet, opisany szerzej w 6.3

– **EmptyGenState** - stan o typie emptyGen, dla którego stateType równa się wartości wyliczeniowej EMPTY_GEN.

```
class EmptyGenState:public BaseState
{
...
    /**
    *   Robot, for which the empty generator is called.
    */
```

```
    Robot robot;
};
```

Wydruk 53: Klasa reprezentująca stan emptyGen - EmptyGenState

Ta klasa zawiera jedynie element wyliczeniowy reprezentujący robota.

– **WaitState** - stan o typie wait_ms, dla którego stateType równa się wartości wyliczeniowej WAIT.

```
class WaitState:public BaseState
{
...
    /**
    *   Time, for which the system will stop (in ms).
    */
    unsigned long long int Timespan;
};
```

Wydruk 54: Klasa reprezentująca stan wait_ms - WaitState

Dla tego typu stanu potrzebny jest jedynie element reprezentujący czas oczekiwania, Aby zapewnić największą możliwą wartość użyłem typu unsigned long long int.

– **StopGenState** - stan o typie send_end_motion_to_ecps, dla którego stateType równa się wartości wyliczeniowej STOP_GEN.

```
class StopGenState:public BaseState
{
...
    /**
    *   Set of robots.
    */
    RobotSet set;
};
```

Wydruk 55: Klasa reprezentująca stan send_end_motion_to_ecps - StopGenState

Ten typ stanu zawiera tylko zestaw robotów do zatrzymania, reprezentowany przez klasę RobotSet.

– **InitiateSensorState** - stan o typie initiateSensorReading, dla którego stateType równa się wartości wyliczeniowej INITIATE_SENSOR_READING.

```
class InitiateSensorState:public BaseState
{
...
    /**
    *   Sensor to be initialized.
    */
    Sensor sensor;
```

```
};
```

Wydruk 56: Klasa reprezentująca stan initiateSensorReading - InitiateSensorState

Jedyną daną przechowywaną przez stan tego typu jest sensor reprezentowany przez typ wyliczeniowy.

– **GetSensorState** - stan o typie getSensorReading, dla którego stateType równa się wartości wyliczeniowej GET_SENSOR_READING.

```
class GetSensorState:public BaseState
{
...
    /**
    *   Sensor, from which the information which be received.
    */
    Sensor sensor;
};
```

Wydruk 57: Klasa reprezentująca stan getSensorReading - GetSensorState

Jedyną daną przechowywaną przez stan tego typu jest sensor reprezentowany przez typ wyliczeniowy.

- Klasa reprezentująca element transition również dziedziczy po elemencie QGraphicsItem.

```
class Transition : public QGraphicsLineItem
{
...
    /**
    * Type of the condition.
    */
    ConditionType CondType;

    /**
    *   Start item of the transition.
    */
    BaseState *myStartItem;

    /**
    *   End item of the transition.
    */
    BaseState *myEndItem;

    /**
    *   String representing the condition of this transition
    *   if the CondType is "iniFile.".
    */
    QString condition;
```

```
25      /**
        *   Pointer to the subtask starting point of the transition.
        */
        BaseState * subtask;
};
```

Wydruk 58: Klasa reprezentująca tranzycję - Transition

Jak widać z powyższego wydruku tranzycja będzie posiadać wskaźniki na stany - początkowy i końcowy, a także na stan wskazywany jako podzadanie. Warunek przejścia będzie reprezentowany przez wartość wyliczeniową oraz ciąg znakowy, w przypadku gdy warunek będzie odnosił się do wartości w pliku INI.

- Element ecp będzie reprezentowany przez klasę robotInit.

```
1  class robotInit
   {
   ...
       /**
5      *   Robot, which is being initialized
       */
       Robot robot;
       /**
       *   Vector of generators, and their init arguments.
10     */
       std::vector < std::pair<GeneratorType, int> > init_values;
   };
```

Wydruk 59: Klasa reprezentująca inicjację robota - robotInit

Jak widać na powyższym wydruku klasa ta będzie zawierać nazwę robota przedstawioną przez typ wyliczeniowy, oraz listę generatorów wraz z ich wartościami inicjalizującymi.

- Klasa reprezentująca SetOfRobots będzie zawierać listę robotów.

```
1  class RobotSet
   {
   ...
       /**
5      *   First set of Robots.
       */
       std::vector<Robot> first;
   };
```

Wydruk 60: Klasa RobotSet reprezentująca element SetOfRobots

Jak widać z powyższego wydruku do zdefiniowania listy użyty został kontener vector, zawierający elementy wyliczeniowe typu Robot.

- Klasa reprezentująca element Trajectory.

```
class Coordinates
{
...
    /**
    *   CoordinateType of the coordinates.
    */
    CoordType coordType;
    /**
    *   MotionType of the coordinates.
    */
    MotionType motionType;
    /**
    *   Vector of Poses of the coordinates.
    */
    std::vector<Pose *> poses;
};
```

Wydruk 61: Klasa Coordinates reprezentująca element Trajectory

Z powyższego wydruku wynika, że klasa ta będzie zawierała wartości wyliczeniowe reprezentujące typ ruchu i typ współrzędnych, oraz listę pozycji.

- Klasa reprezentujące element Pose przedstawiona jest na poniższym wydruku.

```
class Pose
{
...
    /**
    *   Vector representing Accelerations.
    */
    std::vector<double> a;
    /**
    *   Vector representing Velocities.
    */
    std::vector<double> v;
    /**
    *   Vector representing Coordinates.
    */
    std::vector<double> coordinates;
};
```

Wydruk 62: Klasa Pose

Wynika z niego, że klasa Pose będzie zawierać 3 listy liczb zmiennoprzecinkowych, reprezentujące przyspieszenia, prędkości i koordynaty.

- Roboty, Generatory, Transmittery oraz Sensory będą reprezentowane przez wartości wyliczeniowe.

```
/**
```

```
*   Enum Type representing Robots.
*/
enum Robot {irp6ot_m, irp6p_m, festival, conveyor,
              bird_hand,
           ROBOTS_NUMBER};

/**
*   Enum Type representing GeneratorTypes.
*/
enum GeneratorType {ECP_GEN_TEACH_IN, ECP_GEN_NEWSMOOTH,
ECP_GEN_WEIGHT_MEASURE, ECP_GEN_TRANSPARENT, ECP_GEN_TFF_NOSE_RUN,
ECP_ST_BIAS_EDP_FORCE, ECP_GEN_TFF_RUBIK_GRAB,
ECP_GEN_TFF_RUBIK_FACE_ROTATE, ECP_ST_GRIPPER_OPENING,
ECP_GEN_TFF_GRIPPER_APPROACH, ECP_GEN_FESTIVAL,
                    GENERATORS_NUMBER};

/**
*   Enum Type representing Sensors.
*/
enum Sensor {SENSOR_CAMERA_ON_TRACK, SENSOR_CAMERA_SA,
         SENSORS_NUMBER};

/**
*   Enum Type representing Transmitters.
*/
enum Transmitter {TRANSMITTER_RC_WINDOWS,
              TRANSMITTERS_NUMBER};
```

Wydruk 63: Elementy wyliczeniowe w aplikacji

- Pozostałe elementy języka XML reprezentującego zadania robotyczne będą reprezentowane przez ciągi znakowe. Listy dostępnych wartości przedstawione są powyżej. Dla każdej listy dodany jest element oznaczający liczbę elementów danego typu. Dla każdego typu wyliczeniowego powstanie tablica przechowująca ciągi znakowe dla konkretnych elementów danego typu.

7 Podsumowanie

W tym rozdziale podsumowuję wykonaną przeze mnie pracę, opiszę przeprowadzone testy i ich wyniki.

7.1 Przeprowadzone testy

Testowanie edytora, który powstał jako wynik tej pracy można podzielić na dwie części:

1. Testowanie wymagań funkcjonalnych, które zostało przeprowadzone poprzez tworzenie zadań robotycznych. Zadania te były zapisywane i wczytywane z plików XML. Dodatkowo, aby sprawdzić poprawność plików były one wczytywane przez zadanie FSautomat i odtwarzane na symulatorze i fizycznych robotach w laboratorium robotyki IAiIS. Ostatecznym testem poprawności specyfikowanych zadań było stworzenie pliku XML reprezentującego zadanie swarm_demo. Sprawdzanie wykrywania błędów przez edytor było testowane poprzez wprowadzanie błędnych danych i obserwowanie, czy zostają one wychwycone przez system. W tym celu zostały stworzone pliki reprezentujące wszystkie błędy specyfikacji całości zadań oraz zostały przeprowadzone testy dodawania błędnych elementów.

2. Intuicyjność interfejsu sprawdzana była przez testy użytkowników, którzy przedstawiali swoje uwagi do interfejsu. Uwagi te były uwzględniane w aplikacji.

7.2 Wnioski

Aplikacja, którą stworzyłem spełnia wszystkie wymagania, które zostały przedstawione w rozdziale 4. Będzie ona pomocna w dydaktyce, ułatwi nowym użytkownikom rozpoczęcie prac z systemem MRROC++. Platforma Qt okazała się odpowiednia do tego zadania: pozwala na budowę aplikacji wraz z rozmaitymi ich funkcjami, zawiera moduł odpowiedzialny za zapisywanie i odczytywanie plików XML oraz posiada bardzo dobrą dokumentację.

7.3 Perspektywy rozwoju

Rozwój aplikacji RESpecTa jest możliwy pod warunkiem rozwoju zadania FSautomat. Zważywszy na pewne ograniczenia tego zadania (np. brak możliwości uzależnienia działania systemu od kolejności zakończenia zadań przez różne roboty) jest wskazane. W związku z tym jest bardzo prawdopodobne, aby ta aplikacja była rozwijana, gdyż umożliwi to nowym użytkownikom tworzenie bardziej rozwiniętych zadań robotycznych bez konieczności zagłębiania się w sposób działania całej struktury MRROC++.

8 Załączniki

1. Załącznik A: plik instrukcji użycia RESpecTa_HowTo.pdf
2. Załącznik B: plik fsautomat.dtd

Literatura

[1] Marek Kisiel. Język opisu i realizacja zadań przez automat skończony w systemie MRROC++. Praca dyplomowa inżynierska, Politechnika Warszawska, Warszawa, Listopad 2008.

[2] Automat Moore'a [online]. wikipedia : wolna encyklopedia. [dostęp: 2011-08-23 15:12Z]. Dostępny w Internecie: `http://pl.wikipedia.org/wiki/Automat_Moore'a`.

[3] Cezary Zieliński, Wojciech Szynkiewicz, Tomasz Winiarski, and Tomasz Kornuta. Mrroc++ based system description. Technical report, IAiIS, Warszawa, Czerwiec 2007.

[4] Qt reference documentation [online]. [dostęp: 2011-08-23 15:43Z]. Dostępny w Internecie: `http://doc.qt.nokia.com/4.7/index.html`.

[5] Steve Burbeck. Applications programming in smalltalk-80(tm):how to use model-view-controller (mvc)[online].

[6] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library User Guide and Reference Manual*. Pearson Education, Inc., 2002.

[7] Rafał Tulwin. Trajectory generation in MRROC++ applications. Praca dyplomowa inżynierska, Politechnika Warszawska, Warszawa, 2010.