POLITECHNIKA WARSZAWSKA

WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH

INSTYTUT AUTOMATYKI I INFORMATYKI STOSOWANEJ

Rok akademicki 2009/2010

PRACA DYPLOMOWA INŻYNIERSKA

Joanna Stocka

Graficzny edytor automatu specyfikującego zadanie robotyczne zapisane w języku XML.

Opiekun pracy:
dr inż. Tomasz Winiarski

Ocena pracy: .....

.....

Data i podpis Przewodniczącego

Komisji Egzaminu Dyplomowego

Chciałabym podziękować mojemu promotorowi, Tomkowi Winiarskiemu, za pomoc merytoryczną otrzymaną w trakcie realizacji niniejszej pracy, za wszystkie rady i celne uwagi, które miały wpływ na kształt tej pracy.

Chciałabym również ogromnie podziękować moim rodzicom, bez których powstanie tej pracy byłoby niemożliwe.

Streszczenie

Tytuł: Graficzny edytor automatu specyfikującego zadanie robotyczne zapisane w języku XML.

Celem niniejszej pracy inżynierskiej było zaprojektowania oraz implementacja graficznego edytora umożliwiającego definiowanie zadań robotycznych, które mogłyby być wykonane w środowisku MRROC++. Ze względu na możliwość działania aplikacji na wielu platformach, edytor został zrealizowany w języku C++, z użyciem platformy QT. Aplikacja powstała zgodnie z modelem gui-controler-model, gdyż zapewnia on całkowitą separację warstwy danych, aplikacji i prezentacji, a także pozwala na kontrolę poprawności danych już na etapie specyfikacji zadania. Zrealizowany edytor spełnia wszystkie stawiane przed nim wymagania, oferuje także kilka dodatkowych funkcji, które powstały z myślą o przyszłych użytkownikach. Poprawność powstałego narzędzia była testowana w warunkach laboratoryjnych, zarówno na dostępnych symulatorach jak i rzeczywistym sprzęcie.

Abstract

Title: Multi-robot task graphical editor for creation based on XML notation.

The purpose of this thesis was to design and implement graphical editor that allows to create the definition of multi-robot task based on XML notation that can be executed in the MRROC++.

User interface application was implemented in C++ language, in Qt framework. The application was in accordance with the Gui-Controler-model model, because it provides complete separation of the layers of data, application and presentation. In addition, it allows to control of the data correctness at the stage of the task specification.

Spis treści

1 Wstęp **5**

1.1 Geneza pracy 5

1.2 Cel pracy 5

1.3 Układ pracy 6

2 Opis wybranych technologii **7**

2.1 System MRROC++ 7

2.1.1 Struktura ramowa 7

2.1.2 Generatory w systemie MRROC++ 10

2.2 Definicja automatu skończonego 10

2.2.1 Automat Moore'a 11

2.3 XML 12

2.4 SGML 13

2.5 DTD 14

2.6 Platforma QT 15

2.6.1 Sygnały i sloty 16

2.6.2 Graphics View Framework 16

2.6.3 QDomDocument 17

3 Definicja języka opisu zadań robotycznych **18**

3.1 Podstawowe pojęcia 18

3.2 DTD definicji języka 18

3.3 Element <TaskDesctiption> 19

3.4 Element <State> 19

3.4.1 Opis atrybutów 20

3.4.2 Dzieci elementu <State> 25

3.5 Element <SubTask> 26

3.6 Element <transition> 27

3.6.1 Atrybut target 29

3.7 Element <ROBOT> 29

3.8 Element <SetOfRobots> 30

3.9 Element <ECPGeneratorType> 31

3.10 Element <Trajectory> 32

3.11 Element <GeneratorFilePath> 33

3.12 Element <taskInit> 33

3.12.1 Element <ecp> 34

3.12.2 Element <mp> 36

3.13 Element <Speech> 36

3.14 Element <AddArg> . . . 36
3.15 Element <Sensor> . . . 37
3.16 Element <TimeSpan> . . . 38
3.17 Element <GeneratorParameters> . . . 38
3.18 Element <Parameters> . . . 38

4 Koncepcja aplikacji **39**
4.1 Schemat rozwiązania . . . 39
4.2 Wymagania wobec powstającej aplikacji . . . 40

5 Realizacja rozwiązania **42**
5.1 Modułowość . . . 42
5.1.1 Schemat aplikacji . . . 42
5.2 Wizualizacja sceny 2D . . . 44
5.2.1 Scena edytora . . . 46
5.2.2 Reprezentacja stanu . . . 46
5.2.3 Reprezentacja tranzycji . . . 46
5.2.4 Reprezentacja podzadania . . . 47
5.2.5 Zarządzanie listą stanu automatu . . . 47
5.2.6 Implementacja . . . 47
5.2.7 Wynik pracy . . . 50
5.3 Poprawność semantyczna zadania . . . 51
5.4 Poprawność semantyczna stanu . . . 52
5.4.1 Zapewnienie zgodności z DTD . . . 52
5.4.2 Implementacja . . . 53

6 Podsumowanie **57**
6.1 Przeprowadzone testy . . . 57
6.2 Wnioski . . . 58
6.3 Perspektywy rozwoju . . . 59

1 Wstęp

1.1 Geneza pracy

Efektywnym narzędziem wspierającym tworzenie sterowników dla systemów robotycznych są programowe struktury ramowe. Zawierają one wzorce programowe, a także udostępniają narzędzia przydatne przy realizacji programów sterujących. Jako przykład tak opisanej struktury programowej może posłużyć MRROC++. Powstał on w Instytucie Automatyki i Informatyki Stosowanej Politechniki Warszawskiej, jako narzędzie wspomagające tworzenie sterowników dla współbieżnych systemów wielorobotowych. Został napisany w języku C++, działa natomiast pod kontrolą systemu operacyjnego czasu rzeczywistego QNX Neutrino.

System MRROC++ pozwala na budowanie nowych sterowników dla systemów wielorobotowych poprzez wykorzystaniu już istniejących rozwiązań i modułów. Naturalnym więc wydawała się potrzeba stworzenia sformalizowanego języka zadań robotów, co w znaczącym stopniu przyczyniło się do ułatwienia definiowania tych zadań. Postanowiono wykorzystać notację zgodną z logiką maszyny stanowej. Bardzo wygodnym i popularnym standardem przeznaczonym do prezentacji danych w ustruktualizowany sposób jest język XML, dlatego też został on wykorzystany do zapisu definicji zadania zgodnie z regułami maszyny stanowej. Dzięki temu, system MRROC++ umożliwia tworzenie zadań w oparciu o przyjętą konwencję. Możliwe jest zapisanie zadania w formacie pliku XML, którego struktura jest określona przez format określony w pliku DTD.

Po latach użytkowania okazało się, że dotychczasowy mechanizm jest jednak niewystarczający. Definiowanie zadań ręcznie, w plikach tekstowych, jest dość czasochłonne, a także pociąga za sobą możliwość potencjalnych błędów. Z tego też względu powstał pomysł stworzenia graficznego edytora, który będzie umożliwiał definiowanie zadania w oparciu o istniejący język opisu zadań.

1.2 Cel pracy

Celem niniejszej pracy inżynierskiej jest stworzenie graficznego edytora, umożliwiającego tworzenie zadań wieloobrotowych zapisanych w języku XML. Aplikacja powstaje w oparciu o reprezentację automatu skończonego, zdefiniowanego w ramach pracy inżynierskiej Marka Kisiela, która pozwala na interpretację zadania i jego wykonanie w systemie MRROC++. Celem pracy jest implementacja narzędzia, które w znacznym stopniu ułatwi definiowanie zadania. Umożliwi również uniknięcia potencjalnych błędów, które mogłyby powstać w trakcie definiowania zadania, ponieważ program wyposażony będzie w system detekcji błędów, m.in. sprawdzana będzie poprawność dokumentu XML ze strukturą DTD opisującą poprawną strukturę automatu.

Od projektanta zadania wymagane będzie określenie akcji związanej ze stanami, tranzycjami do innych stanów, czy warunków samych tranzycji. Efektem działania takiej aplikacji ma być kod dokumentu XML z opisem zadania. Edytor ma również umożliwiać edycję już stworzonych zadań, tak aby na podstawie kodu XML zawierającym zadanie robotyczne możliwe było odtworzenie semantycznej logiki automatu, jak również jego wizualizacja w edytorze.

Dodatkowym atutem będzie umożliwienie osobom nie znającym języka XML oraz C++, w którym napisany jest kod źródłowy systemu MROCC++, samodzielnego tworzenia zadań. Może to zaowocować wzrostem liczby osób zainteresowanych rozwojem systemu MRROC++. Zadanie użytkowe wykonywane jest w oparciu o jego definicję opisaną w dokumencie XML. Z racji tego, iż powstawanie wyżej wymienionego pliku jest niezależne od platformy, kluczowym celem był wybór technologii, która najlepiej zapewni wieloplatformowość. Ostatecznie wybrane zostało środowisko QT.

1.3 Układ pracy

Rozdział 2 ma na celu przybliżenie technologii oraz pojęć, które zostały wykorzystane w trakcie realizacji projektu.

W rozdziale 3 opisany został język opisu zadań wielorobotowych, na bazie którego oparta jest struktura edytora. Znajduje się tam szczegółowy opis wszystkich dostępnych w pierwotnej wersji języka elementów, jak również jego rozszerzenie powstałe w celu wizualizacji automatu. Umieszczone są w nim również podstawy teoretyczne, na podstawie których zbudowany został język opisu zadania. Jest to definicja automatu skończonego oraz reguły odnoszące się do sposobu realizacji tego typu mechanizmów. Rozdział 4 ma na celu przedstawienie ogólnego schematu działania powstałej aplikacji. Zawiera szczegółowy opis wszystkich stawianych wobec edytora wymagań oraz prezentuje koncepcję rozwiązania. Rozdział 5 zawiera szczegółowy opis zadań zrealizowanych w ramach niniejszejszej pracy inżynierskiej. Ma na celu przestawienie zastosowanych algorytmów oraz szczegółów implementacyjnych. W rozdziale 6 znajdują się opis przeprowadzonych testów. Zawiera on także podsumowanie i wnioski z przeprowadzonych prac.

2 Opis wybranych technologii

Stworzony przeze mnie graficzny edytor automatu realizującego zadanie robotyczne w systemie MRROC++ został zrealizowany jako aplikacja w języku C++. Do budowy interfejsu graficznego wykorzystana została platforma QT dedykowana dla języka C++. Wszystkie użyte w rozwiązaniu technologie postaram się przedstawić w tym rozdziale.

2.1 System `MRROC++`

System `MRROC++` jest platformą programistyczną zapewniającą biblioteki modułów oraz wzorce projektowe umożliwiające konstrułowanie sterowników dla systemów wielorobotowych. Zaimplementowany został w języku C++ i ma typowo obiektową strukturę. Umożliwia konstrułowanie nowych sterowników dedykowanych określonym zadaniom systemu wielorobotowego w oparciu o istniejące sterowniki i moduły. Istotną cechą struktury `MRROC++` jest jej otwartość, umożliwiająca dołączenie do systemu nowych efektorów i czujników.

Poprzednikami systemu `MRROC++` były środowiska:

- `RORC` - system wspomagający tworzenie sterowników dla pojedyńczych robotów,
- `MRROC` - system wspomagający tworzenie sterowników dla systemów wielorobotowych w sposób proceduralny.

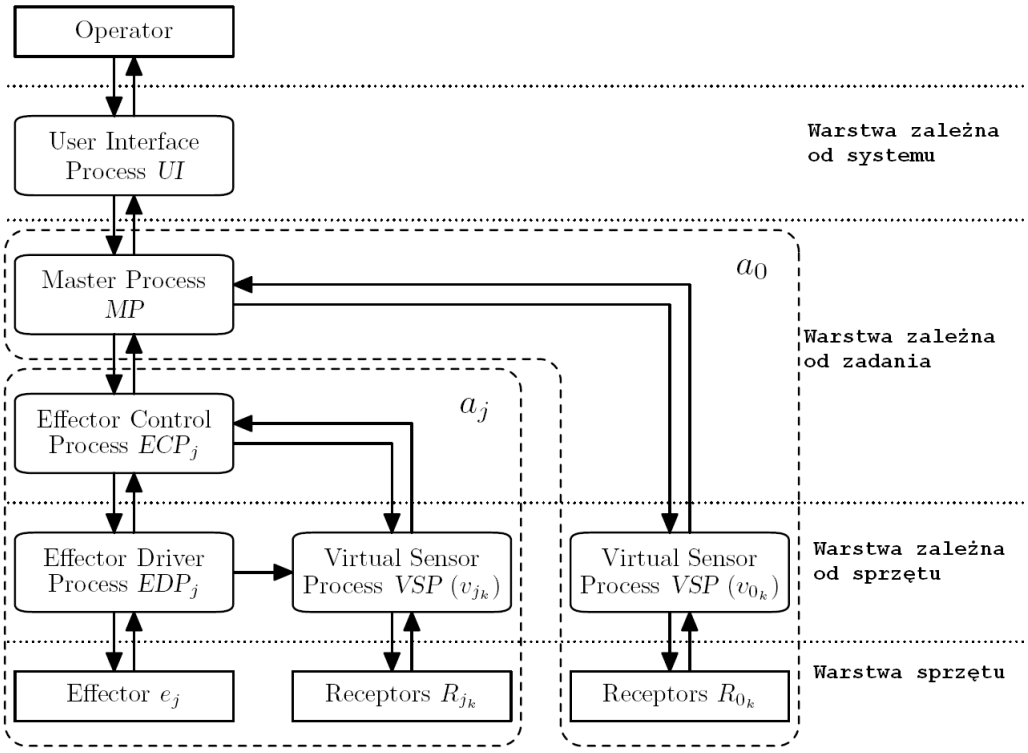
2.1.1 Struktura ramowa

W opisie systemu `MRROC++` używane będą następujące pojęcia:

- `efektor`, element systemu oddziałujący na otoczenie;
- `receptor (sensor)`, czujnik rzeczywisty, dostarczający informacji o stanie otoczenia;
- `system sterowania`, składający się ze sprzętu obliczeniowego i działającego na nim oprogramowania; jest odpowiedzialny za zarządzanie efektorami i receptorami.

Dzięki hierarchicznej strukturze systemu MRROC++ możliwe jest tworzenie sterowników składających się z modułów będących odrębnymi procesami. Procesy uruchamiane są w węzłach sieci, która działa pod kontrolą systemu operacyjnego czasu rzeczywistego QNX. Dzięki modularnej strukturze systemu uproszczona jest jego modyfikacja. Wymiana lub modyfikacja jednego z istniejących elementów nie wiąże się z koniecznocią wymiany czy modyfikacji pozostałych modułów. Struktura systemu MRROC++ została przedstawiona na rynuku 1.

W skład systemu MRROC++ wchodzą następujące elementy:

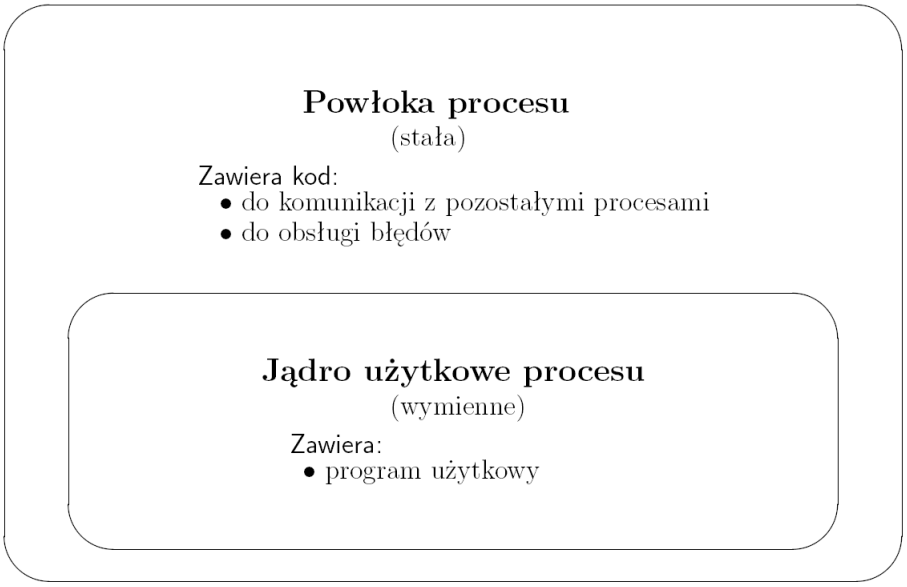


Rysunek 1: Struktura systemu MRROC++

- UI - interfejs użytkownika,
- MP (ang. Master Prosess) to virtualny proces kontrolujący koordynację procesów sterujących robotami.
- ECP (ang. Efector Control Process) to proces sterujący poszczególnymi efektorami, w zależności od zadania które jest aktualnie wykonywane przez system.
- EDP (ang. Efector Driver Process) to proces związany z konkretnym procesem ECP. Stanowi sterownik, który jest charakterystyczny dla danego typu efektora.
- VSP (ang. Virtual Sensor Processes) to proces czujnika virtualnego, który odpowiedzialny jest za odczyt i przetwarzanie danych z czujników rzeczywistych.

Wszystkie procesy MP, ECPoraz VSP mają wspólną strukturę zaprezentowaną na rysunku 2. Każdy z nich można podzielić na dwie podstawowe części:

- powłoka będąca niezmienną częścią procesu, niezależną od wykonywanego zadania. Odpowiada za realizację procesu w systemie operacyjnym, komunikację z innymi procesami w sytemie oraz obsługę błędów.
- jądro będąca wymienną częścią procesu, zależną od zadania realizowanego przez system robotyczny. Realizowane funkcje są zależne od zadania.



Rysunek 2: Ogólna struktura procesuw w systemie MRROC++

Kluczowe cele procesu MP to:

- kordynacja prac wszystkich obecnych w systemie efektorów;
- nasłuchiwanie poleceń od operatora w trakcie wykonywania zadania;
- generacja trajektorii dla współpracujących ze sobą robotów;
- przekazywanie informacji o błędach, nieprawidłowościach i awariach do UI.

Kluczowe cele procesu ECP to:

- przesyłanie informacji do MP i EDP w celu realizacji zadania,
- uruchamianie zadania pojedyńczego efektora,
- odbieranie informacji od procesów MP i wysyłanie odpowiedzi,
- generacja trajektorii ruchu pojedyńczego robota lub grupy współpracujących robotów,
- przekazywanie informacji o błędach, nieprawidłowościach i awariach do UI.

2.1.2 Generatory w systemie MRROC++

Najważniejszą, a zarazem najczęściej używana w ECP i MP instrukcją jest komenda Move, której kluczowym zadaniem jest wywoływanie instrukcji ruchowej klasy generatora. Jako źródło informacji wykorzystuje dane udostępnione poprzez dostępne sensory i efektory. Instrukcja Move kończy się wraz ze spełnieniem warunku terminalnego.

2.2 Definicja automatu skończonego

Automatem skończonym FSA (ang. Finite State Automaton) nazywamy model matematyczny systemu o dyskretnych wejściach i wyjściach, składający się z określonej liczby stanów, przejść między nimi, a także akcji związanych z tymi stanami. Istotne jest, iż w danym momencie system może znajdować się w dokładnie jednym ze skończonej liczby stanów.

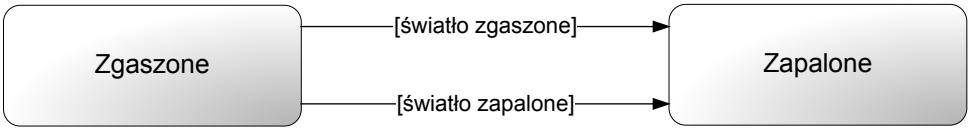
Automat umożliwia uzyskanie odpowiedzi wyjścia w zależności od zdarzenia wejściowego. Swoją pracę automat rozpoczyna w stanie początkowym, kończy natomiast w stanie terminalnym. Funkcje przejść między stanami opisane są przez warunki przejść, które muszą zostać spełnione, aby automat mógł przejść do kolejnego stanu.

Poprawnie semantycznie automat opisany jest przez krotkę $t = \{S, \sigma, A, T, G, s\}$, gdzie:

1. **S** - skończony zbiór stanów automatu,
2. σ - alfabet wejściowy, czyli skończony zbiór sygnałów wejściowych,
3. **A** - alfabet wyjściowy, czyli skończony zbiór sygnałów wyjściowych,
4. **T** - funkcja przejścia $T : Sx\sigma- > S$,
5. **G** - funkcja wyjścia $G : Sx\sigma- > A$,
6. **s** - stan początkowy.

W opisie maszyny stanowej wykorzystane będą następujące pojęcia:

- `Stan` - określa aktualny stan systemu (może być z nim związana akcja);
- `Tranzycja` - wskazuje na zmianę stanu. Jest opisana warunkiem, który musi być spełniony aby zmiana stanu była możliwa;
- `Akcja` - określa aktywność która zostanie wykonana w danej chwili;
- `Warunek przejścia` - określa warunek tranzycji (może przyjmować jedną z wartości określonej w alfabecie).



Rysunek 3: Przykład automatu skończonego

Automaty skończone bardzo często przedstawiane są w postaci grafów skierowanych. Dlatego też, dokładnie takie podejście zostanie zastosowane w niniejszej pracy. Przykład automatu został zaprezentowany na rysunku 3.

Możliwe jest również prezentacja automatu w postaci tabeli przejść i wyjść. Przykład takiego zapisu został przedstawiony w tabeli 1.

	Stan aktualny		
Warunek	Stan S1	Stan S2	Stan S3
Warunek A	Stan S1	Stan S1	Stan S2
Warunek B	Stan S2	Stan S2	Stan S3
Warunek C	Stan S2	Stan S1	Stan S3

Tablica 1: Przykładowa tabela przejść stanów

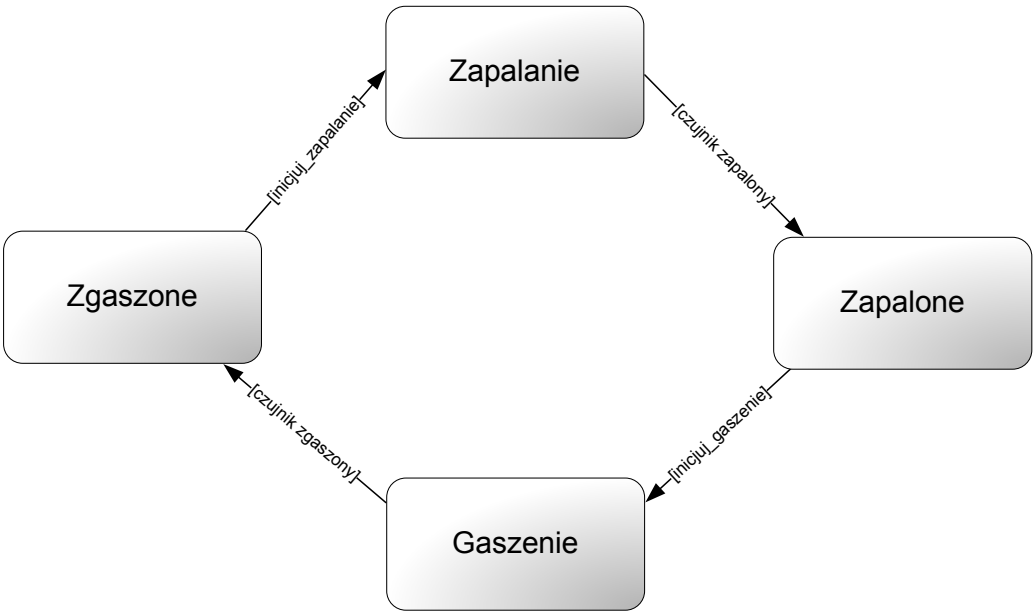
2.2.1 Automat Moore'a

Język opisu zadań dla robotów na którym bazuje niniejsza praca inżynierska powstał w oparciu o definicję maszyny stanowej Moorea'a. Umożliwia on nie tylko definiowanie prostych zadań, stanowiących sekwencję następujących po sobie akcji. Umożliwia również tworzenie bardziej złożonych, zawierające podzadania (każde z podzadań ma strukturę zgodną z definicją maszyny stanowej Moorea'a). Celem stworzenia podzadań było zapewnienie przejrzystości definiowanych zadań, jak również możliwość uniknięcia redundancji stanów automatu.

Każde zdefiniowane w edytorze zadanie (bądź też podzadanie) musi zawierać zarówno stan początkowy jak również stan terminalny. Jest to warunek konieczny, aby powstałe zadanie było zgodne z przyjętymi założeniami. Warunek ten jest sprawdzany już na etapie definiowania automatu, dzięki czemu możliwe jest uniknięcie ewentualnych błędów związanych ze stworzeniem automatu, który byłby niepoprawnie semantycznie.

Język opisu zadań robotycznych powstał na bazie maszyny stanowej Moore'a. Działanie takiego automatu, jest zgodne z konwencją zakładającą, że efektem zmiany warunków wejściowych, jest zmina stanu. Akcja związana jest jedynie ze stanami - wykonywana jest przy wejściu do stanu, nie ma notomiast żadnej akcji związanej z tranzycjami. Zastosowany do opisu automatu model zakłada, że wyjście ze stanu zależy jedynie od stanu w którym maszyna się aktualnie znajduje. Przejście do kolejnego stanu następuje

w sytuacji, gdy warunek tranzycji jest spełniony. Przykład automatu Moore'a został przedstawiony na rysunku 4.



Rysunek 4: Przykład automatu Moore'a

2.3 XML

XML (ang. Extensible Markup Language), czyli w wolnym tłumaczeniu Rozszerzalny Język Znaczników, jest językiem formalnym, przeznaczonym do przedstawiania różnych danych w uniwersalny, ustruaktualizowany sposób. Język XML jest niezależny od platformy, dzięki czemu możliwa jest wymiana dokumentów między różnymi (heterogenicznymi) systemami. Fakt ten miał istotny wpływ na popularyzację tego języka w dobie Internetu. XML jest podzbiorem języka SGML (ang. Standard Generalized Markup Language), eliminującym jego skomplikowany charakter wszędzie tam, gdzie było to możliwe. W związku z czym, każdy dokument XML jest także dokumentem SGML. Przykładowy dokument XML został zamieszczony na wydruku 1:

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <ksiazka-telefoniczna kategoria="bohaterowie">
    <osoba charakter="dobry">
      <imie>Joanna</imie>
5     <nazwisko>Stocka</nazwisko>
      <telefon>123-123-123</telefon>
    </osoba>
    <osoba charakter="zły">
      <imie>Jaś</imie>
```

```
10      <nazwisko>Fasola</nazwisko>
        <telefon>222-222-222</telefon>
    </osoba>
</ksiazka-telefoniczna>
```

Wydruk 1: Przykładowy dokument XML

Dokument rozpoczyna się instrukcją sterującą, zawierającą informacje o wersji języka XML, z którą jest zgodny, oraz o sposobie kodowania znaków. Wszystkie te informacje są opcjonalne, to znaczy można pominąć dowolne z nich, a nawet całą instrukcję sterującą. W przypadku braku któregokolwiek z atrybutów, zostaje mu nadana wartość domyślna, jaką jest 1.0 w przypadku wersji oraz UTF-8 w przypadku kodowania.

Na powyższym wydruku widać, że korzeniem tego dokumentu jest element o nazwie ksiazka-telefoniczna, który ma jeden atrybut o nazwie kategoria i wartości bohaterowie. Korzeń jest rodzicem dwóch elementów o nazwie osoba i przypisanym atrybucie o nazwie charakter. Każdy z nich, jest rodzicem trzech innych elementów o nazwach imie, nazwisko i telefon, których wartość zapisana jest w formie węzłów tekstowych (tekst pomiędzy odpowiednimi znacznikami otwierającym i zamykającym).

2.4 SGML

SGML (ang. Standard Generalized Markup Language) - w wolnym tłumaczeniu oznacza standardowy uogólniony język znaczników. Służy on do ujednolicania struktury i formatu różnego typu danych oraz pozwala zapisać je do pliku tekstowego, w związku z czym jest on całkowicie niezależny od systemu operacyjnego.

W odróżnieniu od języków znaczników dedykowanych do konkretnych zastosowań, jakim jest np. HTML, SGML nie jest jedynie zbiorem określonych znaczników oraz reguł ich użytkowania. Jest on ogólnym językiem umożliwiającym definiowanie dowolnych znaczników i ustalania zasad ich poprawnego użytkowania.

W praktyce języka SGML używa się do dwóch celów:

- precyzyjnego definiowania zbiorów znaczników przeznaczonych do konkretnych zastosowań (np. HTML),
- ujednolicenia formatu dokumentów tekstowych w obrębie dużych firm lub instytucji.

W obydwu przypadkach dokument SGML składa się z trzech części:

- właściwego dokumentu - tekst wraz ze znacznikami;
- DTD (ang. Document Type Definition) - definicji typu dokumentu, czyli definicji wszystkich znaczników oraz reguł ich stosowania (szerzej opisane w 2.4);

- deklaracji dokumentu definiującej ogólne reguły stosowane w zapisie dokumentu (np. maksymalna długość nazwy elementu, znak używany jako początek znacznika domyślnie jest to znak <).

W związku z dużą złożonością standardu SGML jedynie niewielka ilość narzędzi implementuje pełen standard. W związku z trudnościami implementacyjnymi powstał standard XML, który początkowo był jedynie podzbiorem reguł standardu SGML. Ważne jest, że standard XML można obsługiwać narzędziami dedykowanymi dla standardu SGML.

2.5 DTD

DTD (ang. Document Type Definition) w wolnym tłumaczeniu oznacza definicję typu dokumentu. Jest to rodzaj dokumentu definiujących formalną strukturę plików z rodziny SGML (np. XML).

Definicja DTD może być zawarta w pliku dokumentu, którego strukturę definiuje, bądz też zapisana w osobnym pliku tekstowym, czego niewątpliwą zaletą jest możliwość zastosowania tego samego DTD dla wielu dokumentów.

W pliku DTD określa się składnię konkretnego dokumentu XML lub SGML zdefiniowanego na potrzeby użytkownika. Zazwyczaj DTD definiuje wszystkie dopuszczalne elementy dokumentu, zbiory ich atrybutów oraz dopuszczalne wartości. DTD określa także obowiązkowość poszczególnych elementów w dokumencie. W praktyce DTD przeważnie składa się z definicji ELEMENT i definicji ATTLIST, gdzie:

- `ELEMENT` - oznacza definicję elementu,
- `ATTLIST` - oznacza definicję atrybutu elementu.

Standard DTD pozwala też w łatwy sposób określić kardynalność (liczebność) elementów zdefiniowanych w pliku definicji typu dokumentu. Szczegółowy opis dostępnych symboli został przedstawiony w tabeli 2.

Symbol	Znaczenie symbolu
brak symbolu	Element lub zbiór elementów występuje dokladnie jeden raz. Jest to wartość domyślna.
?	Element lub zbiór występuje co najwyżej jeden raz.
+	Element lub zbiór występuje co najmniej jeden raz.
*	Element lub zbiór występuje dowolną ilość razy.

Tablica 2: Kardynalność elementów (zbioru elementów) w pliku DTD

W oparciu o wyżej przedstawione standardy utworzony został język opisu zadań użytkowych dla robotów przemysłowych. Fragment plik DTD definiującego zadanie został przedstawiony na wydruku 2.

```
<!ELEMENT TaskDescription (State+,SubTask*)>
<!ELEMENT SubTask (State+)>

<!ELEMENT State ((ROBOT|SetOfRobots)?,ECPGeneratorType?,
  TrajectoryFilePath?,Trajectory?,taskInit?,Speech?,AddArg?,
  Sensor?,TimeSpan?,GeneratorParameters?,Parameters?,transition+)>
  <!ATTLIST State id CDATA #REQUIRED>
  <!ATTLIST State type CDATA #REQUIRED>

<!ELEMENT SetOfRobots (FirstSet?, SecSet?)>


<!ELEMENT FirstSet (ROBOT+)>
<!ELEMENT SecSet (ROBOT+)>
...
>
```

Wydruk 2: Zawartość pliku fsautomat.dtd

2.6 Platforma QT

Qt jest zestawem przenośnych bibliotek, a także narzędzi programistycznych dedykowanych dla różnych języków programowania. W chwili obecnej biblioteki Qt dostępne są w językach C++, Java, Ada, Pascal, Perl, PHP, Ruby i Python. Podstawowym składnikiem omawianej platformy są klasy służące do budowy interfejsów graficznych programów komputerowych. Jednak począwszy od wersji 4.0 Qt udostępnia także narzędzia do tworzenia programów konsolowych i serwerów. Dodatkowym atutem omawianego środowiska jest jego w pełni obiektowa architektura, zawierająca wiele technologii, które dotychczas były dostępne jedynie w Tk: mechanizm sygnałów i slotów, automatyczne rozmieszczanie widżetów oraz zhierarchizowany system obsługi zdarzeń. Należałoby tu nadmienić, że Qt posiada, niezależne od udostępnianej przez język C++ biblioteki STL, szablony klas kontenerowych. Udostępnia także niezależne od platformy systemowej moduły służące do obsługi m.in: baz danych (SQL), języka XML, wielowątkowości, grafiki trójwymiarowej (OpenGL), plików czy sieci. Platforma Qt dostępna jest na wielu systemach operacyjnych, m.in: Windows, Mac OS X oraz dla urządzeń wbudowanych opartych na Linuksie (Qt Extended), Windows CE i Symbian oraz platformy X11 (Solaris, GNU). Jest one podstawą środowiska graficznego KDE dedykowanego dla systemów unixowych, komunikatora Skype, czy przeglądarki internetowej Opera.

2.6.1 Sygnały i sloty

Mechanizmem zapewniającym komunikację pomiędzy różnymi obiektami wykorzystanym w platformie QT jest mechanizm sygnałów i slotów. Oparty jest na zasadzie bodziec - reakcja. Polega on na wzajemnym skojarzeniu sygnału, w wyniku którego zostanie wywołana określona akcja. Istotne jest, iż zarówno sygnał jak i związany z nim slot, odpowiedzialny za wykonanie określonych funkcji, muszą zawierać identyczną listę parametrów. W przeciwnym wypadku wzajemne połączenie będzie niemożliwe. Przykład wykorzystania został przedstawiony na wydruku 3.

```
1 Obiekt a;
  Obiekt b;
  connect(a, SIGNAL(mojSygnal(bool)), b, SLOT(mojSlot(bool)))
  emit mojSlot(true)
```

Wydruk 3: Przykład wykorzystania mechanizmu sygnałów i slotów

Zaprezentowany wydruk prezentuje wykorzystanie mechanizmów sygnałów i slotów. W linii 3 został przedstawiony przykład połączenia dwuch obiektów. Jeśli obiekt a wyemituje sygnał o nazwie mojSygnal(bool), na obiekcie b zostanie wywołany slot o nazwie mojSlot(bool), czego przykład został przedstawiony w linii 4.

2.6.2 Graphics View Framework

Interfejs graficzny został stworzony przy użyciu narzędzi dostępnych w środowisku QT - Graphics View Framework. Biblioteka ta umożliwia nie tylko wizualizację elementów graficznych, ale również ich obracanie, skalowanie i zapisywanie w preferowanych formacie (dostępne są m.in. formaty: pdf, svg, bmp).

Podstawowe klasy:

1. **Klasa QGraphicsItem** - jest to klasa bazowa dla wszystkich dostępnych elementów graficznych. Umożliwia definiowanie nie tylko prostych figur geometrycznych, takich jak: kwadraty, koła czy linie, ale również tworzenie własnych, niestandardowych elementów graficznych. Niewątpliwą zaletą tego podejścia jest możliwość definiowania złożonych elementów, składających się z wielu różnych figur. Obiekty malowane są w oparciu o lokalny układ współrzędnych w którym są umieszczone.

2. **Klasa QGraphicsScene** - jest to klasa która stanowi kontener do przechowywania instancji klasy QGraphicsItem. Opisuje ona lokalizację obiektów grafiki 2D na powierzchni sceny. Udostępnia także funkcje umożliwiające określenie pozycji obiektów na scenie, a także modyfikacje ich położenia. Opis położenia elementów oparty jest na kartezjańskim układzie współrzędnych, gdzie położenie reprezentowane jest przez współrzędne X i Y oraz rotację odpowiadającą za zmianę

lokalnego układu odniesienia obiektu wokół układu odniesienia całej sceny. Dodatkowym atutem opisywanej klasy jest indeksowanie elementów, zgodnie z algorytmem BSP - drzewo wyszukiwań binarnych (ang. Binary Space Partition). Dzięki temu, złożoność algorytmu wyszukania, a co za tym idzie dodawania lub usuwania elementów jest logarytmiczna. Zmniejsza to w dość znaczący sposób czas operacji wykonywanych na elementach sceny. Niezwykle istotne jest, iż klasa ta nie ma własnego wyglądu, służy ona jedynie do zarządzania elementami.

3. **Klasa QGraphicsView** - jest to klasa, której głównym celem jest wizualizacja sceny. Udostępnia ona interfejs umożliwiający zmianę układu współrzędnych, co ułatwia m.in. operacje takie jak skalowanie, czy obracanie elementów.

2.6.3 QDomDocument

QDomDocument jest klasą do obsługi dokumentów XML. Zasadniczo klasa QDomDocument reprezentuje cały dokument XML - jest to korzeń drzewa dokumentu, zapewnia więc dostęp do jego podstawowych danych, takich jak: elementy, atrybuty, węzły tekstowe, komentarze czy instrukcje przetwarzania.

Analizowany dokument XML jest reprezentowany przez drzewo, którego walidacja jest możliwa przy użyciu metod udostępnianych przez klasę QDOM. Wspomniane drzewo może być modyfikowane, poprzez szereg funkcji przewidzianych w tej klasie. Możliwe jest między innymi dodawanie nowych elementów, atrybutów, czy węzłów tekstowych, jak również usuwanie poszczególnych obiektów.

Całą zawartość dokumentu można uzyskać poprzez metodę setContent(). Funkcja ta przetwarza ciąg przekazany jako dokument XML, w drzewo DOM, którego reprezentację stanowią elementy.

Jedną z zalet opisywanego narzędzia, z punktu widzenia tej pracy, jest możliwość sprawdzania poprawności składniowej i strukturalnej (czyli zgodności z definicją dokumentu zawartą z pliku DTD) dokumentu XML z zapisem zadania użytkowego. W efekcie już na etapie tworzenia nowego sterownika możemy uniknąć szeregu błędów, które w tradycyjnym podejściu do tworzenia zadań robotycznych, byłyby wychwytywane dopiero na etapie uruchamiania systemu sterującego.

3 Definicja języka opisu zadań robotycznych

Język opisu zadań użytkowych dla systemów wielorobotowych w notacji XML, powstały w oparciu o definicję automatu skończonego, został zaczerpnięty z pracy inżynierskiej Marka Kisiela. Z racji tego, iż zaistniała potrzeba wizualizacji graficznej maszyny stanowej konieczne było rozszerzenie istniejącego języka opisu zadań o dodatkowe atrybuty.

Należałoby tu nadmienić, że zdefiniowany język skupia się na opisie automatu skończonego zgodnego z modelem automatu Moore'a. Niewątpliwą korzyścią wynikająca z tak przyjętego założenia jest fakt, że akcja związana jest jedynie ze stanami automatu (nie ma akcji związanej z tranzycjami, jak jest to w przypadku automatu Mealego), co w znaczącym stopniu upraszcza konstruowanie zadania użytkowego, a także zapis jego definicji. Pociąga to jednak za sobą możliwość wystąpienia większej liczby stanów, bądź też ich trudniejszą definicję.

3.1 Podstawowe pojęcia

W trakcie opisu języka zadań robotycznych wykorzystywane będą następujące pojęcia:

- `zadanie` - zadanie robotyczne, które ma zostać wykonane na dostępnych manipulatorach bądź symulatorach. Sekwencja następujących po sobie akcji została stworzona w oparciu o definicję maszyny stanowej Moore'a. Akcje związane z poszczególnymi stanami zapisane są w sformalizowany sposób w dokumencie XML, którego struktura oparta jest w definicję typu dokumentu (DTD).

- `podzadanie` - jest to wydzielone w ramach zadania, mniejsze zadanie. Jego struktura jest analogiczna do struktury właściwego zadania.

- `tranzycja` - wskazuje na zmianę stanu automatu. Jest opisana warunkiem, który musi być spełniony aby przejście stanu było możliwe.

3.2 DTD definicji języka

Oprócz języka znaczników służącego do opisu zadań użytkowych w notacji XML, stworzona również została definicja tego języka zgodna z logiką automatu skończonego. Powstała definicja typu dokumentu zawiera zarówno listę dostępnych elementów służących do budowania maszyny stanowej, jak również relacje zachodzące pomiędzy poszczególnymi elementami wraz z możliwą liczebnością elementów.

W zależności od potrzeb, DTD może być bezpośrednio zawarte w pliku XML, bądź też przechowywane w oddzielnym pliku tekstowym, co w znaczącym stopniu zwiększa czytelność jak również pozwala na zastosowanie tego samego pliku DTD do wielu

różnych dokumentów. Przykład wstawienia definicji typu dokumentu, do zadania użytkowego zapisanego w języku XML, został przedstawiony na wydruku 4:

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

Wydruk 4: Definicja dokumentu DTD w pliku XML

Powyższy zapis oznacza, że korzeniem dokumentu jest element <TaskDescription>, a także że poprawność tego elementu powinna zostać sprawdzona w oparciu o dany plik DTD o nazwie `"fsautomat.dtd"`.

3.3 Element <TaskDesctiption>

Element <TaskDesctiption> jest korzeniem dokumentu XML z zadaniem użytkowym, przechowującym reprezentację całego zadania. Zapis z definicji typu dokumentu odnoszący się do elementu <TaskDesctiption> został przedstawiony na wydruku 5:

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

Wydruk 5: Definicja obiektu TaskDescription w pliku DTD

Powyższy zapis oznacza, iż poprawnie zdefiniowany element <TaskDescription> powinien zawierać co najmniej jeden element <State> oraz dowolną ilość elementów <SubTask>.

3.4 Element <State>

Element <State> jest dzieckiem elementu <TaskDescription>. Stanowi on jeden z podstawowych elementów wykorzystywanych przy specyfikacji zadania w oparciu o definicję automatu skończonego - jest on odpowiedzialny za reprezentację stanu.

Definicja DTD odnosząca się do elementu <State> została przedstawiona na wydruku 6:

```

  <!ELEMENT State((ROBOT|SetOfRobots)?, ECPGeneratorType?,transition+,
   TrajectoryFilePath?, Trajectory?, taskInit?, Speech?,
   AddArg?, Sensor?, TimeSpan?, GeneratorParameters?, Parameters?)>

<!ATTLIST State id CDATA #REQUIRED>
<!ATTLIST State type CDATA #REQUIRED>
<!ATTLIST State xpos CDATA >
<!ATTLIST State ypos CDATA >
<!ATTLIST State stateType CDATA #REQUIRED>
```

Wydruk 6: Definicja elementu <State> w pliku DTD

Kolejne linie przedstawione na powyższym listingu oznaczają odpowiednio:

- Linie 1 i 3 na powyższym wydruku określają wszystkie elementy, które są dziećmi elementu <State>, oraz ich preferowana kardynalność.
- Linie od 5 do 9 - oznaczają, że element <State> posiada trzy atrybuty obligatoryjne o nazwach id, type, stateType oraz dwa atrybuty opcjonalne o nazwach xpos i ypos (szerzej opisane w rozdziale 3.4.1).

3.4.1 Opis atrybutów

W pierwotnej wersji języka opisu zadań element <State> posiadał dwa atrybuty : id oraz type, jednak w związku z potrzebą wizualizacji grafu skierowanego, odpowiedzialnego za reprezentację graficzną automatu skończonego, zaistniała potrzeba rozszerzenia listy atrybutów o dodatkowe pozycje. Wszystkie atrybuty elementu <State> zostały przedstawione w tabeli 3.

Atrybut	Obowiązkowy	Wartość domyślna	Opis
id	TAK	Brak	Identyfikator stanu. Musi posiadać wartość unikalną w obrębie całego zadania.
type	TAK	Brak	Definiuje typ stanu. Może przyjmować jedną z dostępnych wartości.
xpos	NIE	0	Współrzędna x, w kartezjańskim układzie współrzędnych, pozycji na której znajduje się graficzna reprezentacja stanu.
ypos	NIE	0	Współrzędna y, w kartezjańskim układzie współrzędnych, pozycji na której znajduje się graficzna reprezentacja stanu.
stateType	TAK	3	Określa jeden z 6 dostępnych typów stanu w edytorze.

Tablica 3: Opis atrybutów elementu <State>

Atrybut id jest identyfikatorem stanu, w związku z czym musi mieć wartość unikalną w obrębie całego zadania robotycznego. Należy nadmienić, że może on przyjmować niemal dowolną wartość. Wyjątkiem są 2 wartości:

- `_STOP_` - jest to wartość przypisana jako cel (target) tranzycji (transition) w

momencie, kiedy automat skończony wychodzi z cyklu pracy (przechodzi w stan _STOP_),

- `_END_` - jest to wartość przypisana jako cel tranzycji i oznacza zakończenie wykonywania podzadania (SubTask). W praktyce oznacza zlecenie zdjęcia ze stosu id stanu docelowego.

W etapie definiowania systemu przyjęto, że stanem początkowym automatu jest stan, którego id ma wartość INIT. Jest on związany z inicjalizacją środowiska, które jest niezbędne do wykonania zadania użytkowego, zapisanego w dokumencie XML. Dlatego, w myśl przyjętej konwencji, stanem początkowym zawsze będzie stan o id="INIT"i typie="systemInitialization", np.

```
1 <State id="INIT" type="systemInitialization"/>
```

Wydruk 7: Przykład zapisu stanu inicjalizującego zadanie

Atrybut type ma decydujący wpływ na kształt elementu <State>. Może przyjmować jedynie jedną z dostępnych wartości:

- `systemInitialization` - wartość ta jest dostępne jedynie dla stanu o id="INIT". Oznacza, że akcja związana jest z inicjalizacją środowiska, które jest niezbędne do wykonania zadania użytkowego. Przykład stanu:

```
1 <State id="INIT" type="systemInitialization">
    <taskInit>
      <ecp name="ROBOT_IRP6_POSTUMENT">
        <ecp_smooth_gen>10</ecp_smooth_gen>
5     </ecp>
    </taskInit>
    <transition condition="true" target="ica"/>
  </State>
```

- `runGenerator` - wartość ta oznacza, że akcja stanu związana jest z uruchomieniem konkretnego generatora trajektorii. Przykład stanu:

```
1 <State id="stateID" type="runGenerator">
    <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
    <ECPGeneratorType>ECP_GEN_SMOOTH</ECPGeneratorType>
    <TrajectoryFilePath>irp6_ot_ap.trj</TrajectoryFilePath>
5   <Trajectory coordinateType="JOINT" numOfPoses="1">
      <Pose>
        <Velocity>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.05</Velocity>
        <Accelerations>0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1</Accelerations>
        <Coordinates>0 0.1 0.16 0 1.36 2.93 1.37 0.08</Coordinates>
10    </Pose>
```

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

- `emptyGenForSet` - wartość ta oznacza, że akcja stanu związana jest z uruchomieniem pustego generatora dla jednego zestawu robotów i oczekiwaniem na zakończenie wykonywania zadania przez inny zestaw. Przykład stanu:

```
<State id="stateID" type="emptyGenForSet">
  <SetOfRobots>
    <FirstSet>
      <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
      <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
    </FirstSet>
    <SecSet>
      <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
      <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
    </SecSet>
  </SetOfRobots>
  <transition condition="true" target="ica"/>
</State>
```

- `emptyGen` - wartość ta oznacza, że akcja stanu jest związana z uruchomieniem pustego generatora. Przykład stanu:

```
<State id="INIT" type="emptyGen">
  <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
  <transition condition="true" target="ica"/>
</State>
```

- `wait` - wartość ta oznacza, że akcja stanu jest związana z uruchomieniem generatora zatrzymującego wykonywanie zadania, na wskazany w stanie kwant czasu (wartość podana jest w milisekundach).

```
<State id="INIT" type="wait">
  <TimeSpan>1000</TimeSpan>
  <transition condition="true" target="ica"/>
</State>
```

- `stopGen` wartość ta oznacza, że akcja stanu polega na zatrzymaniu ruchu wykonywanego przez pojedynczego robota lub zestaw robotów. Przykład stanu:

```
<State id="stateID" type="stopGen">
  <SetOfRobots>
    <FirstSet>
```

```
        <ROBOT>ROBOT_FESTIVAL</ROBOT>
        <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
        <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
      </FirstSet>
      <SecSet>
        <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
        <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
        <ROBOT>ROBOT_FESTIVAL</ROBOT>
      </SecSet>
    </SetOfRobots>
    <transition condition="true" target="ica"/>
</State>
```

- `cubeStateInit` - wartość oznacza, że akcja związana jest z inicjalizacją programu układającego kostkę Rubika. Przykład stanu:

```
<State id="stateID" type="cubeStateInit">
  <Parameters>BLUE GREEN RED ORANGE WHITE YELLOW</Parameters>
  <transition condition="true" target="ica_1>>identify_colors_2␣"/>
</State>
```

- `initiateSensorReading` - wartość ta oznacza, że akcja stanu związana jest z inicjalizacją odczytów wartości z konkretnego sensora. Przykład stanu:

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

- `getSensorReading` - wartość ta oznacza, że akcja stanu związana jest z pobraniem wartości z konkretnego sensora. Przykład stanu:

```
<State id="stateID" type="getSensorReading">
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <transition condition="true" target="ica"/>
</State>
```

- `cubeStateWriting` - wartość ta oznacza, że akcja stanu związana jest z zapisaniem aktualnego stanu kostki Rubika, a w zasadzie obiektu reprezentującego kostkę. Przykład stanu:

```
<State id="stateID" type="cubeStateWriting">
  <AddArg>1</AddArg>
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <transition condition="true" target="ica"/>
</State>
```

- `cubeStateChange` - wartość ta oznacza, że akcja związana jest ze zmianą aktualnego stanu kostki Rubika, a w zasadzie obiektu reprezentującego kostkę. Przykład stanu:

```
<State id="stateID" type="cubeStateChange">
  <AddArd>4</AddArg>
  <transition condition="true" target="ica"/>
</State>
```

- `communicateWithSolver` - wartość ta oznacza, że akcja stanu polega na komunikacji z solwerem, który jest odpowiedzialny za wyznaczenie sekwencji ruchów układających kostkę Rubika. Przykład stanu:

```
<State id="stateID" type="communicateWithSolver">
<transition condition="stateOperationResult" target="communicate_wws_4"/>
<transition condition="true" target="ica"/>
</State>
```

- `manipulationSeqTranslation` - wartość ta oznacza, że akcja związana ze stanem, polega na przetworzeniu sekwencji ruchów otrzymanej od solwera, do postaci zrozumiałej dla dostępnego interpretera automatu skończonego po stronie systemu MRROC++. Przykład stanu:

```
<State id="stateID" type="manipulationSeqTranslation">
  <transition condition="true" target="ica"/>
</State>
```

- `intermediateState` - wartość ta oznacza, że ze stanem nie jest związana żadna akcja - stanowi on jedynie stan pośredni. Konstrukcja ta jest wykorzystywana w przypadku delegowania wykonywania podzadań. Przykład stanu:

```
<State id="INIT" type="intermediateState">
  <transition condition="true" target="fto_CL_0_1>>man_2"/>
</State>
```

Atrybuty xpos i ypos odpowiadają współrzędnym odpowiednio X i Y położenia graficznej reprezentacji stanu na planszy edytora, w kartezjańskim układzie współrzędnych. Mogą one przyjmować wartość dowolnej liczny naturalnej, jednak w przyjętej definicji edytora, możliwe są jedynie wartości odpowiadające wielkości aktualnej sceny.

Na etapie definicji języka opisu reprezentacji graficznej przyjęto, że oba atrybuty będą miały wartość domyślna równą 0. Dzięki czemu, możliwe jest poprawne wczytanie semantyczne już istniejących zadań robotycznych, które nie miały wcześniej reprezentacji graficznej, a także ich późniejsza wizualizacja.

Ostatnim, wspomnianym już atrybutem elementu <State> jest stateType. Określa on rodzaj elementu odpowiadającego za reprezentację stanu w edytorze. Może przyjmować jedną z sześciu wartości:

- `stateType=1` - oznacza, że wybrany został stan `State INIT`, który jest stanem początkowym całego automatu. Ma on id o wartości INIT i `type=systemInitialization`. Z racji tego, że odpowiada on za inicjalizację środowiska, które jest potrzebne do wykonania zadania, tylko w tym stanie możliwe jest dodanie elementu <taskInit>.
- `stateType=2` - oznacza, że wybrany został stan Task INIT, który jest stanem początkowym podzadania.
- `stateType=3` - oznacza, że wybrany został stan State, który jest zwykłym stanem automatu.
- `stateType=4` - oznacza, że wybrany został element Subtask, reprezentujący podzadanie.
- `stateType=5` - oznacza, że wybrany został stan `END State`, który jest stanem końcowym automatu. Posiada on tranzycję, której celem jest `id="_STOP_"`.
- `stateType=6` - oznacza, że wybrany został stan `END Task`, który jest stanem końcowym podzadania. Posiada on tranzycję, której celem jest `id="_END_"`.

Atrybut stateType został utworzony na potrzeby wizualizacji graficznej automatu, aby w prosty sposób można było odróżnić stan początkowy lub końcowy od pozostałych stanów automatu. Dodatkowym jego atutem jest możliwość kontroli poprawności atrybutów dostępnych dla danego stanu już na etapie specyfikacji automatu.

3.4.2 Dzieci elementu <State>

Dziećmi elementu <State> są:

- `<ROBOT>` - element opcjonalny przechowujący nazwę robota, z którym związana jest akcja zawarta w danym stanie. Występuje zamiennie z elementem <SetOfRobots>.
- `<SetOfRobots>` - element opcjonalny, przechowujący dwa zbiory nazw robotów, z którymi związana jest akcja zawarta w danym stanie. Występuje zamiennie z elementem <ROBOT>.
- `<ECPGeneratorType>` - element opcjonalny, przechowujący nazwę generatora. Występuje co najwyżej jeden raz.

- `<GeneratorFilePath>` - element opcjonalny, przechowujący ścieżkę do pliku z trajektorią. Występuje co najwyżej jeden raz.
- `<Trajectory>` - element opcjonalny, przechowujący trajektorię według przyjętej konwencji. Występuje co najwyżej jeden raz.
- `<taskInit>` - element opcjonalny, odpowiedzialny za inicjalizację środowiska, które jest potrzebne do wykonania zadania. Występuje co najwyżej jeden raz (dostępny jest jedynie w stanie początkowym automatu).
- `<Speech>` - element opcjonalny, przechowujący tekst wypowiadany przez robota `ROBOT_FESTIVAL`. Występuje co najwyżej jeden raz.
- `<AddArg>` - element opcjonalny, przechowujący wartość numeryczną będącą dodatkowym argumentem związanym z akcją danego stanu. Występuje co najwyżej jeden raz.
- `<Sensor>` - element opcjonalny, przechowujący nazwę konkretnego sensora. Występuje co najwyżej jeden raz.
- `<TimeSpan>` - element opcjonalny, przechowujący wartość numeryczną będącą parametrem wywołania akcji typu wait. Pojawia się najwyżej jeden raz.
- `<GeneratorParameters>` - element opcjonalny, przechowujący argumenty wywołania konkretnego generatora. Występuje co najwyżej jeden raz.
- `<Parameters>` - element opcjonalny, przechowujący dodatkowe parametry nie uwzględnione w innych elementach. Pojawia się najwyżej jeden raz.
- `<transition>` - element opcjonalny. Definiuje tranzycję o początku w danym stanie i końcu w jednym z dostępnych stanów automatu. Występuje co najmniej jeden raz.

3.5 Element <SubTask>

W pierwotnej wersji języka opisu zadań element <SubTask> nie posiadał żadnych atrybutów, jednak w związku z zaistniałą potrzebą jego wizualizacji, lista atrybutów została powiększona o atrybut name, odpowiedzialny za nazwę podzadania. Jest on również rodzicem stworzonego elementu <TaskInstance> odpowiadającego za reprezentację graficzną podzadania (element <SubTask> musi zawierać co najmniej jeden element <TaskInstance>). Szczegółowy opis atrybutów elementu <TaskInstance> został przedstawiony w tabeli 4.

Atrybut	Obowiązkowy	Wartość domyślna	Opis
instance	TAK	0	Umożliwia wielokrotne umieszczenie reprezentacji podadania w reprezentacji automatu. Wskazuje reprezentacje podzadania z która związana jest tranzycja.
xpos	NIE	0	Współrzędna x, w kartezjańskim układzie współrzędnych, pozycji na której znajduje się graficzna reprezentacja podzadania.
ypos	NIE	0	Współrzędna y, w kartezjańskim układzie współrzędnych, pozycji na której znajduje się graficzna reprezentacja podzadania.

Tablica 4: Opis atrybutów elementu <SubTask>

Potomkami elementu <Subtask> są elementy <State> oraz <TaskInstance>. Wpis z pliku fsautomat.dtd odnoszący się do elementu <SubTask> został przedstawiony na wydruku 8:

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

Wydruk 8: Element<SubTask> w pliku DTD

Powyższy zapis oznacza, iż poprawnie zdefiniowany element <SubTask> powinien zawierać dowolną, niezerową ilość elementów <State> oraz <TaskInstance>.

Cechą charakterystyczną podzadania jest fakt, iż stan wyjściowy z podzadania zawiera tranzycję, której atrybut target ma wartość `_END_`. Oznacza to zakończenie wykonywania podzadania oraz wymusza zdjęcie nazwy stanu docelowego ze stosu automatu skończonego.

3.6 Element <transition>

Element <transition> jest jednym z podstawowych elementów wykorzystywanych przy opisie automatu skończonego opisującego zadanie użytkowe. Jego zadaniem jest reprezentacja tranzycji, czyli przejścia między stanami. Atrybutami elementu <transition> są: condition i target, ich szczegółowy opis został przedstawiony w tabeli 5.

Zgodnie z definicją języka opisu zadań, każdy stan automatu powinien zawierać co najmniej jedną tranzycję. Jak wcześniej wspomniałam, warunek przejścia zawarty w tranzycji, może przyjmować wartość logiczną (true lub false) lub proste warunki logiczne.

Atrybut	Obowiązkowy	Wartośćdomyślna	Opis
target	TAK	Brak	Jest to id stanu docelowego tranzycji. Musi wskazywać na jeden ze stanów istniejących w automacie.
condition	TAK	BRAK	Jest to warunek przejścia z aktualnego stanu do stanu wskazanego jako cel tranzycji. Może być to wartość logiczna (true lub false), bądz też warunek logiczny, np. iniFile.irp6p_compliant==true.

Tablica 5: Opis atrybutów elementu <transition>

Najczęściej zapis elementu <transition> stosowany w opisie zadań użytkowych, które powstały w ramach testów przeprowadzonych na potrzeby sprawdzenia poprawności niniejszej pracy, ma postać:

```
1   <transition condition="true" target="nextState"/>
```

Wydruk 9: Przykładowy zapis tranzycji

Wartość true dla atrybutu condition, oznacza, że po wykonaniu akcji związanej z aktualnym stanem, automat przejdzie bezpośrednio do kolejnego stanu, określonego poprzez atrybut target. Tak zapis oznacza bezwarunkowe przejście do kolejnego stanu automatu. Wykorzystywane jest głównie w celu opisania sekwencji przejść.

Kolejne podejście oparte jest na wariantowości wykonania przejścia do kolejnego stanu. Umożliwia ono wybór kolejnego stanu na podstawie wartości warunku przejścia, czego przykład zgodny z przyjętą definicją języka został przedstawiony na poniższym wydruku:

```
1   <transition condition="iniFile.irp6p_compliant" target="nextState1"/>
    <transition condition="!iniFile.irp6p_compliant" target="nextState2"/>
```

Zapis ten oznacza: jeżeli wartość pola `iniFile.irp6p_compliant` w pliku konfiguracyjnym fsautomat.ini ma wartość true, następuje przejście do stanu `nextState1`, w przeciwnym wypadku następuje przejście do stanu `nextState2`.

Należałoby nadmienić, iż język opisu zadań użytkowych w swojej pierwotnej wersji zakładał, że każdy stan musi zawierać co najmniej jeden element <transition>, którego atrybut condition ma wartość true. Miało to zapobiec sytuacji, w której żaden z przedstawionych warunków nie jest spełniony, co pociągałoby za sobą zawieszenie automatu na aktualnym stanie. W związku z faktem, iż tranzycje dodawane są do elementu <State> w przypadkowej kolejności, mogło to doprowadzić do różnych wyników działania automatu skończonego w zależności od kolejności wystąpień poszczególnych

tranzycji, ponieważ kilka warunków przejścia mogło być spełnionych jednocześnie. W związku z czym, zaistniała potrzeba modyfikacji istniejącego podejścia, umożliwiająca uniezależnienie działania automatu od kolejności tranzycji. W chwili obecnej za poprawny uznaje się stan, w którym suma wszystkich kombinacji iloczynów warunków tranzycji stanowi zbiór rozłączny (wszystkie iloczyny par warunków przejść są zbiorami pustymi).

3.6.1 Atrybut target

Atrybut target elementu <transition> może zostać zapisany na dwa sposoby:

- `target="stan_1"` - jest to konstrukcja, w której wartością atrybutu jest id stanu docelowego tranzycji,
- `target="ica_1»identify_colors_1"` - jest to konstrukcja, w której wartość atrybutu zawiera id dwóch stanów oddzielonych separatorem `"»"`. Jest to instrukcja wykonania podzadania o stanie początkowym określonym przez id znajdujące się po lewej stronie separatora, a następnie przejście do stanu, którego atrybut id ma wartość opisaną przez wyrażenie znajdujące się po prawej stronie separatora. Wymagane jest, aby po lewej stronie separatora `"»"` znajdowało się id poprawnie zdefiniowanego podzadania, natomiast po jego lewej stronie id istniejącego stanu automatu.

  Przedstawiony wyżej zapis `target="ica_1»identify_colors_1"` oznacza: idź do stanu `identify_colors_1` ale najpierw wykonaj podzadanie, którego stanem początkowym jest stan o id=`"icca_1"`.

3.7 Element <ROBOT>

Element <ROBOT> przechowuje nazwę robota, z którym związana jest akcja wykonywana w danym stanie. Element może przyjmować jedną z dostępnych wartości, odpowiadających nazwom robotów dostępnych w systemie MRROC++. W chwili obecnej są to:

- `ROBOT_UNDEFINED`
- `ROBOT_IRP6_ON_TRACK`
- `ROBOT_IRP6_POSTUMENT`
- `ROBOT_FESTIVAL`
- `ROBOT_CONVEYOR`
- `ROBOT_SPEAKER`

- ROBOT_IRP6_MECHATRONIKA
- ROBOT_ELECTRON
- ROBOT_HAND
- ROBOT_SPEECHRECOGNITION

Poniższy przykład przedstawia przykład użycia opisywanego elementu:

```
1  <ROBOT>ROBOT_FESTIVAL</ROBOT>
```

3.8 Element <SetOfRobots>

Element <SetOfRobots> jest rodzicem dwóch elementów: <FirstSet> oraz <SecSet>, spośród których, każdy może wystąpić co najwyżej jeden raz. Ich potomkami są elementy <ROBOT>, przy czym element <ROBOT> musi wystąpić co najmniej jeden raz w każdym z występujących zbiorów.

Definicja DTD odnosząca się do elementu <SetOfRobots> została przedstawiona na listingu 10:

```
1  <!ELEMENT SetOfRobots(FirstSet?, SecSet?)>

   <!ELEMENT FirstSet(ROBOT+)>
   <!ELEMENT SecSet(ROBOT+)>
5

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

Wydruk 10: Element <SetOfRobots> w pliku DTD

Przykład opisywanego elementu został przedstawiony na listingu:

```
1  <SetOfRobots>
     <FirstSet>
       <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
       <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
5      <ROBOT>ROBOT_FESTIVAL</ROBOT>
     </FirstSet>
     <SecSet>
       <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
       <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
10     <ROBOT>ROBOT_FESTIVAL</ROBOT>
     </SecSet>
   </SetOfRobots>
```

3.9 Element <ECPGeneratorType>

Element <ECPGeneratorType> odpowiedzialny jest za określenie generatora. Może przyjmować jedną z wartości, która zdefiniowana jest w strukturze MRROC++ i odnosi się do konkretnego generatora:

- `ECP_GEN_SMOOTH` - oznacza generator `ecp_smooth_generator`
- `ECP_GEN_TRANSPARENT` - oznacza generator `ecp_generator_t`
- `ECP_GEN_TEACH_IN` - oznacza generator `ecp_teach_in_generator`
- `ECP_GEN_TFF_NOSE_RUN` - oznacza generator `ecp_tff_nose_run_generator`
- `RCSC_GRIPPER_OPENING` - oznacza generator `ecp_sub_task_gripper_opening`
- `ECP_GEN_FESTIVAL` - oznacza generator `festival_generator`
- `ECP_GEN_BIAS_EDP_FORCE` - oznacza generator `bias_edp_force_generator`
- `ECP_GEN_TFF_GRIPPER_APPROACH` - oznacza użycie generatora `ecp_tff_gripper_approach_generator`
- `ECP_GEN_TFF_RUBIK_FACE_ROTATE` - oznacza użycie generatora `ecp_tff_rubik_face_rotate_generator`
- `ECP_GEN_TFF_RUBIK_GRAB` - oznacza generator `ecp_tff_rubik_grab_generator`
- `ECP_TOOL_CHANGE_GENERATOR` - oznacza generator `ecp_tool_change_generator`
- `ECP_WEIGHT_MEASURE_GENERATOR` - oznacza użycie generatora `weight_meassure_generator`

Definicja DTD odnosząca się do elementu <ECPGeneratorType> została przedstawiona na listingu:

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

Poniższy wydruk prezentuje przykład wystąpienia elementu <ECPGeneratorType> w pliku XML.

```
1   <ECPGeneratorType>ECP_GEN_SMOOTH</ECPGeneratorType>
```

3.10 Element <Trajectory>

Element <Trajectory> stanowi reprezentację trajektorii zgodną z przyjętą definicją języka. Odpowiada ona trajektorii wykorzystywanej przez `ecp_smooth_generator`. Jego atrybuty zostały szczegółowo opisane w tabeli 6:

Element <Trajectory> jest rodzicem obligatoryjnego elementu <Pose>, występującego co najmniej jeden raz, który z kolei jest rodzicem elementów:

Atrybut	Obowiązkowy	Wartość domyślna	Opis
coordinateType	TAK	BRAK	Wartość określa tryb pracy generatora. Może przyjmować jedynie jedną z dostępnych wartości (ABSOLUTE lub RELATIVE).
numOfPoses	TAK	BRAK	Liczba pozycji, które zawiera dana trajektoria.

Tablica 6: Opis atrybutów elementu <Trajectory>

- <Velocity> - Przechowuje wartości prędkości, rozdzielone separatorem (przyjęto, że separatorem jest spacja). Występuje dokładnie jeden raz.
- <Accelerations> - Przechowuje wartości przyśpieszeń, rozdzielone separatorem (przyjęto, że separatorem jest spacja). Występuje dokładnie jeden raz.
- <Coordinates> - Przechowuje wartości współrzędnych położenia, rozdzielone separatorem (przyjęto, że separatorem jest spacja). Występuje dokładnie jeden raz.

Do niedawna element <Pose> zawierał także dwa inne elementy, odpowiedzialne odpowiednio za:

- <StartVelocity> - przechowywał wartości prędkości początkowych,
- <EndVelocity> - przechowywał wartości prędkości końcowych.

Jednak w związku ze zmianą generatora `ecp_smooth_gen`, konieczna była modyfikacja istniejącej definicji, polegająca na wyeliminowaniu wyżej wspomnianych elementów, w celu zapewnienia spójności z nową wersją generatora.

Przykład trajektorii zapisanej w pliku z rozszerzeniem .trj został przedstawiony poniżej:

```
1 JOINT
  2
  0.55   0.5  0.5  0.5  0.5  0.5  0.5  1.0
```

```
0.2 0.1 0.1 0.1 0.1 0.1 0.1 0.01
0.0 0.10 2.27 0.02 1.21 2.57 -2.66 0.05

0.55 0.5 0.5 0.5 0.5 0.5 0.5 1.0
0.2 0.1 0.1 0.1 0.1 0.1 0.1 0.01
0.0 0.10 2.27 0.02 1.21 2.57 -2.66 0.065
```

Odpowiada to zapisowi stanu po stronie XML:

```
<Trajectory coordinateType="JOINT" numOfPoses="2">
  <Pose>
    <Velocity>0.55 0.5 0.5 0.5 0.5 0.5 0.5 1.0</Velocity>
    <Accelerations>0.2 0.1 0.1 0.1 0.1 0.1 0.1 0.01</Accelerations>
    <Coordinates>0.0 0.10 2.27 0.02 1.21 2.57 -2.66 0.05</Coordinates>
  </Pose>
  <Pose>
    <Velocity>0.55 0.5 0.5 0.5 0.5 0.5 0.5 1.0</Velocity>
    <Accelerations>0.2 0.1 0.1 0.1 0.1 0.1 0.1 0.01</Accelerations>
    <Coordinates>0.0 0.10 2.27 0.02 1.21 2.57 -2.66 0.065</Coordinates>
  </Pose>
</Trajectory>
```

3.11 Element <GeneratorFilePath>

Element <GeneratorFilePath> zawiera ścieżkę do pliku z trajektorią, wykorzystywaną przez generator `ecp_smooth_gen`. Element ten jest używany zamiennie z elementem <Trajectory>, w zależności od konfiguracji zawartej w pliku konfiguracyjnym.

Definicja DTD odnosząca się do elementu <GeneratorFilePath> została przedstawiona na listingu:

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

Przykład wykorzystanie omawianego elementu przedstawia listing poniżej:

```
<GeneratorFilePath>zadanie.trj</GeneratorFilePath>
```

3.12 Element <taskInit>

Element <taskInit> zawiera informacje dotyczące inicjalizacji środowiska, które jest niezbędne do prawidłowego wykonania zadania robotycznego. Element ten występuje jedynie w stanie początkowym maszyny stanowej, który jest odpowiedzialny za inicjalizację wskazanych elementów po stronie systemu MRROC++, co pociąga za sobą id="INIT" oraz type="systemInitialization".

Definicja DTD odnosząca się do elementu <taskInit> została przedstawiony na listingu:

```
<!ELEMENT taskInit(ecp+,mp?)>
<!ELEMENT \ECP\ (ecp_gen_t?, ecp_tool_change_gen?, ecp_tff_nose_run_gen?,
ecp_tff_rubik_grab_gen?, ecp_tff_gripper_approach_gen?,
ecp_tff_rubik_face_rotate_gen?, ecp_teach_in_gen?, bias_edp_force_gen?,
ecp_smooth_gen?, weight_meassure_gen?, ecp_sub_task_gripper_opening?)>

<!ATTLIST \ECP\ name CDATA #REQUIRED>
<!ELEMENT ecp_gen_t (#PCDATA)>
<!ELEMENT ecp_tool_change_gen (#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_gen (#PCDATA)>
<!ELEMENT ecp_smooth_gen (#PCDATA)>
<!ELEMENT weight_meassure_gen (#PCDATA)>
<!ELEMENT ecp_sub_task_gripper_opening (#PCDATA)>

<!ELEMENT \MP\ ( cube_state?, Sensor *, Transmitter?)>
<!ELEMENT cube_state (#PCDATA)>
<!ELEMENT Transmitter (#PCDATA)>
<!ELEMENT Sensor (#PCDATA)>
```

Wydruk 11: Element <taskInit> w pliku DTD

Element <taskinit> jest rodzicem następujących elementów:

- `<ecp>` - element obligatoryjny, zawierający listę elementów będących odpowiednikami obiektów, które mają być powołane po stronie konkretnego ECP. Występuje co najmniej jeden raz.
- `<mp>` - element opcjonalny, przechowujący informacje o obiektach, które mają zastać powołane do życia po stronie systemu MRROC++, na poziomie MP. Element pojawia się co najwyżej jeden raz.

3.12.1 Element <ecp>

<ecp> jest elementem przechowującym elementy odpowiadające obiektom, które zostały powołane go stronie konkretnego procesu ECP, który jest określany na podstawie atrybutu name, odpowiadającemu nazwie robota z którym dane obiekty mają być związane.

Atrybut `name` może przyjmować wartości odpowiadające nazwom robotów dostępnych w systemie MRROC++. Stąd też ograniczenia odnoszące się do tego atrybutu są analogiczne do ograniczeń związanych z zawartością elementu <ROBOT>.

<ecp> jest rodzicem elementów, będących odpowiednikami obiektów powołanych do życia na poziomie ECP, w związku z czym ich nazwy są analogicznymi do użytych w systemie MRROC++. Do wspomnianych elementów należą:

- `<ecp_gen_t>`
- `<ecp_tool_change_gen>`
- `<ecp_tff_nose_run_gen>`
- `<ecp_tff_rubik_grab_gen>`
- `<ecp_tff_gripper_approach_gen>`
- `<ecp_tff_rubik_face_rotate_gen>`
- `<ecp_teach_in_gen>`
- `<bias_edp_force_gen>`
- `<ecp_smooth_gen>`
- `<weight_meassure_gen>`
- `<ecp_sub_task_gripper_opening>`

Pojawienie się jakiegokolwiek z wyżej wymienionych elementów w sekcji <ecp> oznacza, że na poziomie ECP robota wymienionego w atrybucie name ma być powołany do życia obiekt odpowiadający danemu elementowi. Dodatkowo, jeśli element ten posiada wartość, powinna ona zostać przekazana do konstruktora danego elementu. Przykładowy kod w języku XML został zamieszczony na wydruku:

```
1 <ecp name="ROBOT_IRP6_ON_TRACK">
    <ecp_tool_change_gen>1</ecp_tool_change_gen>
    <ecp_sub_task_gripper_opening></ecp_sub_task_gripper_opening>
  </ecp>
```

Wydruk 12: Przykładowy zapis elementu <ecp>

Przedstawiony przykład prezentuje powołanie dwóch obiektów: `ecp_sub_task_gripper_opening` oraz `ecp_tool_change_generator` na poziomie procesu `ecp_irp6ot_fsautomat`. Do konstruktora pierwszego z nich została przekazana wartość 1, która jest następnie rzutowana na wartość logiczna (w tym przypadku jest to wartość true).

3.12.2 Element <mp>

<mp> jest elementem przechowującym informacje, które są wykorzystywane do inicjalizacji obiektów na poziomie procesu MP. Występuje w zależności od tego, czy inicjalizacja konkretnych obiektów jest potrzebna do wykonania danego zadania użytkowego. Element może pojawić się co najwyżej jeden raz.

<mp> jest rodzicem elementów, będących odpowiednikami obiektów powoływanych do życia na poziomie konkretnego MP:

- `<Sensor>` - element opcjonalny, zawierający konfigurację obiektu `sensor_m`. Element może występować wielokrotnie.
- `<Transmitter>` - element zawierający konfigurację obiektu `transmitter_m`. Pojawia się najwyżej jeden raz.
- `<cube_state>` - element opcjonalny, odpowiadający obiektowi CubeState. Pojawiasię co najwyżej jeden raz.

3.13 Element <Speech>

Element <Speech> przechowuje argument wywołania generatora `ECP_GEN_FESTIVAL` - tekst wypowiadany przez robota `ROBOT_FESTIVAL`. Może on więc występować jedynie w stanach, których akcja związana jest z wymienionym robotem i generatorem.

Przykład wykorzystania tego elementu został przedstawiony na wydruku:

```
1     <State id="stateID" type="runGenerator">
       <ROBOT>ROBOT_FESTIVAL</ROBOT>
       <ECPGeneratorType>ECP_GEN_FESTIVAL</ECPGeneratorType>
       <Speech>Jestem fajnym robotem</Speech>
5      <transition condition="true" target="nextState"/>
     </State>
```

Wydruk 13: Przykład wykorzystania elementu <Speech>

3.14 Element <AddArg>

Element <AddArg> przechowuje dodatkowe dane numeryczne konieczne do wykonania akcji związanej ze stanem, który zawiera ten element. W zależności od typu stanu, w którym został zapisany może oznaczać:

- stan, gdzie type=`"runGenerator"` oznacza, że wartość numeryczna przechowywana przez <AddArg> jest przekazywana jako drugi argument metody `set_next_ecps_state()`, którego wartość domyślna w przyjętej implementacji wynosi 0. Poniżej przykład użycia w omawianym kontekście:

```
<State id="approach_4" type="␣runGenerator">
  <ROBOT>ROBOT_FESTIVAL</ROBOT>
  <ECPGeneratorType>ECP_GEN_TRANSPARENT</ECPGeneratorType>
  <Speech></Speech>
  <AddArg>1</AddArg>
  <transition condition="true" target="approach_5"/>
</State>
```

Wydruk 14: Przykład użycia elementu <AddArg>

- stan gdzie type="emptyGen" oznacza, że wartość numeryczna przechowywana przez <AddArg>, jest przekazywana jako pierwszy argument metody run_ext_empty_gen(), którego wartość domyślna, podobnie jak wcześniej, wynosi 0. Poniżej przykład użycia w omawianym kontekście:

```
<State id="approach_7" type="emptyGen">
  <ROBOT>ROBOT_IRP6_POSTUMENT</ROBOT>
  <AddArg>1</AddArg>
  <transition condition="true" target="approach_8"/>
</State>
```

- stan gdzie type="cubeStateWriting" służy do identyfikacji ścianki kostki Rubika, która jest aktualnie zapisywana. Poniżej przykład użycia w omawianym kontekście:

```
<State id="identify_colors_2" type="cubeStateWriting">
  <AddArg>0</AddArg>
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <transition condition="true" target="identify_colors_3"/>
</State>
```

3.15 Element <Sensor>

Element <Sensor> przechowuje nazwę sensora, z którym związana jest akcja zawarta w danym stanie. Może przyjmować wartość jednego z dostępnych w systemie MRROC++ czujników. Poniższy wydruk prezentuje przykład wykorzystania tego elementu:

```
<State id="stateID" type="initiateSensorReading">
  <Sensor>SENSOR_CAMERA_ON_TRACK</Sensor>
  <transition condition="true" "target="nextState"/>
␣␣</State>
```

3.16 Element <TimeSpan>

Element <TimeSpan> przechowuje wartość numeryczną, wyrażoną w milisekundach, będącą parametrem wywołania akcji typu wait, zatrzymującej działanie systemu MRROC++ na podany kwant czasu. Poniższy listing prezentuje przykład użycia opisywanego elementu:

```
1   <TimeSpan>1000</TimeSpan>
```

3.17 Element <GeneratorParameters>

GeneratorParameters jest elementem przechowującym dodatkowe dane, które są wykorzystywane jako parametr przy wykonywaniu ruchu przez wskazany generator na poziomie procesu ECP. Może zawierać kilka argumentów, które oddzielone są między sobą separatorem. Przyjęto, że separatorem tym jest znak tabulacji. Poniżej zamieszczono wydruk ilustrujący użycie tego elementu:

```
1   <State id="approach_28" type="runGenerator">
      <ROBOT>ROBOT_IRP6_ON_TRACK</ROBOT>
      <ECPGeneratorType>ECP_GEN_TFF_RUBIK_GRAB</ECPGeneratorType>
      <GeneratorParameters>0.057 0.00005 0</GeneratorParameters>
5     <transition condition="true"target="approach_29"/>
    </State>
```

Wydruk 15: Przykład wykorzystania elementu <GeneratorParameters>

Powyższy stan skutkuje wykonaniem następującego kodu na poziomie procesu ECP:

```
1 rgg->configure(0.057, 0.00005, 0);
  rgg->Move( );
```

gdzie rgg oznacza generator `ecp_tff_rubik_grab_generator`.

3.18 Element <Parameters>

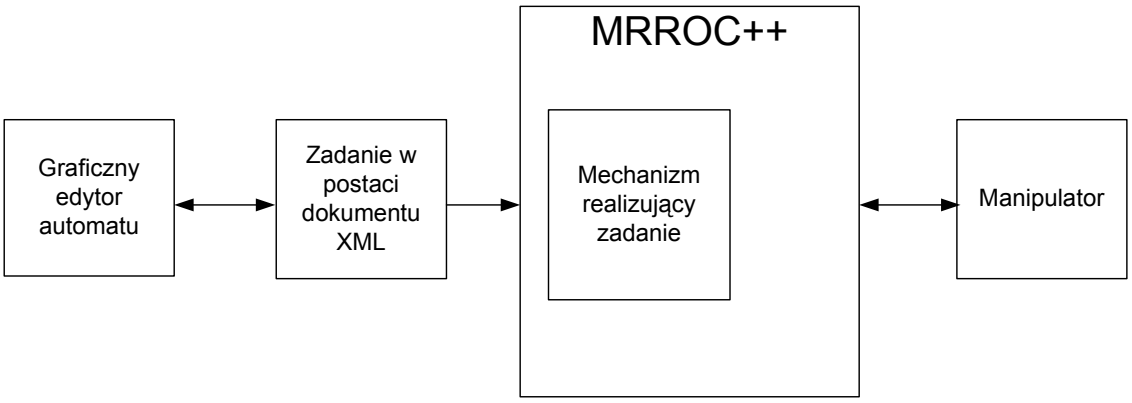
Element <Parameters> zawiera dodatkowe parametry wykorzystywane w wykonywaniu akcji związanej z aktualnym stanem. Może przechowywać kilka argumentów, które oddzielone są między sobą separatorem, którym jest znak tabulacji. Poniższy wydruk ilustruje użycie opisywanego elementu <Parameters>:

```
1   <State id="identif_colors_1" type="cubeStateInit">
      <Parameters>BLUE GREEN RED ORANGE WHITE YELLOW</Parameters>
      <transition condition="true" target="_STOP_"/>
    </State>
```

Jak widać na przedstawionym przykładzie, parametry przechowywane przez element <Parameters>, używane są do inicjalizacji stanu kostki Rubika.

4 Koncepcja aplikacji

Wymaganą funkcjonalność tworzenia zadań użytkowych dla robotów, które mogłyby być wykonywane w systemie MRROC++ oferuje realizacja graficznego edytora, będącego tematem niniejszej pracy inżynierskiej. Funkcje niezbędne do prawidłowego wykonania zadania zostały podzielone między graficzny edytor, uruchamiany po stronie użytkownika oraz ściśle zintegrowany z systemem MRROC++ system odpowiedzialny za prawidłową interpretację, a także wykonanie zadania. Ogólna struktura działania systemu została przedstawiona na schemacie 5.



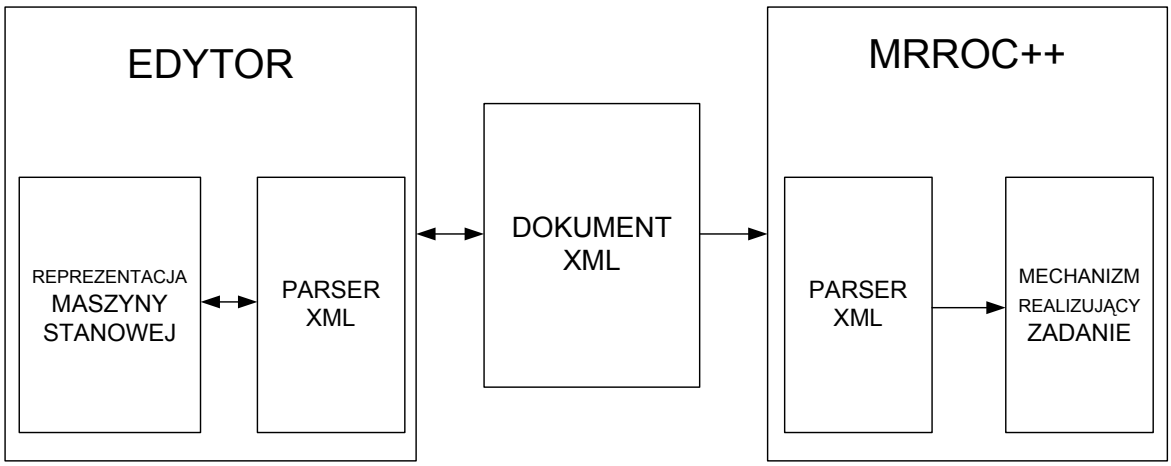
Rysunek 5: Ogólny schemat działania systemu

Edytor umożliwia tworzenie zadań dla robotów, które następnie są zapisywane w formacie XML. Zadania w takiej postaci, mogą zostać poprawnie zinterpretowane i uruchomione w systemie MRROC++ bądź na dostępnych symulatorach.

4.1 Schemat rozwiązania

Kluczowym celem powstającej aplikacji było umożliwienie tworzenia automatu skończonego, zgodnego z przyjętym językiem opisu w możliwie prosty sposób. Celem takiego rozwiązania było umożliwienie korzystania z systemu MRROC++ zarówno twórcom systemu jak i początkującym użytkownikom.

Na podstawie automatu skończonego stworzonego w edytorze generowany jest plik zawierający zadanie robotyczne, zapisane z języku XML, które może zostać wykonane w systemie MRROC++. Zawiera on również atrybuty definiujące położenie poszczególnych stanów na ekranie edytora, dlatego też możliwe jest poprawne odtworzenie reprezentacji graficznej automatu. Stworzenie opcji wizualizacji automatu skończonego wymagało modyfikacji języka opisu zadań robotycznych. Konieczne było bowiem zapisanie pozycji poszczególnych elementów na ekranie edytora. Struktura działania systemu została przedstawiona na rysunku 6.



Rysunek 6: Szczegółowy schemat działania systemu

Po stronie edytora zaimplementowany został parser XML, którego zadaniem jest zapisanie zadania w postaci pliku XML. Ma on również umożliwić wczytanie już istniejących zadań w celu ich modyfikacji (ich wizualizację, a także zamianę na poprawny semantycznie automat skończony).

Aby zapewnić niezależność reprezentacji graficznej i właściwego modelu danych, aplikacja powstała w oparciu o istniejący model gui-controler-model. Dzięki temu możliwa jest stała kontrola poprawności właściwych danych oraz całkowita separacja wizualizacji graficznej maszyny stanowej od semantycznej części zadania.

4.2 Wymagania wobec powstającej aplikacji

Kluczowym celem niniejszej pracy inżynierskiej było zaprojektowanie oraz implementacja narzędzia, spełniającego szereg stawianych wobec niego wymagań:

- wizualizacja automatu skończonego, reprezentowanego przez graf skierowany,
- definiowanie zadań użytkowych dla systemów wielorobotowych,
- zapewnienie zgodności zadania z językiem opisu zadań, zapisanym w definicji typu dokumentu,
- zapewnienie poprawności automatu z definicją automatu skończonego,
- możliwość zapisywania i odczytywania zadań użytkowych dla robotów z pliku,
- wieloplatformowość,

- `interaktywność.`

Aby wyjść na przeciw oczekiwaniom przyszłych użytkowników, aplikacja została rozszerzona o szereg dodatkowych funkcji, wzbogacających interfejs użytkownika i ułatwiających korzystanie z edytora:

1. `Skalowanie` - proporcjonalna zmiana wielkości wszystkich obiektów (stanów i tranzcyji) znajdujących się w edytorze. Ułatwia zarządzanie zadaniem.

2. `Sprawdzanie poprawności automatu` - dzięki czemu możliwe jest sprawdzenie, czy powstający automat jest zgodny z przyjętą konwencją, a także redukcja błędów dzięki otrzymanemu raportowi.

3. `Aktywacja i dezaktywacja elementów` - powodująca, że elementy nieaktywne nie będą brane pod uwagę podczas generacji pliku XML.

4. `Wyróżnianie tranzycji aktywnego stanu` - powoduje, że tranzycje stanu aktywnego są wyróżnione poprzez zmianę ich koloru. Ułatwia to wyszukiwanie stanów docelowych poszczególnych tranzycji.

5. `Wyszukiwanie stanów` - powodujące wyszukanie i zaznaczenie stanu w liście dostępnych stanów.

6. `Edycja elementów` - umożliwia zmianę wartości atrybutów, a także listę atrybutów stanu. Możliwa jest również edycja tranzycji.

7. `Drukowanie do formatu pdf` - umożliwia zapisanie graficznej reprezentacji automatu do formatu pdf.

5 Realizacja rozwiązania

5.1 Modułowość

W trakcie implementacji rozwiązania wykorzystana została jedna z najpopularniejszych metod programistycznych, a mianowicie "Dziel i zwyciężaj"(ang. divide and conquer).

Aplikacja podzielona została na moduły, w taki sposób, aby każdy z nich stanowił pewną funkcjonalną całość, odpowiadającą za część projektu. Dzięki temu, każdy z modułów wymagał implementacji grupy ściśle powiązanych ze sobą funkcji. Dzięki temu możliwe było skupienie uwagi jedynie na aktualnie implementowanym module, co w znaczącym stopniu przyczyniło się do redukcji potencjalnych błędów. Niekiedy, szczególnie w przypadku dość złożonych modułów, istniała potrzeba dekompozycji na mniejsze części.

Po zakończeniu implementacji każdego z modułów możliwe było poddanie go testom jednostkowym, których celem było odnalezienie błędów. Pozwoliły one również, potwierdzić bądź nie słuszność podjętych wcześniej decyzji projektowych. Warto byłoby wspomnieć, że prowadzone były również testy, które miały na celu zbadać wpływ danego modułu na inne, które są z nim powiązane. Oba podejścia pozwalają zapobiec niebezpieczeństwu późnego wykrycia poważnych błędów.

Podejście to ma również znaczenie estetyczne. Pisanie aplikacji w oparciu o moduły umożliwia nie tylko prostszą pielęgnację kodu, ale również prostszą jego modyfikację.

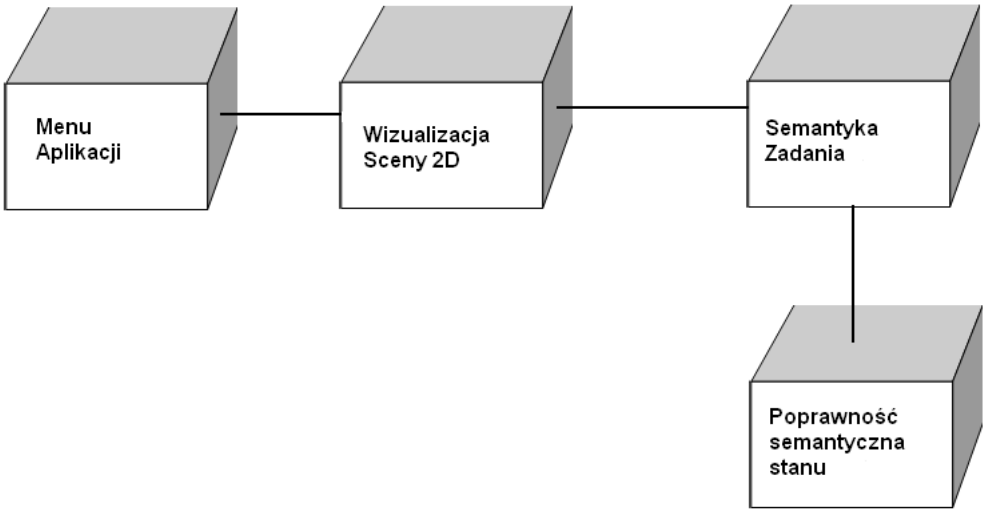
Prezentowana strategia implementacji projektu jest niezwykle popularna w zespołach programistycznych, gdzie każdy z programistów zajmuje się odrębną częścią aplikacji, a na końcu odbywa się stopniowa integracja modułów. Dodatkowo, strategia ta jest intuicyjna i w znaczącym stopniu przyśpiesza czas realizacji projektów programistycznych.

5.1.1 Schemat aplikacji

Wszystkie wymagania funkcjonalne zostały podzielone na 4 główne moduły widoczne na schemacie 7. Każdy moduł zawiera grupę ściśle powiązanych ze sobą funkcji.

Pomimo tego, że aplikacja ma budowę modułową, poszczególne moduły mogą się między sobą komunikować poprzez udostępniony w Qt mechanizm sygnałów i slotów (szerzej opisane w rozdziale 2.6.1).

Jak widać ze schematu, graficzna reprezentacja automatu jest całkowicie niezależna od właściwego zadania użytkowego, które ma zostać wykonane w systemie robotycznym MRROC++, z tego też względu reprezentacja graficzna została odseparowana od właściwej części zadania. Aplikacja została stworzona zgodnie z modelem: gui-controler-model, gdzie:



Rysunek 7: Moduły aplikacji

- `Gui` - stanowi wizualizacja automatu;
- `Kontroler` - stanowi mechanizm odpowiedzialny za komunikację gui z modelem, a także kontrolę poprawności;
- `Model` - jest to zadanie robotyczne w postaci obiektowej.

Należałoby wspomnieć, iż kluczową rolę w projekcie odgrywa kontroler, który jest odpowiedzialny za komunikację między poszczególnymi modułami, zrealizowaną poprzez wspomniany wcześniej mechanizm slotów.

Kolejność w jakiej były implementowane moduły:

- `Wizualizacja sceny 2D`
- `Poprawność semantyczna stanu`
- `Poprawność semantyczna zadania`
- `Menu Aplikacji`

Realizacja aplikacji wymagała początkowo implementacji mechanizmu umożliwiającego modelowanie automatu skończonego na scenie 2D, zwizualizowanego w postaci grafu skierowanego. Z racji tego, iż powstające środowisko aplikacyjne powinno być w pełni interaktywne, tj. powinny być udostępnione mechanizmy umożliwiające zmianę położenia elementów, jak i wartości które kryją się pod poszczególnymi elementami, konieczne było dodanie wielu elementów kontrolnych. Oparte one są na wywoływaniu poszczególnych zdarzeń w aplikacji. Mimo tego, że aplikacja została wzbogacona

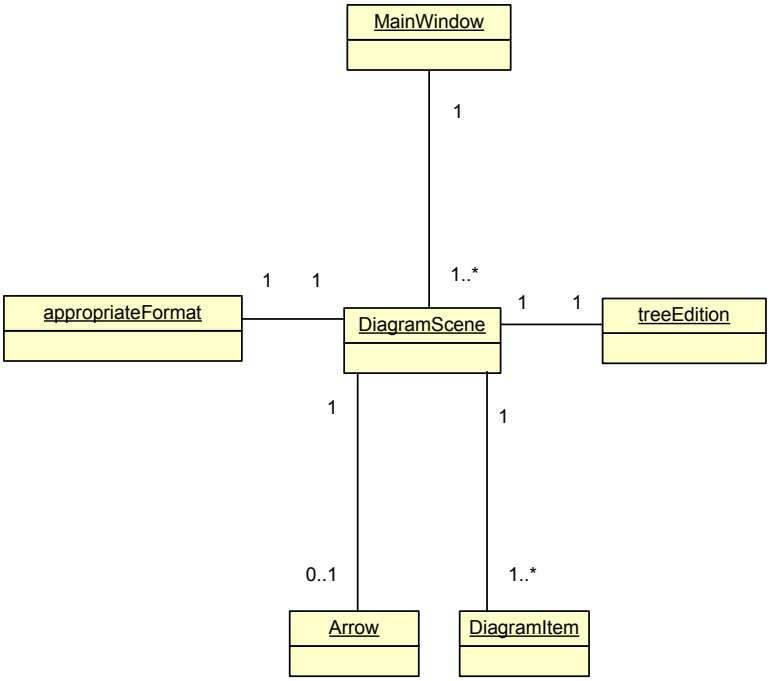
o wiele efektów graficznych, najtrudniejszy okazał się mechanizm odpowiadający za zapewnienie poprawności semantycznej stanu automatu z notacją zapisaną w pliku definicji typu dokumentu. Wymagało to zaimplementowania mechanizmu, który będzie umożliwiał dodawanie atrybutów w oparciu o wartość atrybutu `"type"` stanu. Kolejną funkcją miała być kontrola poprawności dodawanych elementów, co uniemożliwiłoby stworzenie stanu, który miałby niepoprawne wartości atrybutów. Wymagało to zaimplementowania całego mechanizmu kontroli poprawności, który powstał w oparciu o bibliotekę QProperty dostarczony przez środowisko Qt.

Aby zapewnić możliwość wczytania zadań do powstałej aplikacji niezbędne okazało się napisanie parsera XML, który rozumiałby składnię zdefiniowanego języka opisu zadań, a następnie tworzyłby odpowiednie stany i tranzycje. Konieczne było też dokonanie kilku modyfikacji w systemie odpowiadającym za wykonanie zadania po stronie systemu MRROC++. Na koniec został zaimplementowany moduł umożliwiający użytkownikowi prostą komunikację z aplikacją, a mianowicie menu. Umożliwia ono zapis oraz odczyt zadań robotycznych z/do pliku, skalowanie automatu stanów czy też kontrolę poprawności automatu. Realizacji projektu towarzyszyły liczne zabiegi optymalizacyjne. Projekt został zakończony szeregiem testów w warunkach laboratoryjnych przy użyciu symulatora zbik3d. Pozwoliły one dopracować szczegóły i w pełni dostosować aplikację do potrzeb użytkowników.

5.2 Wizualizacja sceny 2D

Jak nadmieniono we wstępnie, kluczowym celem opisywanego modułu jest stworzenie narzędzia umożliwiającego wizualizację automatu skończonego, przedstawionego w postaci grafu skierowanego, gdzie stany reprezentowane są przez węzły, natomiast tranzycie przez krawędzie grafu. Diagram klas odpowiadający za realizację wymienionych funkcji został przestawiony na rysunku 8:

- `Klasa MainWindow` - Graficzny edytor został zaimplementowany jako aplikacja języka C++, wykonana w środowisku QT, dlatego też klasa MainWindow dziedziczy po klasie QMainWindow. Jednym z zadań tej klasy jest stworzenie głównego okna aplikacji dlatego też jest ona odpowiedzialna za obsługę zdarzeń wywołanych przez komponenty wchodzące w skład tego okna. Wymusiło to implementację slotów odpowiedzialnych za wykonanie akcji związanej z poszczególnymi sygnałami. Graficzny edytor rozpoczyna swoje działanie od statycznej funkcji MainWindow::main(), która tworzy instancję obiektu klasy MainWindow. Konstruktor tej klasy tworzy obiekty reprezentujące roboty i generatory występujące po stronie edytora, następnie wywoływane są metody tworzące i wyświetlające główne okno aplikacji. Dalsze działanie programu obejmuje obsługę zdarzeń wy-



Rysunek 8: Schemat modułu wizualizacji graficznej automatu

generowanych przez użytkownika, jak i obsługę komunikatów otrzymanych z systemu.

Prezentowana klasa pełni również funkcje kontrolera, który odpowiedzialny jest za wymianę informacji między elementami gui a modelem.

- `Klasa DiagramScene` - klasa ta stanowi kontener dla elementów które zostaną wyświetlone na scenie (elementy Arrow i DiagramItem). Została ona stworzona na bazie klasy QDiagramScene znajdującej się w środowisku Qt. Komunikuje się ona z kontrolerem, poprzez udostępniony w Qt mechanizm sygnałów i slotów.

- `Klasa DiagramItem`- klasa odpowiedzialna jest za wizualizację stanu w automacie. Zawiera ona w sobie wartość unikalnego identyfikatora stanu. Stworzona jest na bazie klasy QDiagramItem znajdującej się w środowisku Qt.

- `Klasa Arrow` - klasa odpowiedzialna jest za wizualizację tranzycji od stanu początkowego do stanu docelowego. Zawiera ona w sobie wartość argumentu transition, w celu jego wyświetlenia w edytorze. Stworzona jest na bazie klasy QDiagramItem znajdującej się w środowisku Qt.

- `Klasa appropriateFormat` - klasa odpowiedzialna jest za panel w edytorze, który zawiera listę stanów automatu skończonego, wraz z ich atrybutami oraz tranzycjami. Poprzez udostępnienie menu kontekstowego, umożliwia również wykonywanie operacji, tj. usunięcie lub edycja elementów takich jak stan czy tranzycja.

5.2.1 Scena edytora

Reprezentacja graficzna sceny edytora, która odpowiedzialna jest za wizualizację grafu skierowanego reprezentującego w sposób graficzny schemat maszyny stanowej została zrealizowana przy użyciu udostępnionych w ramach platformy QT klas QGraphicsScene i QGraphicsView.

Pierwsza z nich stanowi kontener do przechowywania instancji klasy QGraphicsItem. Opisuje ona lokalizację obiektów grafiki 2D na powierzchni sceny. Opis położenia elementów oparty jest na kartezjańskim układzie współrzędnych, gdzie położenie reprezentowane jest przez współrzędne X i Y oraz transformacje odpowiadającą za zmianę lokalnego układu odniesienia obiektu wokół układu odniesienia całej sceny.

Druga z użytych klas, QGraphicsView, służy jedynie do wizualizacji sceny edytora. Udostępnia również interfejs umożliwiający zmianę układu współrzędnych, co niewątpliwie ułatwia zarządzanie reprezentacją graficzną sceny, poprzez skalowanie, bądź też zmianę pozycji elementów.

5.2.2 Reprezentacja stanu

Reprezentacja graficzna stanu została zrealizowana w oparciu o klasę QGraphicsItem zawartej w bibliotece QGraphicsView Framework. Reprezentowana jest przez koło, na którym znajduje się id stanu, któremu odpowiada graficzna reprezentacja stanu.

W trakcie realizacji projektu, została przeciążona metoda paint() w klasie QGraphicsItem, dzięki której możliwa jest modyfikacja reprezentacji graficznej stanu w przypadku wystąpienia sytuacji wyjątkowych, tj. zaznaczenie, aktywowanie lub dezaktywowanie stanu. Umożliwia ona również zmianę widocznego na stanie id oraz odmalowanie wizualizacji reprezentacji stanu po zmianie jego położenia w kartezjańskim układzie współrzędnych.

Aby wyjść na przeciw potrzebom użytkowników, zaimplementowany został mechanizm umożliwiający wykonanie szeregu funkcji, m.in. modyfikację stanu automatu. Poprzez wykorzystanie menu kontekstowego, rozwiązanie to jest nie tylko wygodne dla projektanta, ale również ogranicza czas modyfikacji automatu.

5.2.3 Reprezentacja tranzycji

Reprezentacja graficzna tranzycji została zrealizowana w oparciu o klasę QGraphicsItem zawartej w bibliotece QGraphicsView Framework. Reprezentowana jest przez strzałkę grafu skierowanego, której grot skierowany jest w kierunku stanu docelowego.

Zgdonie z definicją tranzycji w definicji maszyny stanowej, jest ona określona jako obiekt o ściśle określonym stanie początkowym i końcowym. Z tego względu w trakcie realizacji projektu zastosowano analogiczne podejście. Umożliwia to uniknięcie potencjalnych błędów, takich jak stworzenie reprezentacji graficznej tranzycji, która nie

istnieje, bądź też jest niezgodna z przyjętą definicją języka. Dodatkowo, udostępnia mechanizm dostępu do podstawowych parametrów tranzycji, tj. stan początkowy, stan końcowy, jak również warunek tranzycji, przyjmujący jedną z dostępnych wartości.

Zastosowane podejście w znaczącym stopniu upraszcza wizualizację maszyny stanowej. Dzięki powiązaniu reprezentacji tranzycji bezpośrednio z reprezentacją stanów, a nie z konkretną pozycją obietku w kartezjańskim układzie współrzędnych, możliwe jest automatyczne odmalowanie tranzycji po zmianie położenia reprezentacji graficznej stanu.

5.2.4 Reprezentacja podzadania

Reprezentacja graficzna podzadania, podobnie jak reprezentacja stanu automatu została zrealizowana w oparciu o klasę QGraphicsItem zawartą w bibliotece QGraphicsView Framework. Reprezentowana jest przez koło, na którym znajduje się nazwa podzadania, któremu odpowiada reprezentacja graficzna podzadania.

Graficzna reprezentacja podzadania zostanie umieszczona na głównej scenie edytora, natomiast wszystkie należące do tego podzadania stany i tranzycje zostaną zwizualizowane na oddzielnej scenie. Takie podejście w znaczącym stopniu upraszcza definiowanie i zarządzanie stanami w automacie, ale również umożliwia uniknięcie redundancji stanów, poprzez możliwość wielokrotnego wykorzystania tych samych podzadań.

5.2.5 Zarządzanie listą stanu automatu

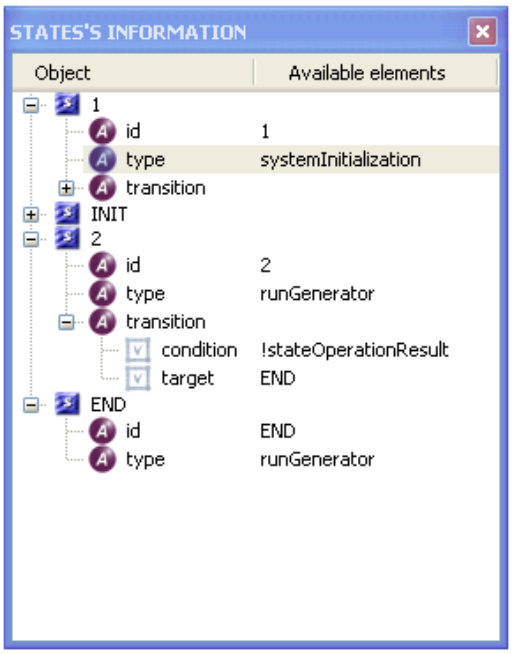
Aby zapewnić łatwe zarządzanie stanami znajdującymi się w edytorze, zaimplementowany został mechanizm pozwalający na stałą kontrolę listy znajdujących się w automacie stanów wraz z ich tranzycjami. Zapewnia on reprezentację graficzną maszyny stanowej w postaci drzewa, zawierającego aktualną listę stanów automatu, wraz z ich podstawowymi atrybutami oraz listą tranzycji. Przykładowe okno zostało zaprezentowane na rysunku 9.

5.2.6 Implementacja

Jako mechanizm komunikacji między opisanymi na diagramie 8 klasami służy opisany system sygnałów i slotów. Jeśli scena otrzyma sygnał świadczący o tym, że powinien zostać dodany element, generuje sygnał do kontrolera, którego celem jest sprawdzenie w modelu możliwości dodania danego elementu. W zależności od otrzymanych rezultatów podejmowana jest odpowiednia akcja. Szczegółowy opis użytych funkcji został przedstawiony na rysunku 10.

Szczegółowy opis użytych sygnałów i slotów po stronie klasy MainWindow został opisany poniżej:

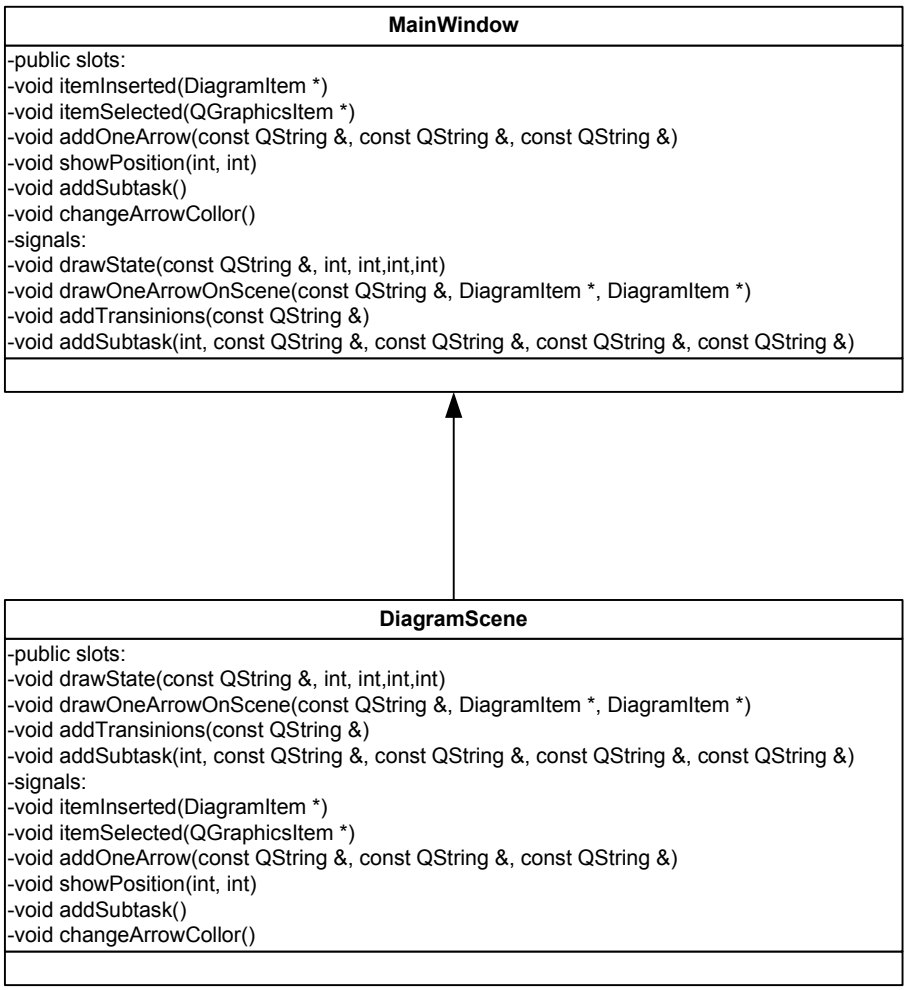
- `Slots`



Rysunek 9: Okno pokazujące stany automatu skończonego

- `itemInserted(DiagramItem*)` - funkcja ustawia na obiekcie przekazanym jako parametr funkcji flagę inserted..
- `addOneArrow(const QString, const QString, const QString)` - funkcja sprawdza, czy możliwe jest dodanie tranzycji o stanie początkowym określonym przez pierwszy argument funkcji, do stanu o id określonym przez 2 warunek funkcji.
- `showPosition(int, int)` - funkcja zapisuje aktualną pozycję kursora na panelu edytora
- `addSubtask()` - funkcja tworzy podzadanie. Nazwa tego podzadania zostaje zdefiniowana w ciele funkcji. Końcowym jej etapem jest emisja sygnału żądania wizualizacji podzadania.

- `Signals`
  - `drawState(const QString, int, int, int, int)` - sygnał żądania namalowania stanu, o id określonym przez pierwszy argument funkcji.
  - `drawOneArrowOnScene(const QString,DiagramItem*, DiagramItem*)`- sygnał przekazuje informację o konieczności umieszczenia na scenie tranzycji o warunku określonym przez pierwszy parametr funkcji, o początku i końcu opisanym odpowiednio jako drugi i trzeci parametr sygnału.
  - `addSubtask(int, const QString, const QString, const QString, const QString)` - sygnału żądania wizualizacji podzadania.



Rysunek 10: Schemat rozwiązania modułu reprezentacji graficznej automatu

Szczegółowy opis użytych sygnałów i slotów po stronie klasy DiagramScene został opisany poniżej:

- `Slots`
  - `drawState(const QString, int, int,int,int)` - funkcja maluje stan automatu na scenie
  - `drawOneArrowOnScene(const QString, DiagramItem *, DiagramItem *)` - funkcja maluje tranzycję oraz odpowiadający jej warunek przejścia
  - `addSubtask(int, const QString, const QString, const QString, const QString)` - funkcja odpowiedzialna za wizualizację podzadania o nazwie określonej przez 2 parametr wywołania fukcji.
- `Signals`
  - `itemInserted(DiagramItem*)` - sygnał przekazuje informację o elemencie, który został zaznaczony.

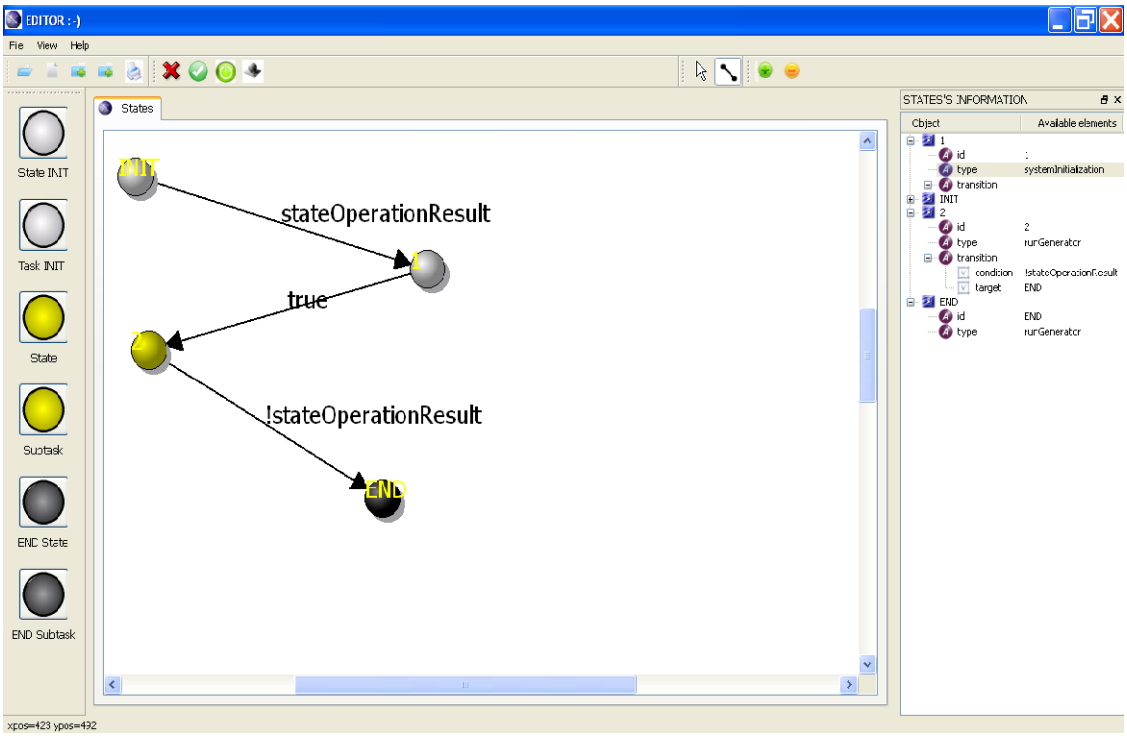
– `addOneArrow(const QString, const QString, const QString)` - sygnał przekazuje informację o chęci dodania tranzycji

– `showPosition(int, int)` - sygnał przekazuje aktualna pozycję kursora na scenie

– `addSubtask()` - sygnał przekazuje informację o chęci dodania nowego podzadania

5.2.7 Wynik pracy

Finalny wygląd aplikacji klienckiej został zaprezentowany na rysunku 11.



Rysunek 11: Okno prezentujące obecny wygląd edytora

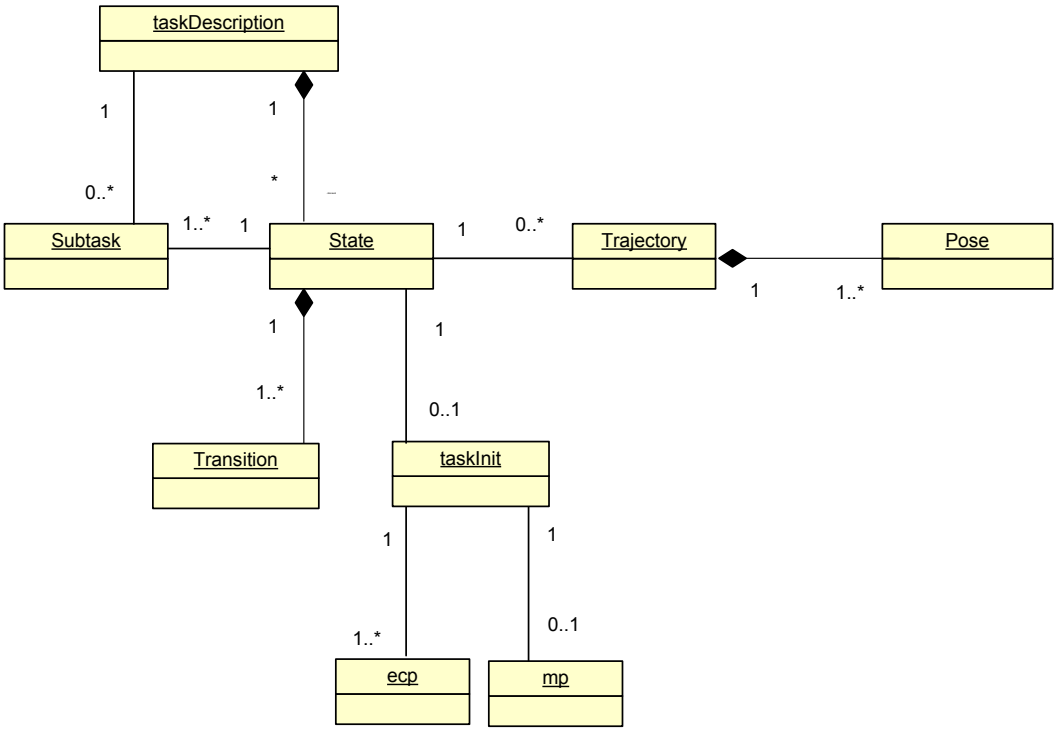
Automat reprezentowany jest przez graf skierowany, w którym stany oznaczone są przez symbol koła. Strzałki natomiast odzwierciedlają tranzycje, gdzie grot strzałki skierowany jest w kierunku stanu docelowego, stąd też przy każdej z nich znajduje się również warunek przejścia do stanu, do którego prowadzi tranzycja.

Nie od rzeczy byłoby wspomnieć, iż cechą dystynktywną wszystkich stanów jest id - unikalny identyfikator stanu. Każda wartość może pojawić co najwyżej jeden raz w obrębie danego zadania. Należałoby wspomnieć, iż aplikacja dostarcza narzędzia, umożliwiające nie tylko specyfikację prostych zadań, ale również zadań zawierających podzadania. Edycja każdego z podzadań realizowana jest w oddzielnym oknie, co znacznie ułatwia tworzenie zadania.

5.3 Poprawność semantyczna zadania

Aby zdefiniowany język opisu zadań wielorobotowych w dokumencie XML miał rację bytu w powstającym edytorze, należało zaimplementować mechanizm, który potrafiłby interpretować tak zapisane zadania, tworzyć nowe bądź modyfikować już istniejące zadania, a następnie je wizualizować. Powstałe rozwiązanie oparte jest na opisanym wcześniej mechanizmie sygnałów i slotów.

Diagram klas przedstawiony na rysunku 12 opisuje strukturę mechanizmu interpretującego zadanie po stronie edytora.



Rysunek 12: Schemat obiektowej reprezentacji zadania robotycznego

Przedstawiony powyżej obiekt klasy taskDescription odpowiedzialny jest za przetrzymywanie całego zadania robotycznego. Obiekt tej klasy reprezentuje element <TaskDescription> w systemie MRROC++. Obiekty pozostałych klas służą do reprezentowania elementów istniejących w dokumencie XML zawierającym zadanie robotyczne:

- `State` - obiekt tej klasy reprezentuje element <State> w systemie MRROC++.
- `SubTask` - obiekt tej klasy reprezentuje element <SubTask>. Jest kontenerem zawierającym obiekty klasy State.
- `Trajectory` - obiekt tej klasy reprezentuje element <Trajectory> w systemie MRROC++.
- `Pose` - obiekt tej klasy reprezentuje element <Pose>.

- `Transition` - obiekt tej klasy reprezentuje element <Transition> w systemie MRROC++,
- `taskInit` - obiekt tej klasy reprezentuje element <taskInit>.
- `ecp` - obiekt tej klasy reprezentuje element <ecp>.
- `mp` - obiekt tej klasy reprezentuje element <mp>.

Dość istotną kwestią, poruszoną w ramach niniejszej pracy inżynierskiej, było określenie kryteriów definiujących poprawność całego zadania robotycznego. Kryterium sprawdzające poprawność poszczególnych stanów i tranzycji okazało się bowiem niewystarczające, gdyż mechanizm wykonujący zadanie po stronie systemu MRROC++, nie oferował sprawdzania poprawności tranzycji wychodzących z konkretnego stanu. Pociągało to za sobą niebezpieczeństwo wystąpienia sytuacji, w której kilka warunki przejść kilku różnych tranzycji wychodzących konkretnego stanu jest jednocześnie spełnionych, co przeczy definicji automatu skończonego, na podstawie którego bazuje język opisu zadań. W związku z faktem, iż wartością atrybutu condition poszczególnych tranzycji sprawdzane były w kolejności ich występowania, mogło to doprowadzić do różnych wyników działania automatu. W związku z czym, zaistniała potrzeba modyfikacji istniejącej notacji, umożliwiająca uniezależnienie działania automatu od kolejności wystąpień poszczególnych tranzycji. W chwili obecnej, za poprawny uznaje się stan, którego struktura jest zgodna z definicją typu dokumentu, natomiast iloczyny warunków poszczególnych warunków stanowią zbiory rozłączne, przy czym suma warunków tranzycji pokrywa całą przestrzeń.

Dodatkowo, każde zadanie musi zawierać stan początkowy automatu o id=`"INIT"`, co najmniej jeden stan końcowy oraz dowolną ilość podzadań.

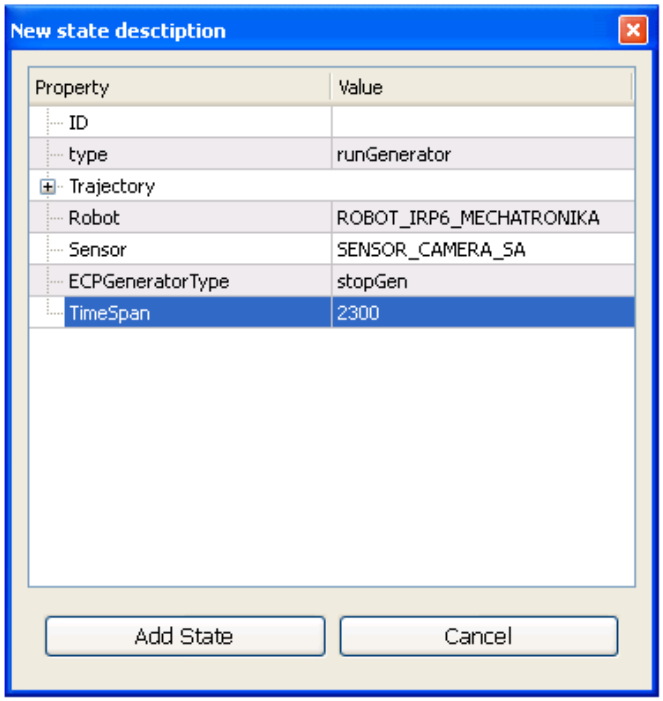
5.4 Poprawność semantyczna stanu

5.4.1 Zapewnienie zgodności z DTD

Kolejnym, stawianym wobec powstającego edytora wymaganiem było zapewnienie poprawności powstającego zadania z przyjętą definicją języka opisu zadań użytkowych. W tym celu konieczne było zdefiniowanie mechanizmu odpowiedzialnego za kontrolę poprawności stanu automatu, z definicją zawartą w pliku DTD. Z dość oczywistych względów, najlepszym rozwiązaniem wydaje się być podejście umożliwiające kontrolę poprawności już na etapie specyfikacji stanu. Dzięki temu możliwa jest stała kontrola poprawności całego automatu a także dodawanie atrybutów w zależności od typu stanu. Opisywany mechanizm został zrealizowany w oparciu o udostępnioną w QT bibliotekę QTreePropertyBrowser.

Rozwiązanie zostało zaimplementowane w sposób, który wyklucza możliwość nieprawidłowego dodania elementów. Lista dostępnych atrybutów jest dedykowana dla konkretnego typu stanu, co gwarantuje nam że atrybuty dostępne tylko w niektórych stanach (np. taskInit odpowiedzialny za inicjalizację środowiska dostępny jest tylko w stanie początkowym automatu) mogą wystąpić tylko w stanach, dla których zostały określone. Dodatkowo, sprawdzana jest poprawność typów dla poszczególnych atrybutów, co zapewnia zgodność każdego z elementów języka opisu zadań z przyjętą notacją.

Okno umożliwiające dodanie nowego stanu, wraz z dodaniem odpowiednich atrybutów, opartym na opisanym systemie kontroli poprawności zostało przedstawione na rysunku 13.



Rysunek 13: Okno prezentujące dodanie nowego stanu do edytora

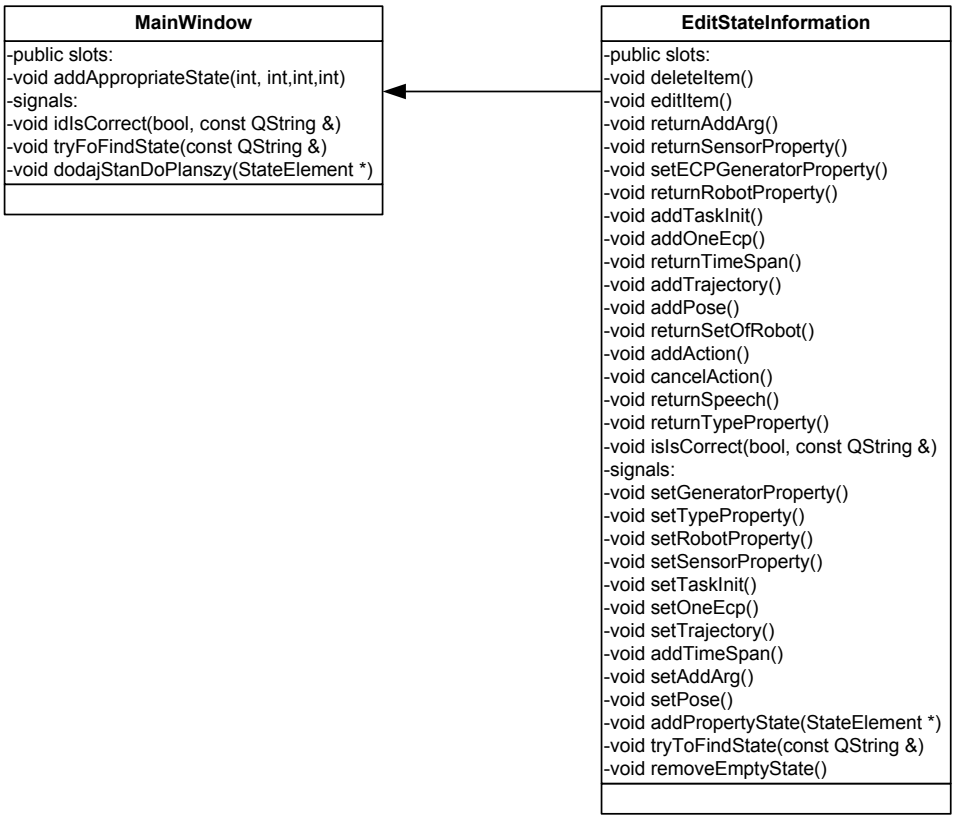
Z racji tego, iż wartości elementów takich jak type, sensor, czy ECPGeneratorType, jest określona w definicji typu dokumentu (DTD), możliwe było stworzenie dla każdego z nich listy dostępnych wartości.

5.4.2 Implementacja

Implementacja rozwiązania oparta jest na mechanizmie sygnałów i slotów, wywoływanych między klasą MainWindow a klasą EditStateInformation. Wszystkie wykorzystane funkcje zostały przedstawione na schemacie 5.4.2.

Opis sygnałów i slotów wykorzystanych po stronie klasy MainWindow:

- `Slots`



Rysunek 14: Schemat rozwiązania odpowiadającego za zapewnienie poprawności stanu

  - `void addAppropriateState(int, int, int, int)` - funkcja dodająca stan do zadania, w przypadku gdy stan był poprawny;
  - `void tryToFindState(constQString)` - funkcja sprawdza, czy w automacie istnieje stan o id równym argumentowi wywołania funkcji. Jeśli wartość ta dotychczas nie wystąpiła, przekazuje informację, że stan może zostać dodany. W przeciwnym wypadku zwraca błąd;
  - `void dodajStanDoPlanszy(StateElement *)` - funkcja odpowiedzialna jest za dodanie do automatu skończonego nowego stanu, o wartości określonej poprzez parametr funkcji;

- `Signals`
  - `void idIsCorrect(bool, const QString)` - sygnał wymuszający sprawdzenie poprawności id podanego jako parametr funkcji.

Opis sygnałów i slotów wykorzystanych po stronie klasy EditStateInformation:

- `Slots`
  - `void deleteItem()` - funkcja usuwa atrybut zaznaczony poprzez conteXMenu;

- `void editItem()` - funkcja tworzy obiekt klasy State na podstawie dodanych poprzez ContextMenu atrybutów;
- `void returnAddArg()` - funkcja tworzy obiekt QProperty typu int, definiujący atrybut AddArg;
- `void returnSensorProperty()` - funkcja tworzy obiekt QProperty definiujący element <Sensor>;
- `void setECPGeneratorProperty()` - funkcja tworzy obiekt QProperty definiujący element <ECPGeneratorType>;
- `void returnRobotPrpperty()` - funkcja tworzy obiekt QProperty definiujący element <ROBOT>;
- `void addTaskInit()` - funkcja tworzy obiekt QProperty definiujący element <taskInit>;
- `void addOneEcp()` - funkcja tworzy obiekt QProperty definiujący element <ecp>;
- `void returnTimeSpan()` - funkcja tworzy obiekt QProperty definiujący element <TimeSpan>;
- `void addTrajectory()` - funkcja tworzy obiekt QProperty definiujący element <Trajectory>;
- `void addPose()` - funkcja tworzy obiekt QProperty definiujący element <Pose>;
- `void returnSetOfRobot()` - funkcja tworzy obiekt QProperty definiujący element <SetOfRobot>;
- `void addAction()` - funkcja odpowiada akcji związanej z przyciskiem AddState;
- `void cancelAction()` - funkcja powoduje zakończenie dodawania stanu, zakończone niepowodzeniem
- `void returnSpeech()` - funkcja tworzy obiekt QProperty definiujący element <Speech>;
- `void returnTypeProperty()` - funkcja tworzy obiekt QProperty definiujący element <type>;
- `void isIdCorrect(bool, const QString)` - funkcja odpowiedzialna jest za dodanie, bądz też nie, do modelu automatu stanu stworzonego w obiekcie klasy (o id określonym poprzez drugi parametr funkcji) w zależności od wartości zmiennej typu bool otrzymanej jako pierwszy parametr funkcji.

- `Signals`

- `void setGeneratorProperty()` - sygnał wymusza dodanie atrybutu ECP-GeneratorType;
- `void setTypeProperty()` - sygnał wymusza dodanie atrybutu type;
- `void setRobotProperty()` - sygnał wymusza dodanie atrybutu ROBOT;
- `void setSensorProperty()` - sygnał wymusza dodanie atrybutu Sensor;
- `void setTaskInit()` - sygnał wymusza dodanie atrybutu taskInit;
- `void setOneEcp()` - sygnał wymusza dodanie elementu ecp do atrybutu taskInit;
- `void setTrajectory()` - sygnał wymusza dodanie atrybutu Trajectory;
- `void addTimeSpan()` - sygnał wymusza dodanie atrybutu TimeSpan;
- `void setAddArg()` - sygnał wymusza dodanie atrybutu AddArg;
- `void setPose()` - sygnał wymusza dodanie nowej pozycji do tranzycji;
- `void addPropertyState(StateElement *)` - sygnał wymusza dodanie do modelu automatu stanu przekazanego jaka parametr funkcji;
- `void tryToFindState(const QString)` - sygnał wymusza sprawdzenie, czy stan o id podanym jako parametr funkcji istnieje.

6 Podsumowanie

6.1 Przeprowadzone testy

W celu sprawdzenia poprawności działania edytora konieczne było przeprowadzenie szeregu testów. Należały do nich zarówno proste testy, które miały na celu sprawdzenie poprawności podstawowych funkcji, takich jak wizualizacja automatu skończonego w edytorze, jak i nieco bardziej skomplikowane, odpowiedzialne za sprawdzenie poprawności bardziej zaawansowanych funkcji odpowiedzialnych za poprawne wykonanie zadania stworzonego w edytorze po stronie systemu MRROC++.

Pierwsze z wyżej wymienionych testów przeprowadzane były na prostych przykładach, testujących poszczególne funkcje powstającej aplikacji. Wspomniane testy jednostkowe przygotowywane były do sprawdzenia poszczególnych funkcji udostępnianych przez aplikację.

Podstawowe funkcje odpowiedzialne za wizualizację automatu, jak również tworzenie odpowiadającej mu części semantycznej, odpowiedzialnej za zadanie testowane były na dość prostych przykładach. Testy polegały na tworzeniu, zapisywaniu do pliku, a następnie ponownym wczytywaniu zadań. Dzięki temu możliwe było określenie czy wszystkie elementy zostały utworzone poprawnie, a także czy sam schemat odpowiedzialny za wizualizację automatu został namalowany poprawnie.

Istotnym elementem w testowaniu aplikacji było sprawdzenie poprawności wykonywania zadań stworzonych przez edytor po stronie systemu MRROC++. Naturalnym więc wydawało się tworzenie tego typu zadań na podstawie już istniejących, aby możliwa było weryfikacja ich poprawności. Aby sprawdzić, czy stworzone przez edytor zadanie jest poprawne semantycznie i zgodne z przyjętą notacją, postanowiono wykorzystać już istniejące zadanie układania kostki Rubika. Wybór ten wydawał się być dość naturalny, ponieważ wykonanie zadania wymaga użycia niemal wszystkich generatorów zawartych w strukturze MRROC++. Umożliwiłoby więc sprawdzenie większości dostępnych w systemie elementów. Z powodu ciągłej ewolucji systemu, a także generatorów konieczna była modyfikacja istniejących zadań robotycznych, aby możliwe było ich uruchomienie w systemie MRROC++. Mimo tego, że zadanie to na pozór wydawało się proste, przysporzyło dość wielu problemów. Oprócz dodania atrybutów odpowiadających za wizualizację automatu konieczna było dokonanie kilku modyfikacji wynikający z ewolucji poszczególnych elementów jak i całego systemu. Zmianie uległ m.in. format w jakim zapisywane były trajektorie dla generatora Smooth. Zostały z niego usunięte dwa atrybuty, co pociągało za sobą konieczność modyfikacji tych elementów w pliku z zadaniem oraz w definicji typu dokumentu (DTD).

Kolejna modyfikacja wynikała z wprowadzenia opcjonalnej wartości trybu pracy, który dotąd zawsze miał stałą wartość (tryb ABSOLUTE). Z racji tego, iż zaistniała

potrzeba dedykowania trybu pracy dla konkretnego stanu, konieczna stała się modyfikacja mechanizmu realizującego zadanie po stronie systemu MRROC++ oraz modyfikacja języka opisu zadań.

Dodatkowo, trajektoria generowana przez generator `gen_gripper_oppening` została zastąpiona przez generator `gen_smoooth`, co pociągnęło za sobą konieczność kolejnych modyfikacji zadania. Dodatkowo, niektóre użyte w zadaniu sensory zostały usunięte, co wiązało się z koniecznością usunięcia niektórych stanów z zadania.

Dostosowane do obecnych potrzeb i możliwości systemu zadanie zostało uruchomione w trybie testowym. W celach testowych używano również symulatora robotycznego zbik3d, który wizualizował zachowanie automatu.

6.2 Wnioski

W ramach pracy powstał graficzny edytor umożliwiający definiowanie i wizualizację zadań wielorobotowych w postaci automatu skończonego, zapisywanego w postaci dokument XML. Stworzony mechanizm powstał w oparciu o istniejący język opisu zadań robotycznych stworzony w Instytucie Automatyki i Informatyki Stosowanej, w związku z czym zadanie zdefiniowane w edytorze może zostać wykonane po stronie systemu MRROC++. Wykorzystany język zawiera definicję elementów wykorzystanych do opisu zadań w omówionej notacji, a także zbiór reguł określających sposób ich konstruowania. Wszystkie reguły zostały zapisane w pliku definicji typu danych - DTD.

Dość istotnym elementem pracy okazała się konieczność modyfikacji zarówno istniejącego języka opisu zadań wielorobotowych, jak również mechanizmu ich realizacji. Realizacja edytora wymagała modyfikacji definicji typu języka. Konieczne okazało się bowiem umieszczenie w niej dodatkowych atrybutów odpowiedzialnych za reprezentację graficzną zadania. W związku z czym również zadanie zapisane w dokumencie XML musi posiadać wspomniane atrybuty. Dodatkowo konieczna okazała się modyfikacja samego algorytmu wykonującego zadania. Kolejną zmianą było rozszerzenie listy dostępnych warunków przejść dla trajektorii, poprzez modyfikację algorytmu sprawdzania warunków przejść.

Stworzone rozwiązanie spełnia wszystkie stawiane wobec niego wymagania. Edytor umożliwia tworzenie zadań użytkowych implementowanych na bazie struktury ramowej MRROC++ w postaci automatu skończonego. Udostępnia też stworzenie oraz wizualizację takiego automatu w oparciu o istniejący plik zgodny z przyjętą notacją. Mechanizm wyposażony jest w obsługę błędów, która umożliwia kontrolę poprawności zadania już na etapie jego definiowania. Posiada również wiele dodatkowych funkcji, których zadaniem jest ułatwienie specyfikacji zadań. Poprawność działania edytora została zweryfikowana poprzez różnego rodzaju testy. Największym z nich jest spraw-

dzenie poprawności zadania układania kostki Rubika. Wspomniane zadanie zostało kilkukrotnie wczytywane do edytora i zapisywane do pliku w formacie xml. Tak powstałe zadanie zostało następnie uruchomione w systemie MRROC++ przy wykorzystaniu mechanizmu interpretującego i wykonującego zdefiniowane zadanie. Wykorzystano również mechanizm porównywania plików tekstowych, który wykazał, że oba dokumenty różnią się jedynie kolejnością atrybutów, jednakże są całkowicie zgodne pod względem semantycznym.

6.3 Perspektywy rozwoju

W wyniku przeprowadzonych prac został zaprojektowany i zaimplementowany edytor graficzny umożliwiający definiowanie zadań robotycznych na podstawie automatu skończonego. Ewolucja będzie dotyczyć zarówno stanów, w miarę modyfikacji generatorów w MRROC++, jak i tranzycji, w miarę wprowadzania złożonych, wariantowych planerów. Naturalną kontynuacją rozpoczętych już prac będzie funkcjonalne dopracowanie edytora na bazie sugestii i doświadczeń użytkowników.

Literatura

[1] C. Zieliński, W. Szynkiewicz, T. Winiarski, and T. Kornuta. MRROC++ Based System Description. Technical Report 06-9, IAIS, Warsaw, 2006.

[2] C. Zieliński, T. Winiarski, W. Szynkiewicz, M. Staniak, W. Czajewski, and T. Kornuta. MRROC++ Based Controller of a Dual Arm Robot System Manipulating a Rubik's Cube. Technical Report 06-10, IAIS, Warsaw, 2006.

[3] M. Kisiel. Język opisu i realizacja zadań przez automat skończony w systemie MRROC++. Praca inżynierska, IAIS, Warszawa, 2008.

[4] P. Kazienko, K. Gwiazda. XML na poważnie. Helion, Warszawa, 2008.

[5] Xml inclusions (xinclude) version 1.0 (second edition), http://www.w3.org/tr/xinclude/.

[6] QT documentation version 4.6 http://doc.qt.nokia.com/4.6/index.html.