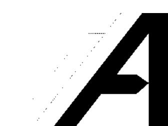
# Podręcznik programowania systemu MRROC++ dla robota IRP-6 na torze jezdnym

Cezary Zieliński, Wojciech Szynkiewicz

Instytut Automatyki i Informatyki Stosowanej
Politechnika Warszawska

Raport IAIS nr 99-21
Warszawa, Kwiecień 1999

Politechnika Warszawska
Instytut Automatyki i
Informatyki Stosowanej
ul. Nowowiejska 15/19
00–665 Warszawa
tel/fax (48) 22 – 8253719



# Spis treści

**1 Wstęp** **2**

1.1 Struktura sterownika . . . . . . . . . . 2

1.2 Proces EDP . . . . . . . . . . 7

1.3 Struktura procesu MP . . . . . . . . . . 10

1.4 Struktura procesu ECP . . . . . . . . . . 11

1.5 Program użytkowy . . . . . . . . . . 14

**2 Klasy bazowe** **15**

2.1 Klasy bazowe dla procesu MP . . . . . . . . . . 15

2.1.1 Roboty . . . . . . . . . . 15

2.1.2 Czujniki . . . . . . . . . . 17

2.1.3 Generatory . . . . . . . . . . 17

2.1.4 Warunki wstępne . . . . . . . . . . 19

2.2 Elementy dodatkowe . . . . . . . . . . 19

2.3 Obsługa sytuacji awaryjnych . . . . . . . . . . 19

2.4 Klasy bazowe dla procesu ECP . . . . . . . . . . 20

2.4.1 Robot . . . . . . . . . . 20

2.4.2 Czujniki . . . . . . . . . . 22

2.4.3 Generatory . . . . . . . . . . 24

2.4.4 Warunki wstępne . . . . . . . . . . 26

2.5 Obsługa sytuacji awaryjnych . . . . . . . . . . 26

**3 Klasy konkretne** **27**

3.1 Klasy konkretne dla procesu MP . . . . . . . . . . 27

3.1.1 Roboty . . . . . . . . . . 27

3.1.2 Generatory . . . . . . . . . . 27

3.1.3 Warunki . . . . . . . . . . 29

3.2 Klasy konkretne dla procesów ECP . . . . . . . . . . 29

3.2.1 Robot . . . . . . . . . . 29

3.2.2 Generatory . . . . . . . . . . 30
3.2.3 Warunki . . . . . . . . . . 34
3.2.4 Czujniki . . . . . . . . . . 35

**4 Funkcje i metody procesu MP** **38**

**5 Funkcje i metody procesu ECP** **62**

**6 Funkcje i metody procesu VSP** **112**

**A Słownik terminów** **116**

# Rozdział 1

# Wstęp

## 1.1 Struktura sterownika

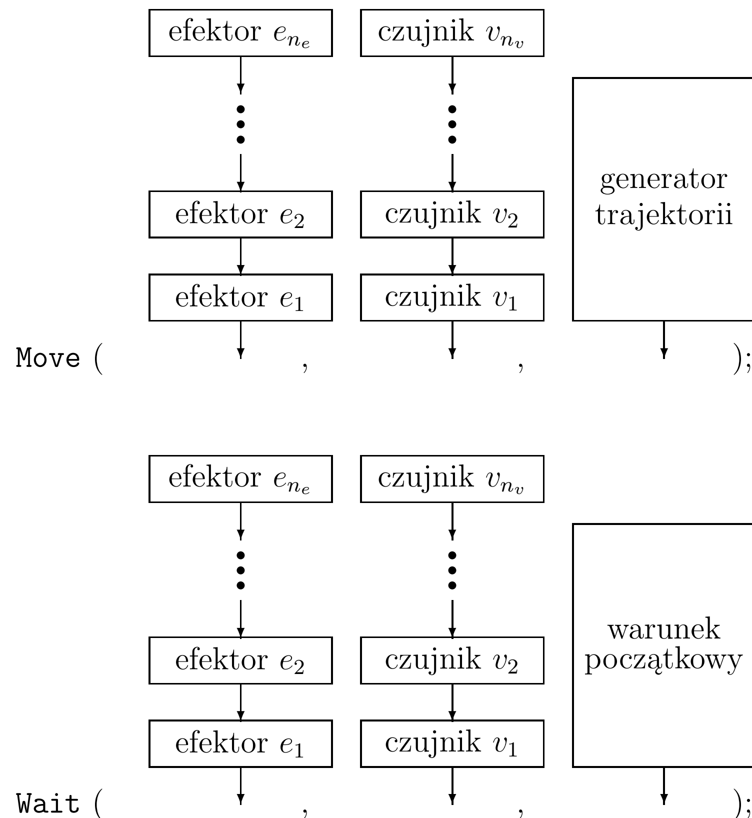
W każdym systemie robotycznym można wyróżnić trzy części:

- **efektory** – czyli te elementy systemu, które oddziałują na otoczenie, zmieniając jego stan (mogące przemieszczać jakieś obiekty) – mogą to być roboty, ale również np. podajniki lub obrabiarki,
- **receptory** – czujniki rzeczywiste (sprzętowe) zbierające informacje o stanie otoczenia, np. kamery, czujniki zbliżeniowe, dalmierze, czujniki sił i momentów sił,
- **podsystem sterujący** – składający się zarówno ze sprzętu obliczeniowego (komputera lub sieci komputerów) jak i jego oprogramowania.

Ponieważ czujniki rzeczywiste rzadko dostarczają informacji w postaci bezpośrednio przydatnej do sterowania ruchem, więc dane pomiarowe z kilku prostych czujników albo z jednego złożonego musza ulegać agregacji, czyli ekstrakci informacji użytecznej. Ta zagregowana informacja stanowi odczyt czujnika wirtualnego.

Czujniki wirtualne mogą być wykorzystane albo do monitorowania aktualnego stanu systemu i jego otoczenia albo do sterowania, czyli do wpływania na przyszłe stany tych dwóch elementów. Monitorowanie może być użyte do stwierdzenia, czy spełnione są warunki rozpoczęcia ruchu, wtedy mówimy o monitorowaniu warunku wstępnego lub początkowego. Może też być wykorzystane przy określaniu warunku zakończenia ruchu, wtedy mamy do czynienia z monitorowaniem warunku końcowego – odbywa się to w trakcie trwania ruchu. Ostatnim przypadkiem, do którego przydatne jest monitorowanie, jest wykrywanie sytuacji awaryjnych. Język `MRROC++` umożliwia użycie czujników do wszystkich trzech typów monitorowania jak i sterowania systemem. Do sterowania ruchem efektorów, przy wykorzystaniu czujników, służą instrukcje ruchowe. Można zdefiniować jedną złożoną instrukcję ruchu, która obejmie wszystkie możliwe przypadki sterowania. Niemniej jednak, dla zachowania prostoty zapisu, zdecydowano się na wprowadzenie dwóch oddzielnych instrukcji (rys. 1.1). Pierwszą jest `Move`, która odpowiedzialna jest za sterowanie ruchem oraz monitorowanie warunku końcowego. Drugą jest instrukcja `Wait`, realizująca monitorowanie warunku początkowego. Monitorowanie awarii zorganizowane zostało jako obsługa wyjątków, a więc odbywa się niejako równolegle do wykonania powyższych instrukcji, jak też dowolnych innych instrukcji programu sterującego. Obie z instrukcji korzystają z dwu list obiektów: robotów (albo ogólniej efektorów) oraz czujników wirtualnych. Ponadto instrukcja `Move` wykorzystuje obiekt zwany generatorem trajektorii. Obiekt ten określa zarówno warunek końcowy jak i sposób sterowania ruchem efektorów. Natomiast instrukcja `Wait` wymaga dostarczenia obiektu, który definiuje warunek początkowy. Oba te obiekty sta-

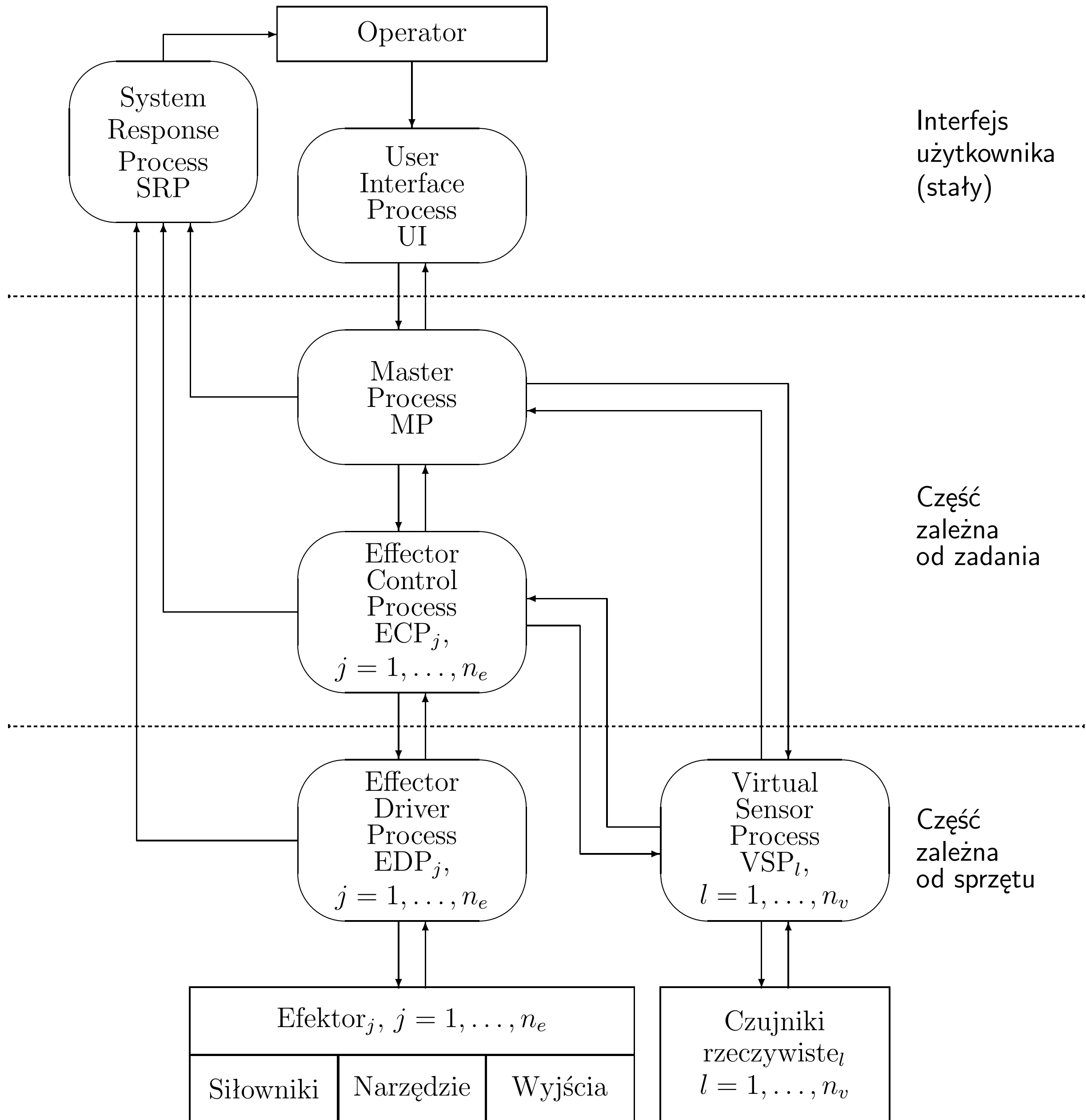
nowią porcję kodu w języku `C++` dostarczaną przez użytkownika i zależną od rodzaju zadania, które system ma wykonać.




**Rys. 1.1:** Instrukcje ruchu systemu MRROC++

Zgodnie koncepcją tworzenia układów sterujących za pomocą języka `MRROC++`, sterowniki systemów wielorobotowych mają hierarchiczną strukturę funkcjonalną. Poszczególne funkcje utworzonego sterownika realizowane są przez moduły w postaci odrębnych procesów działających w węzłach sieci lokalnej. Modularność struktury sprawia, że wymiana jednego z elementów systemu nie pociąga za sobą zmiany pozostałych. Ogólna struktura funkcjonalna tego typu sterowników przedstawiona została na rys. 1.2.

Można wyróżnić kilka warstw w układzie sterowania. Warstwa najniższa odwołuje się bezpośrednio do sprzętu. W jej skład wchodzą procesy obsługi urządzeń (ang. *drivers*) (efektorów, czujników). Procesy EDP (ang. *Effector Driver Processes*) odpowiedzialne są za rozwiązanie prostego i odwrotnego zadania kinematyki oraz bezpośredni dostęp do sprzętu, a więc za funkcje charakterystyczne dla danego typu robota, a nie za realizację zadania. Procesy VSP (ang. *Virtual Sensor Processes*) realizują odczyt i przetwarzanie danych z czujników rzeczywistych. Warstwa druga – procesy ECP (ang. *Effector Control Processes*) – realizuje algorytmy sterowania poszczególnymi efektorami adekwatnie do wykonywanego aktualnie przez system zadania. Warstwa nadrzędna – proces MP (ang. *Master Process*) – odpowiada za koordynację procesów sterujących efektorami. Warstwy, w skład których wchodzą procesy MP i ECP, zależne są jedynie od zadania, które ma być zrealizowane przez system. Program użytkowy stanowiący zapis sposobu realizacji postawionego przed systemem zadania umieszczony jest wewnątrz procesów MP i ECP. Dodatkowo istnieje jeszcze warstwa nie związana bezpośrednio z realizacją zadania – procesy UI (ang. *User Interface*) i SRP (ang. *System Response Process*) zapewniające komunikację




**Rys. 1.2:** Struktura sterownika `MRROC++` ($n_e$ – liczba efektorów, $n_v$ – liczba czujników wirtualnych)

sterownika z operatorem systemu.

Warstwa komunikacji użytkownika z rozproszonym sterownikiem wielorobotowym składa się z dwóch modułów: procesu *User Interface* obsługującego zlecenia operatora oraz procesu *System Response Process* wyświetlającego komunikaty o stanie systemu. Oba procesy realizują komunikację z użytkownikiem za pośrednictwem okienkowego środowiska graficznego QNX Windows 4.2. Użytkownik musi zadbać o zapewnienie prawidłowych mechanizmów komunikacji pomiędzy napisanym przez siebie procesem sterującym (*Master Process*) a procesem *User Interface*. Mechanizmy te pozwalają na rozpoczęcie oraz przerwanie wykonania procesu sterującego efektorami.

Proces UI obsługuje zlecenia operatora systemu i odpowiada za inicjację oraz zakończenie działania sterownika wielorobotowego. Użytkownik może sterować ręcznie robotami za pomocą okienkowego menu, poruszając się w ramach skończonej listy dostępnych zleceń. Zakres dostępnych zleceń zależny jest od bieżącego stanu systemu. Lista zleceń obejmuje: ładowanie i usuwanie procesów EDP i MP, uruchamianie i zakończenie oraz wstrzymywanie i wznawianie wykonania procesu MP, obsługa ruchów ręcznych (synchronizacja robota, ręczne ruchy na poziomie: położeń wałów silników, współrzędnych wewnętrznych oraz zewnętrznych), zakończenie działania systemu.

Proces SRP odbiera komunikaty od wszystkich procesów, które informują użytkownika o zaistnieniu szczególnej sytuacji (zmiana stanu systemu, informacja o błędzie), formatuje i wyświetla komunikaty na ekranie monitora, przechowuje w buforze nadchodzące komunikaty, tak aby można było śledzić zachowanie systemu od momentu jego uruchomienia.

Zarówno proces MP jak i każdy z procesów ECP składa się z dwu podstawowych części (rys. 1.3):

- stałej powłoki (niemodyfikowalnej części procesu) oraz
- jądra, czyli części modyfikowanej przez użytkownika.

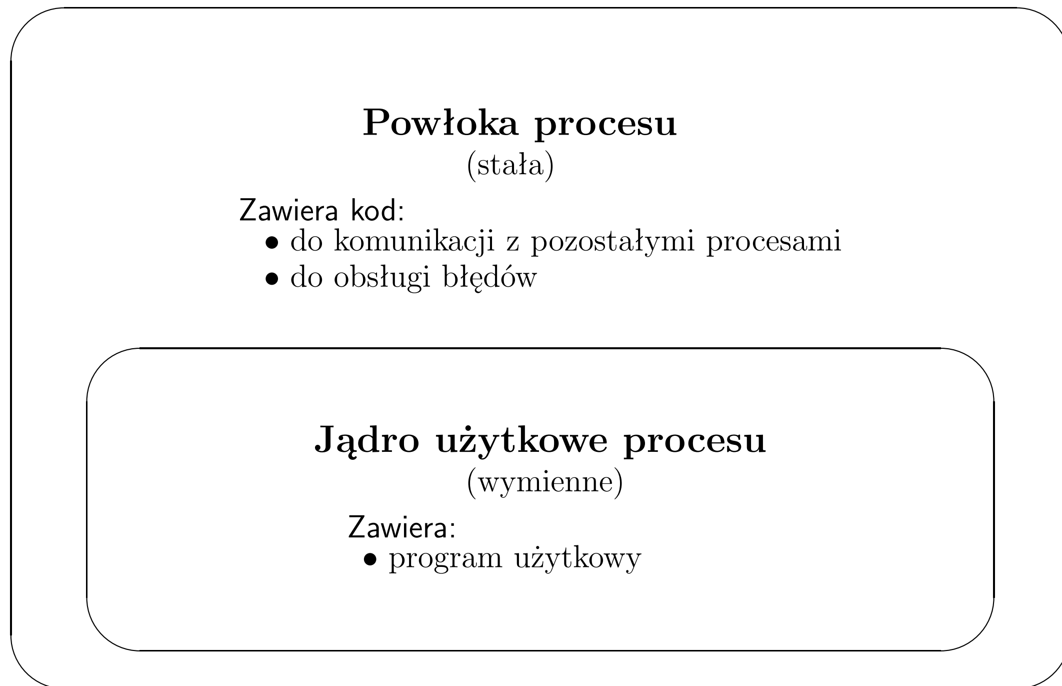
W części stałej procesu dokonuje się: rejestracja tego procesu w systemie operacyjnym, powoływanie jego procesów potomnych oraz inicjacja kanałów komunikacyjnych między nim a procesami współpracującymi. Struktura oraz funkcje jądra zależą od rodzaju oraz złożoności zadania. W zależności od rodzaju zadania i liczebności efektorów, programista umieszcza instrukcje `Move` i `Wait` oraz inne polecenia pomocnicze (instrukcje `C++`) zarówno w jądrze procesu MP jak i procesów $\mathrm{ECP}_j$ ($j = 1, \ldots, n_e$, gdzie $n_e$ jest liczbą efektorów).

Biorąc pod uwagę sposób współpracy robotów, wyróżnia się trzy grupy zadań:

- zadania, w których roboty działają całkowicie *niezależnie*,
- zadania, w których roboty *luźno* ze sobą współpracują, wymagana jest wtedy synchronizacja robotów w wybranych punktach przestrzeni,
- zadania, w których roboty *ściśle* ze sobą współpracują, wymagana jest wówczas ścisła koordynacja czasowo-przestrzenna robotów wzdłuż trajektorii ruchu.

Dla **robotów działających niezależnie**, poza fazą początkową, kiedy powoływane i uruchamiane są procesy oraz inicjowane łącza komunikacyjne między procesami, proces MP zawieszony jest w oczekiwaniu na zlecenie operatora. Procesy $\mathrm{ECP}_j$ autonomicznie sterują ruchem poszczególnych robotów. W tym przypadku instrukcje `Move` i `Wait` są umieszczane w wymiennych częściach procesów $\mathrm{ECP}_j$. Instrukcje te dotyczą pojedynczego efektora związanego z danym procesem $\mathrm{ECP}_j$.

W przypadku **luźnej współpracy robotów** proces MP pełni funkcje takie jak uprzednio, a ponadto synchronizuje procesy $\mathrm{ECP}_j$. W algorytmie realizacji zadania, w części wykonywanej przez proces MP, zapisuje się, które procesy $\mathrm{ECP}_j$ mają być synchronizowane i w których chwilach. Zgłoszenie żądania synchronizacji jest zapamiętywane i zgłaszające się procesy $\mathrm{ECP}_j$ są zawieszane aż do chwili zgłoszenia się wszystkich synchronizowanych procesów $\mathrm{ECP}_j$. Wówczas proces MP przesyła do nich zlecenia odwieszenia. Takie rozwiązanie pozwala na jednoczesną synchronizację więcej niż dwu procesów sterowania efektorami. Poza synchronizacją, proces MP może dodatkowo realizować przekazywanie wiadomości (danych) między procesami $\mathrm{ECP}_j$. W tym przypadku również, instrukcje `Move` i `Wait` są umieszczane w wymiennych częściach procesów $\mathrm{ECP}_j$. Tutaj również instrukcje te dotyczą pojedynczego efektora związanego z danym procesem $\mathrm{ECP}_j$.




**Rys. 1.3:** Struktura procesów ECP i MP.

W zadaniach wymagających **ścisłej współpracy robotów** rola procesu MP jest dominująca. Proces MP wykonuje całość programu sterującego wszystkimi efektorami. Użytkownik nie pisze własnych procesów $\mathrm{ECP}_j$, które są tylko pośrednikami między procesem MP, a opisanymi dalej procesami $\mathrm{EDP}_j$. W tym przypadku instrukcje `Move` i `Wait` umieszczane są w wymiennej części procesu MP. Instrukcje te dotyczą wszystkich efektorów związanych z procesem MP.

Procesy sterowania efektorami $\mathrm{ECP}_j$ są tworzone przez użytkownika stosownie do potrzeb zadania. Algorytm realizacji danego zadania implikuje strukturę, funkcje oraz liczbę procesów $\mathrm{ECP}_j$. Ze względu na sposób współpracy robotów (efektorów) wyróżniamy dwie podstawowe struktury procesów sterowania efektorami: strukturę dla zadań, w których roboty działają całkowicie *niezależnie* i strukturę dla zadań, w których roboty *luźno* ze sobą współpracują.

W pierwszym przypadku zakłada się, iż między efektorami nie ma żadnych interakcji, więc procesy $\mathrm{ECP}_j$ są całkowicie niezależne i realizują programy sterujące poszczególnymi efektorami. Złożoność algorytmu sterowania zależy zarówno od zadania jakie ma wykonać efektor, jak też rodzaju efektora (robot, taśmociąg, itp.).

Luźna współpraca polega na synchronizacji efektorów w wybranych punktach przestrzeni. Oznacza to synchronizację odpowiednich procesów $\mathrm{ECP}_j$ w ściśle określonych miejscach programu sterującego. Procesy $\mathrm{ECP}_j$ nie synchronizują się między sobą, lecz poprzez MP, który pełni rolę koordynatora. Proces MP wysyła potwierdzenie dopiero po zgłoszeniu się wszystkich procesów sterujących, które w danym miejscu programu winny być zsynchronizowane. Po odebraniu odpowiedzi realizowane są kolejne instrukcje programu, aż do następnego miejsca synchronizacji. Wybierając odpowiednie miejsca w programie steru-

jącym, w których konieczna jest synchronizacja poszczególnych $\mathrm{ECP}_j$, możemy realizować fizyczną koordynację działania efektorów.

Proces $\mathrm{ECP}_i$ zgłaszający żądanie nie musi „wiedzieć”, z jakimi $\mathrm{ECP}_j$, $i \neq j$, będzie synchronizowany. To, jakie procesy $\mathrm{ECP}_j$ synchronizowane są w danej chwili wynika z algorytmu wykonania zadania. Poza synchronizacją, możliwa jest wymiana, za pośrednictwem koordynatora, wiadomości między procesami $\mathrm{ECP}_j$.

Proces $\mathrm{EDP}_j$ jest odpowiedzialny za zarządzanie pracą pojedynczego efektora (robota). Po uruchomieniu, proces ten oczekuje na zlecenia od innych procesów (klientów) systemu. Proces $\mathrm{EDP}_j$ spełnia rolę interpretera poleceń przysyłanych do niego przez proces $\mathrm{ECP}_j$. Sam proces $\mathrm{EDP}_j$ ma najczęściej strukturę złożoną i składa się z wielu podprocesów.

Komunikacja procesu $\mathrm{ECP}_j$ z procesami $\mathrm{VSP}_l$, których on używa, może być zrealizowana jako przekaz:

- interaktywny lub
- nieinteraktywny.

W przypadku komunikacji interaktywnej $\mathrm{ECP}_j$ wysyła żądania danych do swoich $\mathrm{VSP}_l$. Procesy $\mathrm{VSP}_l$ odczytują adekwatne czujniki rzeczywiste, dokonują agregacji danych i przesyłają uzyskany w ten sposób odczyt czujnika wirtualnego z powrotem do $\mathrm{ECP}_j$. W międzyczasie $\mathrm{ECP}_j$ może sterować efektorem. Jeżeli odczyty nie nadejdą do chwili gdy $\mathrm{ECP}_j$ wykona operację odczytu danych, jest on zawieszany do chwili ich nadejścia.

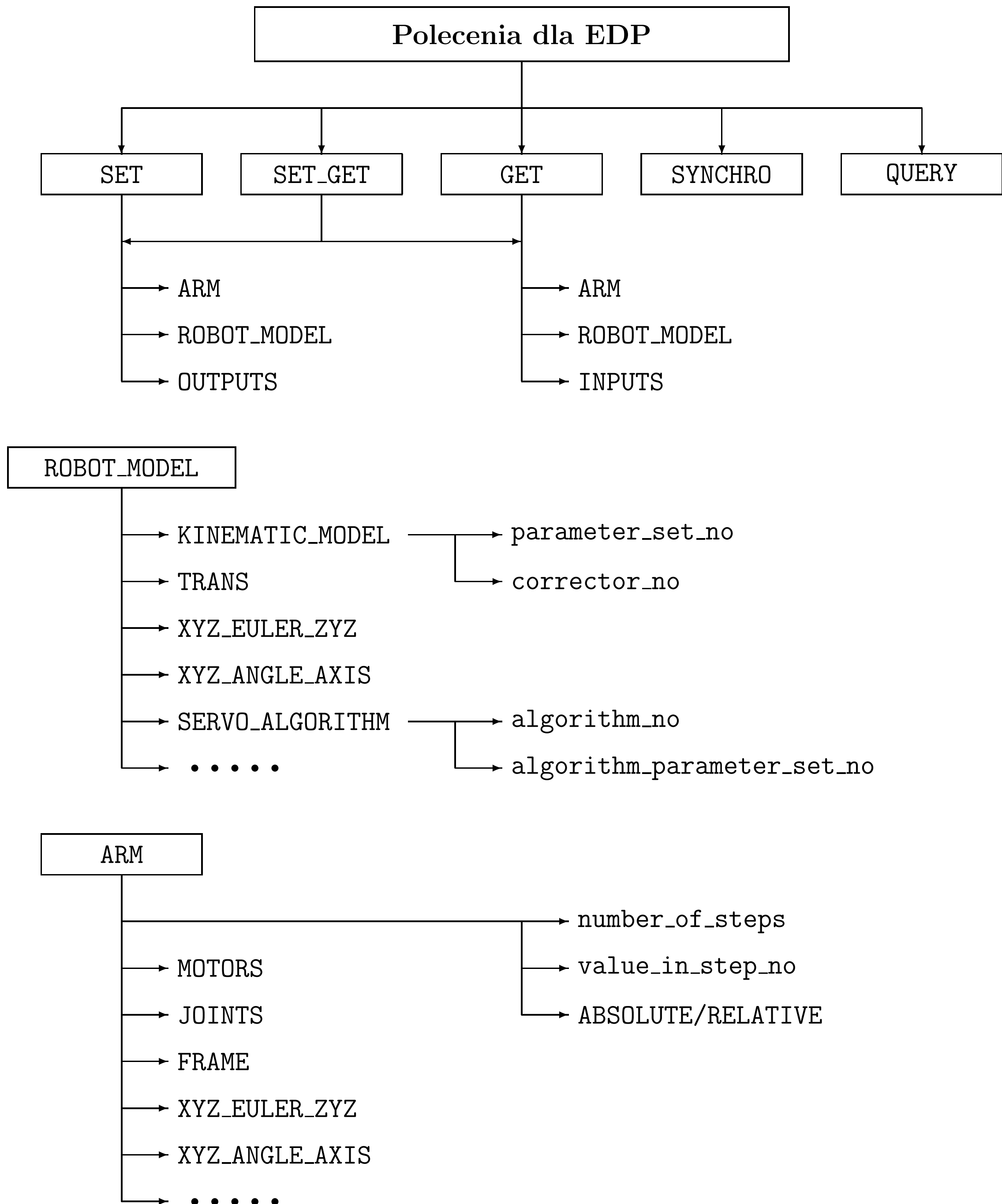
W przypadku komunikacji nieinteraktywnej $\mathrm{VSP}_l$ wyzwalany jest przerwaniami. Gdy czasomierz wygeneruje przerwanie, $\mathrm{VSP}_l$ odczytuje adekwatne czujniki rzeczywiste, dokonuje agregacji danych i otrzymany wynik wstawia do bufora. Następnie proces $\mathrm{VSP}_l$ jest zawieszany. W ten sposób każdy z procesów $\mathrm{ECP}_j$ może odczytać ostatnie dane w każdej chwili. Oczywiście dostęp procesów do bufora musi być synchronizowany (wzajemne wykluczanie).

Dla architektury sprzętowej składającej się z lokalnej sieci komputerów klasy PC, jednym z najlepiej dostosowanych do potrzeb implementacji złożonych struktur sterowania jest rozproszony system wielozadaniowy czasu rzeczywistego QNX 4.2x. System ten składa się z egzekutora wielozadaniowego i zespołu zadań (procesów) systemowych, współpracujących ze sobą i zadaniami użytkowymi zgodnie z modelem wymiany usług (ang. *client-server model*). Odwołania zadań użytkowych do zadań systemowych mają postać wiadomości, przekazujących podczas *spotkania* żądanie wykonania określonego zlecenia. Możliwość realizacji spotkania między dowolnymi zadaniami w sieci sprawia, że każde zadanie może korzystać z dowolnych zasobów całej sieci lokalnej. Modularna struktura i brak prywatnych połączeń między zadaniami użytkowymi umożliwiają elastyczne konfigurowanie systemu stosownie do potrzeb poszczególnych aplikacji.

Każde dołączenie robota nowego typu do systemu pociąga za sobą stworzenie dla niego procesu $\mathrm{EDP}_j$. Ogólna struktura tego procesu została zdefiniowana w systemie `MRROC++`.

## 1.2 Proces EDP

Proces $\mathrm{EDP}_j$ spełnia rolę interpretera poleceń przysyłanych z $\mathrm{ECP}_j$ lub UI. Ma on strukturę automatu skończonego, określającą jakie polecenia są legalne w danym stanie tego procesu EDP. Listę poleceń dla procesu EDP, gdy efektorem jest robot, przedstawiono na rys. 1.4.




**Rys. 1.4:** Lista poleceń dla EDP

Komunikacja z procesem EDP (*driver'em* robota) zawsze składa się z dwóch faz. W fazie pierwszej informuje się go co ma być zrobione. Wtedy przesyłany jest jeden z rozkazów: `SYNCHRO`, `SET`, `GET`, `SET_GET` wraz z adekwatnymi parametrami. W fazie drugiej proces EDP jest pytany o rezultat wykonania uprzednio wysłanego rozkazu za pomocą polecenia `QUERY`. Dopiero wtedy EDP przyśle dane, o które był pytany lub jeśli nie żądano żadnych danych, potwierdzi wykonanie polecenia albo poinformuje o zaistniałym błędzie. Proces EDP zawsze zachowuje się biernie — stanowi *server* dla *klienta* (procesów ECP i UI).

Po uruchomieniu procesu EDP można mu wysłać jedno z dwóch poleceń: `SYNCHRO` lub

`SET`. Przed synchronizacją robota rozkaz ruchu czyli zlecenie `SET ARM` dostępne jest tylko w jednej postaci. Można jedynie zlecać ruch względny zadawany w postaci przyrostów położenia wałów silników (`SET ARM MOTORS RELATIVE`). Inne formy rozkazu `SET ARM` są niedopuszczalne w tym stanie i powodują błąd. Spowodowane jest to przyrostowym pomiarem położenia wałów silników, a więc koniecznością inicjacji liczników położenia po włączeniu sterownika. Wysłanie rozkazu `SYNCHRO` spowoduje przemieszczenie wszystkich osi robota do położenia czujników synchronizacji. Synchronizacja wykonywana jest tylko na początku działania systemu. Może być ewentualnie wykonywana powtórnie po wystąpieniu błędu fatalnego, który powoduje utratę synchronizacji, wówczas proces EDP wraca do stanu początkowego, takiego jaki jest tuż po uruchomiemiu i inicjacji procesu EDP. Informacja o sposobie zakończenia synchronizacji (bezbłędne lub z błędem) przekazywana jest procesowi ECP po wydaniu przez niego polecenia `QUERY`. Jeśli robot jest zsynchronizowany, powtórny rozkaz `SYNCHRO` nie będzie wykonany zostanie jedynie przesłany komunikat z błędem (`ALREADY_SYNCHRONISED`).

Po bezbłędnym wykonaniu synchronizacji, do dyspozycji procesu ECP są tylko rozkazy `SET`, `GET`, `SET_GET` (we wszystkich wariantach) – w pierwszej fazie komunikacji z EDP; oraz rozkaz `QUERY` – w drugiej fazie. Polecenie `SET` może dotyczyć:

- zmiany parametrów geometrycznych narzędzia aktualnie zamontowanego na manipulatorze,
- ustawienia wyjść binarnych układu sterującego,
- zmiany zbioru parametrów modelu kinematycznego robota,
- zmiany korektora kinematycznego,
- zmiany algorytmów regulacji,
- zmiany parametrów (nastaw) serwomechanizmów osi,
- przemieszczenie ramienia robota.

Można spowodować każdą z tych czynności osobno, wszystkie razem albo dowolną ich kombinację, w zależności od użycia parametrów rozkazu `SET`.

Polecenie `GET` powoduje:

- odczytanie parametrów geometrycznych narzędzia aktualnie zamontowanego na manipulatorze,
- odczytanie stanu wejść binarnych układu sterującego,
- odczytanie aktualnego numeru algorytmu regulacji,
- odczytanie numeru aktualnego zbioru parametrów serwomechanizmów,
- odczytanie aktualnego położenia ramienia robota.

Można spowodować każdą z tych czynności osobno, wszystkie razem albo dowolną ich kombinację, w zależności od użycia parametrów rozkazu `GET`.

Polecenie `SET_GET` jest superpozycją poleceń `SET` i `GET`. Przykładowo za jego pomocą można zlecić ustawienie wyjść, odczytanie wejść, przemieszczenie ramienia oraz odczytanie jego położenia po zakończeniu ruchu. Oczywiście dane zwrotne zostaną przekazane dopiero po wydaniu polecenia `QUERY`. Położenie i orientacja narzędzia (`TOOL`) względem kołnierza manipulatora mogą być określone na różne sposoby m.in.:

- jako macierz reprezentująca trójścian związany z końcówką,
- jako trzy współrzędne kartezjańskie oraz trzy kąty Eulera (ZYZ) trójścianu związanego z końcówką,
- jako trzy współrzędne kartezjańskie oraz wektor stanowiący oś obrotu trójścianu związanego z końcówką (długość tego wektora jest proporcjonalna do kąta obrotu).

Położenie i orientacja narzędzia zamocowanego na ramieniu robota (`ARM`) względem globalnego układu odniesienia mogą być określone w następujący sposób:

- jako macierz reprezentująca trójścian związany z końcówką,
- jako trzy współrzędne kartezjańskie oraz trzy kąty Eulera (ZYZ) trójścianu związanego z końcówką,
- jako trzy współrzędne kartezjańskie oraz wektor stanowiący oś obrotu trójścianu związanego z końcówką (długość tego wektora jest proporcjonalna do kąta obrotu).

Ponadto położenie ramienia (`ARM`) może być określane poprzez podanie współrzędnych wewnętrznych łańcucha kinematycznego lub przez zadanie położeń wałów silników elektrycznych.

Odczyt położenia robota wykonywany jest jedynie we współrzędnych bezwzględnych, natomiast ruch może być zrealizowany we współrzędnych bezwzględnych lub względnych (przyrostowo w stosunku do aktualnej pozycji).

Każde polecenie ruchu (`SET ARM`) traktowane jest jako makrokrok. Dodatkowym parametrem polecenia jest liczba kroków, w której ma być wykonany makrokrok. Ponieważ odczyt ostatniego przyrostu położenia wałów silników odbywa się równocześnie z zadaniem następnego przyrostu położenia wałów silników do wykonania, to aby uzyskać nieprzerwany ruch składający się z wielu makrokroków należy zażądać odczytu wcześniej niż ruch się skończy. Do tego celu służy parametr `value_in_step_no`. Określa on, w którym kroku realizacji makrokroku ma być dokonany odczyt położenia, a następnie wysłany do wyższych warstw sterownika. Jeśli wartość tego parametru jest o jeden większa od zadanej liczby kroków, to odczyt będzie przesłany po zakończeniu ruchu robota. Nie należy jednakże zakładać, iż jest to już położenie końcowe. Najczęściej jest jeszcze pewien uchyb położenia, który regulator będzie starał się wyzerować w następnych kilku kolejnych krokach regulacji.

Przesłanie polecenia `GET` procesowi EDP, w celu uzyskania ostatniego odczytu położenia ramienia, spowoduje odczytanie aktualnych położeń wałów silników i dokonanie stosownych przeliczeń.

## 1.3 Struktura procesu MP

Jak już wspomniano, Master Process składa się z dwóch zasadniczych części (rys. 1.3):

- powłoki (niezmiennej części procesu – niezależnej od wykonywanego zadania),
- jądra (wymiennej częś procesu – zależnej od programu użytkowego realizującego zadanie zlecone systemowi przez użytkownika).

Zadaniem powłoki jest:

- kontakt z operatorem (nasłuch jego poleceń),
- powoływanie, a po zakończeniu pracy likwidacja procesów ECP i VSP,
- utworzenie łączy komunikacyjnych z innymi procesami,
- obsługa zgłoszonych wyjątków, czyli reakcja na zaistniałe błędy,
- sygnalizacja operatorowi sytuacji awaryjnych (przekazanie procesowi SRP informacji o zaistniałej sytuacji),
- realizacja programu użytkowego zawartego w jądrze dostarczanym przez programistę.

Proces MP rozpoczyna swe działanie rejestracji w w systemie operacyjnym `QNX`. Kolejnym krokiem jest utworzenie połączeń komunikacyjnych z UI oraz utworzenie pośredników do komunikacji z UI. Potem tworzone są procesy ECP sterujące efektorami (robotami) wchodzącymi w skład systemu. Następnie MP przechodzi w stan oczekiwania na polecenie *START* od UI. Po otrzymaniu tego polecenia MP rozpoczyna wykonanie programu użytkowego. Po jego zakończeniu oczekuje na polecenie *STOP* od UI. Otrzymanie tego polecenia powoduje przejście do stanu początkowego.

Nasłuch poleceń operatora odbywa się poprzez przyjmowanie sygnałów (`SIGTERM`) oraz wiadomości od pośredników (`proxy`). W ten sposób proces MP informowany jest o takich poleceniach jak: *START*, *STOP*, *PAUSE*, *RESUME*. Zadaniem procesu MP jest adekwatna reakcja na te polecenia, a więc przekazanie ich do innych części systemu, w szczególności do procesów ECP.

Jeżeli w trakcie działania procesu MP zostanie wykryty błąd, to zostanie zgłoszony wyjątek, który zostanie obsłużony na najwyższym poziomie tego procesu, a wiec w funkcji `main`. Ten sam mechanizm sygnalizacji błędów – wyjątki – stosowany jest zarówno przez stała powłokę jak i wymienne jądro. W ten sposób użytkownik (programista) w swoim kodzie (programie użytkowym) zgłasza wyjątki, tam gdzie antycypuje powstanie sytuacji awaryjnych, natomiast nie musi sie zajmować ich obsługą – to zapewnia powłoka automatycznie.

Struktura jądra procesu MP przedstawiona jest na rys. 1.5. Jądro procesu MP zawiera:

- deklaracje obiektów, które będą używane w programie użytkowym,
- program użytkowy.

Wspomnianymi obiektami są:

- obiekty klasy wywiedzionej z abstrakcyjnej klasy bazowej `robot` (efektor),
- obiekty klasy wywiedzionej z abstrakcyjnej klasy bazowej `sensor` (czujnik),
- obiekty klasy wywiedzionej z abstrakcyjnej klasy bazowej `generator`,
- obiekty klasy wywiedzionej z abstrakcyjnej klasy bazowej `condition`.

Bufory komunikacyjne przedstawione na rys. 1.5 stanowią części składowe klas wywiedzionych z klas bazowych `robot` i `sensor`. Ponadto w jądrze tworzone są listy, które następnie będą stanowiły argumenty instrukcji `Move` i `Wait`.

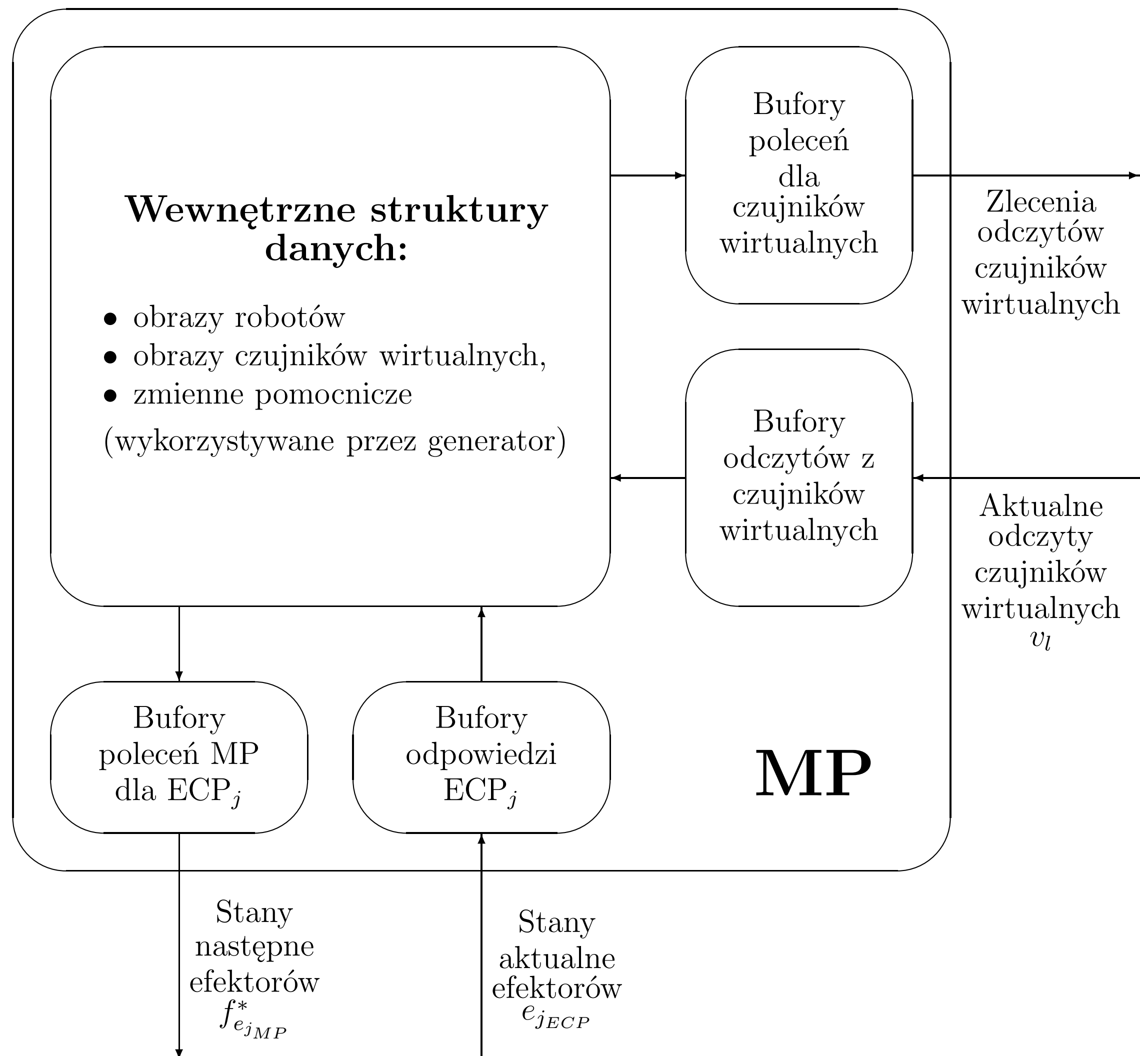
## 1.4 Struktura procesu ECP

Podobnie jak w przypadku procesu MP, Effector Control Process składa się z dwóch części (rys. 1.3):

- powłoki (niezmiennej części procesu – niezależnej od wykonywanego zadania),
- jądra (wymiennej części procesu – zależnej od programu użytkowego realizującego zadanie zlecone systemowi przez użytkownika).

Zadaniem powłoki jest:

- kontakt z procesem MP (nasłuch jego poleceń),
- powoływanie, a po zakończeniu pracy likwidacja procesów VSP współdziałających z tym procesem ECP,
- utworzenie łączy komunikacyjnych z innymi procesami,
- obsługa zgłoszonych wyjątków, czyli reakcja na zaistniałe błędy,




**Rys. 1.5:** Wewnętrzna struktura jądra procesu MP.

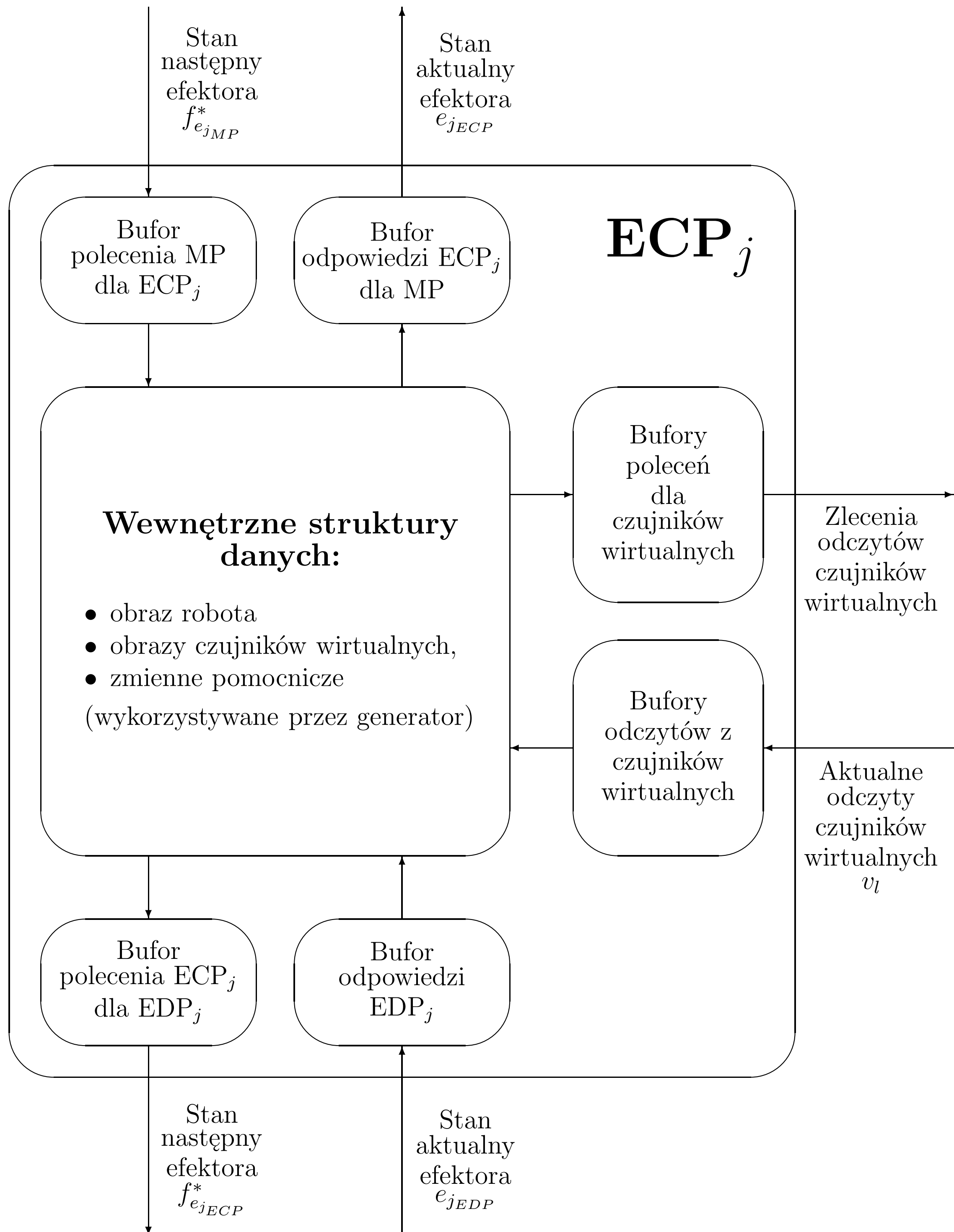
- sygnalizacja procesowi MP sytuacji awaryjnych, a ponadto przekazanie procesowi SRP informacji o zaistniałej sytuacji,
- realizacja programu użytkowego zawartego w jądrze dostarczanym przez programistę.

Proces ECP rozpoczyna swe działanie od własnej rejestracji (pod określoną nazwą globalną) w systemie operacyjnym `QNX` oraz lokalizacji procesów: MP, EDP_MASTER (stanowiącego główna, a czasami jedyną, część procesu EDP), SRP i UI. Następnie tworzy łącza komunikacyjne z tymi procesami. Po wykonaniu tych czynności przechodzi do realizacji programu użytkowego.

Jeżeli w trakcie działania procesu ECP zostanie wykryty błąd, to zostanie zgłoszony wyjątek, który zostanie obsłużony na najwyższym poziomie tego procesu, a wiec w funkcji `main`. Ten sam mechanizm sygnalizacji błędów – wyjątki – stosowany jest zarówno przez stałą powłokę jak i wymienne jądro. W ten sposób użytkownik (programista) w swoim kodzie (programie użytkowym) zgłasza wyjątki, tam gdzie antycypuje powstanie sytuacji awaryjnych, natomiast nie musi sie zajmować ich obsługą – to zapewnia powłoka

automatycznie.

Do stworzenia procesu ECP (Efector Control Process) potrzebne są podobne elementy do tych, które używano do napisania programu użytkowego wchodzącego w skład procesu MP. Struktura jądra procesu ECP przedstawiona jest na rys. 1.6.




**Rys. 1.6:** Wewnętrzna struktura jądra procesu ECP.

Jądro procesu ECP zawiera:

- deklaracje obiektów, które będą używane w programie użytkowym,
- program użytkowy.

Wspomnianymi obiektami są:

- obiekt klasy wywiedzionej z abstrakcyjnej klasy bazowej `robot` (efektor),
- obiekty klasy wywiedzionej z abstrakcyjnej klasy bazowej `sensor` (czujnik),
- obiekty klasy wywiedzionej z abstrakcyjnej klasy bazowej `generator`,
- obiekty klasy wywiedzionej z abstrakcyjnej klasy bazowej `condition`.

Bufory komunikacyjne przedstawione na rys. 1.6 stanowią części składowe klas wywiedzionych z klas bazowych `robot` i `sensor`. Ponadto w jądrze tworzona jest lista czujników, która następnie będzie stanowiła argument instrukcji `Move` i `Wait`.

## 1.5 Program użytkowy

Program użytkowy jest napisany w `C++` z użyciem funkcji bibliotecznych systemu `MRROC++`. Realizuje zadanie zlecone systemowi do wykonania przez użytkownika. Program użytkowy stanowi jądra procesów MP i ECP.

Najczęściej program użytkowy skonstruowany jest jako pętla (`for`), w której umieszczone są instrukcje (funkcje) `Move` i `Wait` oraz dodatkowe instrukcje języka `C++`.

W procesie MP rozpoczyna on swe działanie od wysłania zlecenia wszczęcia wykonania do wszystkich podległych mu procesów ECP (`start_all`) oraz powołania do życia obiektów (w sensie `C++`) klas wywiedzionych z `robot` oraz `sensor`. Następnie tworzy listę lub listy robotów i czujników. Ponadto tworzy obiekty klas wywiedzionych z klasy `generator` i `condition`.

Program użytkowy procesu ECP rozpoczyna swe działanie od powołania do życia pojedynczego obiektu (w sensie `C++`) klasy wywiedzionej z `robot` oraz obiektów klas wywiedzionych z klasy `sensor`. Następnie tworzy listę lub listy czujników. Ponadto tworzy obiekty klas wywiedzionych z klasy `generator` i `condition`. Po zakończeniu fazy inicjacji przechodzi do oczekiwania na polecenie wszczęcia działania (`START_TASK`) od procesu MP.

# Rozdział 2

# Klasy bazowe

## 2.1 Klasy bazowe dla procesu MP

Dla procesu MP zdefiniowano następujące abstrakcyjne klasy bazowe:

- `robot` (efektor),
- `sensor` (czujnik),
- `generator`,
- `condition` (warunek).

### 2.1.1 Roboty

**Klasa robot** jest bazową klasą abstrakcyjną (w sensie `C++`), z której wywodzi się klasy pochodne reprezentujące roboty określonych typów. Obiekty klas pochodnych klasy `robot` reprezentują w procesie MP roboty, czyli urządzenia techniczne. Każdorazowo, gdy do systemu wprowadzany jest nowy typ robota trzeba stworzyć odpowiadającą mu klasę wywiedzioną z `robot`. Następnie należy powołać do życia obiekt tej klasy.

Klasa `robot` w swej części prywatnej zawiera dwa bufory na pakiety komunikacyjne z procesem ECP sterującym odpowiednim robotem. Jeden z buforów służy do przechowywania poleceń MP dla ECP, natomiast drugi do zbierania odpowiedzi ECP dla MP. Programista nie korzysta z tych skomplikowanych struktur danych. Wszelkie polecenia i odpowiedzi robota przetwarzane są przez generatory trajektorii. Do tego celu używany jest **obraz robota** – struktura zawarta w klasie `robot`. Jest to przejrzysta struktura danych, w której programista piszący generator umieszcza polecenia dla ECP wraz z parametrami oraz z której odczytuje stan robota. Wszystkie składowe w tej strukturze rozpoczynające się od przedrostka `current` dotyczą ostatnio odczytanego położenia ramienia, natomiast te zaczynające się przedrostkiem `next` zawierają pozycję zadaną. Oczywiście tylko jedna z wielu możliwych reprezentacji pozycji, zarówno odczytanej jak i zadanej, jest w danym momencie wykorzystywana i aktualna. Jest to ta reprezentacja, z której w danej chwili korzysta generator trajektorii. Klasa `robot` ma dwie metody służące do kontaktu między obrazem robota a procesem ECP – są to: `create_command` i `get_reply`. Za pomocą metody `create_command`, na podstawie danych umieszczonych w obrazie robota, wypełniany jest bufor przeznaczony do wysłania pakietu komunikacyjnego do ECP. Natomiast za pomocą metody `get_reply` wypełniany jest obraz robota danymi zawartymi w pakiecie komunikacyjnym przysłanym z ECP. Struktura tych pakietów jest bardzo zawiła, gdyż dla zmniejszenia liczby przesyłanych danych zastosowano wielokrotne złożenia `unii`.

Należy więc unikać bezpośredniego korzystania z tych struktur danych.

Pozostałymi metodami klasy `robot` są:

- `execute_motion` — zleca wykonanie polecenia zawartego w pakiecie komunikacyjnym procesowi ECP,
- `start_ecp` — zleca rozpoczęcie działania procesowi ECP,
- `terminate_ecp` — zleca zakończenie działania procesowi ECP.

Klasa `robot` ma następującą postać:

```
class robot {
   // Klasa bazowa dla robotów (klasa abstrakcyjna)
   // Każdy robot konkretny (wyprowadzony z klasy bazowej)
   // musi zawierać pola danych (składowe) dotyczące
   // ostatnio zrealizowanej pozycji oraz pozycji zadanej
protected:
  MP_COMMAND_PACKAGE mp_command;  // Bufor z rozkazem dla ECP
                                  // - uzytkownik nie powinien z tego korzystac
  ECP_REPLY_PACKAGE ecp_reply;    // Bufor z odpowiedzia z ECP
                                  // - uzytkownik nie powinien z tego korzystac

public:
  pid_t ECP_pid;
  char  ECP_name[80];                 // Nazwa pliku z kodem wykonywalnym ECP
  robot_ECP_transmission_data ecp_td; // Obraz robota wykorzystywany przez generator
                                      // - do użytku użytkownika (generatora)

  robot (char* name) { strcpy(ECP_name, name); }  // konstruktor

  class MP_error {  // Klasa obsługi błędow robotów
    public:
      unsigned word32 error_class;   //
      unsigned word32 mp_error;      //
      MP_error (unsigned word32 err0, unsigned word32 err1)
                   { error_class = err0; mp_error = err1;}
  }; //end: class MP_error

  virtual void execute_motion (void) = 0;
    // Zlecenie wykonania ruchu przez robota
    // (realizowane przez klasę konkretną):
    // na poziomie MP jest to polecenie dla ECP.

  virtual void terminate_ecp (void) = 0;
    // Zlecenie zakończenia wykonania programu użytkowego w ECP
    // (realizowane przez klasę konkretną):
    // na poziomie MP jest to polecenie dla ECP.

  virtual void start_ecp ( void ) = 0;
    // Zlecenie rozpoczęcia wykonania programu użytkowego w ECP
    // (realizowane przez klasę konkretną):
    // na poziomie MP jest to polecenie dla ECP.

  virtual void create_command ( void ) = 0;
    // wypełnia bufor wysyłkowy do EDP na podstawie danych zawartych w obrazie robota
    // Ten bufor znajduje się w robocie

  virtual void get_reply ( void ) = 0;
    // pobiera z pakietu przesłanego z ECP informacje
```

```
    // i wstawia je do obrazu robota

}; // end: class robot
```

### 2.1.2 Czujniki

Abstrakcyjna klasa bazowa dla czujników ma następującą postać:

```
class sensor {
   // Klasa bazowa dla czujników (klasa abstrakcyjna)
   // Czujniki konkretne wyprowadzane są z klasy bazowej
  public:

    virtual void initiate_reading (void) = 0;
       // żądanie odczytu od VSP
       // (realizowane przez klasę konkretną)

    virtual void get_reading (void) = 0;
       // odebranie odczytu od VSP
       // (realizowane przez klasę konkretną)

}; // end: class sensor
```

### 2.1.3 Generatory

Zdefiniowano bazową klasę abstrakcyjną `generator`. Wszystkie generatory wykorzystywane przez instrukcję `Move` muszą być wywiedzione z tej klasy. Ma ona następującą postać:

```
class generator {
    // Klasa bazowa dla generatorów trajektorii (klasa abstrakcyjna)
    // Służy zarówno do wyznaczania następnej wartości zadanej jak i
    // sprawdzania spełnienia warunku końcowego

public:
  virtual ~generator(){ };  // destruktor

  virtual BOOLEAN first_step (list<sensor>* sensor_list, list<robot>* robot_list) = 0;
      // generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)

  virtual BOOLEAN next_step (list<sensor>* sensor_list, list<robot>* robot_list) = 0;
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)

  virtual void copy_data(list<robot>* robot_list) = 0;
     // Kopiuje dane z bufora zawierającego pakiet komunikacyjny otrzymany z ECP
     // do obrazów robotów

  virtual void copy_generator_command (list<robot>* robot_list) = 0;
     // Kopiuje polecenie stworzone przez generator, a przechowywane w obrazach robotów,
     // do buforów zawierających pakiety komunikacyjne przesyłane do ECP

  class MP_error {  // Klasa obsługi błędow generatora na poziomie MP
```

```
    public:
      unsigned word32 error_class;
      unsigned word32 mp_error;
      MP_error (unsigned word32 err0, unsigned word32 err1)
                   { error_class = err0; mp_error = err1;}
  }; //end: class MP_error

}; // end: class generator
```

Jest oczywiste, że nie można tworzyć obiektów klas bazowych bo są to klasy abstrakcyjne. Dopiero obiekty klas pochodnych (generatory konkretne) mogą być wykorzystane w programie użytkowym. Metody generatora konkretnego, na podstawie informacji zawartej w liście czujników i robotów, sprawdzą warunek końcowy, a jeżeli nie będzie on spełniony, to wygenerują nowe wartości zadane, które zapisze w wewnętrznych strukturach danych robotów konkretnych (obrazach robotów) umieszczonych na liście. Każdy robot konkretny musi zawierać informacje zarówno o położeniu aktualnym jak i tym, które ma być osiągnięte.

**Warunek końcowy** określa, czy wykonanie instrukcji osiągnęło stan końcowy – wtedy ruch jest przerywany, czy też, przy nie spełnieniu tego warunku, ruch należy kontynuować. Spełnienie warunku końcowego powoduje zwrócenie wartości 0 przez `next_step`, w przeciwnym przypadku, tzn. kontynuacji ruchu, zwracane jest 1. Warunek końcowy powinien być funkcją prywatną konkretnego generatora.

Poleceniami MP dla ECP są:

- `NEXT_POSE` – zlecające wykonanie kolejnego ruchu (jego efekt zależy od generatora na poziomie ECP),
- `START_TASK` – rozpoczynające wykonanie programu użytkowego w ECP,
- `END_MOTION` – kończące wykonanie ruchu w ECP,
- `STOP` – kończące wykonanie programu użytkowego w ECP na polecenie operatora.

Dla robotów pracujących niezależnie jedynym zadaniem generatora na poziomie MP jest pobudzanie do działania generatorów na poziomie ECP. Robi on to poprzez sformowanie, a następnie wysyłanie przez instrukcję `Move`, rozkazu `NEXT_POSE` do ECP. Dla robotów luźno współpracujących dodatkowo proces MP może synchronizować wkonanie procesów ECP. Program użytkowy w ECP może być kończony na polecenie MP zleceniem `STOP` albo z własnej iniciatywy, wtedy ECP przysyła do MP `TASK_TERMINATED`. Oczywiście wybór wariantu zależy od programisty piszącego program użytkowy. Po obu stronach, tzn. ECP i MP, musi być przyjęty ten sam wariant komunikacji.

W przypadku ścisłej współpracy robotów, poza pobudzaniem do działania generatorów na poziomie ECP, generator na poziomie MP ma obowiązek obliczania kolejnych wartości zadanych dla wszystkich robotów biorących udział w ruchu. Wtedy generatory na poziomie ECP ograniczają się jedynie do przekazywania odebranych od MP przesyłek do procesów EDP.

Stworzenie nowego generatora sprowadza się do napisania dwóch jego metod `first_step` oraz `next_step`. Zazwyczaj obliczenia dla pierwszego kroku realizacji trajektorii ruchu różnią się nieco od tych dla każdego następnego, dlatego wprowadzono dwie różne metody. Obie metody mogą zlecać dowolne polecenia do realizacji przez ECP, a w konsekwencji przez EDP. Są to zazwyczaj zlecenia ruchu, ale mogą to również być polecenia: odczytania aktualnego położenia ramienia w dowolnej reprezentacji, zmiany parametrów lub korektora modelu kinematycznego, zmiany stosowanych algorytmów regulacji lub przełączania

wyjść i odczytu wejść binarnych.

### 2.1.4 Warunki wstępne

Klasą obiektów odpowiedzialnych za przechowywanie i obliczanie warunków wstępnych jest `condition`. Klasa `condition` ma następującą postać:

```
class condition {
   // Klasa bazowa dla warunków oczekiwania (klasa abstrakcyjna)
   // W ramach sprawdzania spełnienia warunku początkowego
   // moze służyc zarówno do odczytania aktualnego położenia robotów
   // jak i określenia reakcji operatora.
   // Spełnienie warunku początkowego kończy oczekiwanie (Wait).
   // Stanowi ono warunek początkowy, którego spełnienie umożliwia
   // rozpoczęcie wykonania następnej instrukcji Move.

 public:

  virtual BOOLEAN condition_value (list<robot>* robot_list,
                                   list<sensor>* sensor_list) = 0;
     // bada wartosc warunku poczatkowego
     // (zadanie realizowane przez klasę konkretną)
     // TRUE - konczy czekanie (funkcja wait)
     // FALSE - kontynuuje oczekiwanie

  class MP_error {  // Klasa obslugi bledow
    public:
      unsigned word32 error_class;
      unsigned word32 mp_error;
      MP_error (unsigned word32 err0, unsigned word32 err1)
                   { error_class = err0; mp_error = err1;}
  }; //end: class MP_error

};// end: class condition
```

## 2.2 Elementy dodatkowe

Do prawidłowej organizacji współpracy procesów ECP i MP przydatne są następujące procedury:

- do zakończenia programów użytkowych we wszystkich procesach ECP – `terminate_all`
- do rozpoczęcia programów użytkowych we wszystkich procesach ECP – `start_all`
- do oczekiwania na polecenie `START` od operatora – `wait_for_start`
- do oczekiwania na polecenie `STOP` od operatora – `wait_for_stop`

## 2.3 Obsługa sytuacji awaryjnych

Wystąpienie błędu sygnalizuje się instrukcją `C++ throw`. Wyjątek przechwytywany i analizowany jest przez funkcję `main` procesu MP. Istnieją trzy klasy obsługi błędów: `MP_main_error`, `robot::MP_error`, `generator::MP_error`. Dwie ostatnie służą do sygnalizacji błędów powstałych w metodach klas wywiedzionych z `robot` i `generator`, natomiast

pierwsza do sygnalizacji błędów powstałych poza nimi. Definicja nowego błędu wymaga następujących czynności:

- wprowadzenia do kodu programu użytkowego instrukcji `throw` z argumentem będącym obiektem jednej z powyższych klas,
- rozszerzenia listy warunków instrukcji `switch` w instrukcji `catch` w funkcji `main`, jeżeli nie jest to błąd z grupy `ECP_ERRORS` lub `MP_SYSTEM_ERROR`,

Fakt wystąpienia błędu zostanie zasygnalizowany procesowi SRP, który umieści na ekranie monitora, w odpowiednim okienku, komunikat o rodzaju błędu.

## 2.4 Klasy bazowe dla procesu ECP

Dla procesów ECP zdefiniowano następujące abstrakcyjne klasy bazowe:

- `robot` (efektor),
- `sensor` (czujnik),
- `generator`,
- `condition` (warunek).

### 2.4.1 Robot

**Klasa robot** jest bazową klasą abstrakcyjną (w sensie `C++`), z której wywodzi się klasy pochodne reprezentujące roboty określonych typów. Obiekt jednej z klas pochodnych klasy `robot` reprezentuje w procesie ECP robota, czyli urządzenia techniczne, dla którego w tym procesie zapisano treść programu użytkowego. Każdorazowo, gdy do systemu wprowadzany jest nowy typ robota trzeba stworzyć odpowiadającą mu klasę wywiedzioną z `robot`. Następnie należy powołać do życia obiekt tej klasy. W każdym procesie ECP istnieje tylko jeden obiekt klasy robot.

Klasa `robot` w swej części `protected` zawiera dwa bufory na pakiety komunikacyjne z procesem EDP sterującym odpowiednim robotem (`EDP_command_and_reply_buffer`) oraz dwa bufory do kontaktu z procesem MP (`mp_command` i `ecp_reply`). Jeden z buforów służy do przechowywania poleceń ECP dla EDP, natomiast drugi do zbierania odpowiedzi EDP dla ECP. Programista nie korzysta z tych skomplikowanych struktur danych. Wszelkie polecenia i odpowiedzi robota przetwarzane są przez generatory trajektorii. Do tego celu używany jest **obraz robota**. Jest to przejrzysta struktura danych, w której programista piszący generator umieszcza polecenia dla EDP wraz z parametrami oraz z której odczytuje stan robota. Wszystkie składowe w tej strukturze rozpoczynające się od przedrostka `current` dotyczą ostatnio odczytanego położenia ramienia, natomiast te zaczynające się prefiksem `next` zawierają pozycję zadaną. Oczywiście tylko jedna z wielu możliwych reprezentacji pozycji, zarówno odczytanej jak i zadanej, są w danym momencie wykorzystywana i aktualna. Jest to ta reprezentacja, z której w danej chwili korzysta generator trajektorii. Klasa `robot` posiada dwie metody służące do kontaktu między obrazem robota a procesem EDP – są to: `create_command` i `get_reply`. Za pomocą metody `create_command`, na podstawie danych umieszczonych w obrazie robota (`EDP_data`), wypełniany jest bufor przeznaczony do wysłania pakietu komunikacyjnego do ECP. Natomiast za pomocą metody `get_reply` wypełniany jest obraz robota danymi zawartymi w pakiecie komunikacyjnym przysłanym z ECP. Struktura tych pakietów jest wielce zawiła, gdyż dla zmniejszenia liczby przesyłanych bajtów zastosowano wielokrotne złożenia `unii`. Należy

więc unikać bezpośredniego korzystania z tych struktur danych.

Pozostałymi metodami klasy `robot` są:

- `execute_motion` — zleca wykonanie polecenia zawartego w pakiecie komunikacyjnym procesowi EDP,
- `synchronise` — zleca procesowi EDP synchronizację robota,
- `terminate_ecp` — zleca zakończenie działania procesowi ECP
- `ecp_termination_notice` — informuje MP o zakończeniu zadania użytkownika
- `ecp_wait_for_start` — oczekuje na polecenie `START_TASK` od MP
- `ecp_wait_for_stop` — oczekuje na polecenie `STOP` od MP
- `get_mp_command` — oczekuje na polecenie od MP
- `mp_command_type` — bada typ polecenia z MP
- `set_ecp_reply` — ustawia typ odpowiedzi z ECP do MP
- `copy_mp_to_edp_buffer` — kopiuje bufor przesyłany z MP do bufora wysyłanego do EDP

Klasa `robot` ma następującą postać:

```
class robot {
   // Klasa bazowa dla robotów (klasa abstrakcyjna)
   // Każdy robot konkretny (wyprowadzony z klasy bazowej)
   // musi zawierać pola danych (składowe) dotyczące
   // ostatnio zrealizowanej pozycji oraz pozycji zadanej

   word08 EDP_name[30];  // Nazwa procesu EDP
   pid_t EDP_MASTER_Pid; // Identyfikator procesu driver'a edp_m
 protected:
   // Funkcja generator.next_step() przygotowuje rozkazy dla EDP wypełniając
   // strukture EDP_command_and_reply_buffer.instruction, ktora jest
   // następnie wysyłana przez funkcję execute_motion() do EDP.
   // Struktura EDP_command_and_reply_buffer.reply_package zawierająca
   // odpowiedź EDP na wysłany rozkaz.
   ecp_buffer EDP_command_and_reply_buffer;
   MP_COMMAND_PACKAGE mp_command; // Polecenie od MP
   ECP_REPLY_PACKAGE ecp_reply;   // Odpowiedz ECP do MP

 public:
   robot_EDP_transmission_data EDP_data; // Obraz robota wykorzystywany przez
                                         // generator

  virtual void execute_motion (void) = 0;
    // Zlecenie wykonania ruchu przez robota
    // (realizowane przez klasę konkretną):
    // na poziomie ECP jest to polecenie dla EDP

  robot(word08 *edp_name);    // konstruktor

  virtual void set_edp_master_pid ( pid_t ) = 0;
      // Przekazanie identyfikatora procesu EDP_MASTER

  virtual void set_mp_pid ( pid_t ) = 0;
      // Przekazanie identyfikatora procesu MP

  virtual void synchronise ( void ) = 0;
      // Zlecenie synchronizacji robota. Pobranie aktualnych polozen.

  virtual void create_command (void) = 0;
```

```
     // wypełnia bufor wysyłkowy do EDP na podstawie danych zawartych
     // w obrazie robota wykorzystywanych przez generator
     // Ten bufor znajduje sie w robocie

  virtual void get_reply (void)  = 0;
    // pobiera z pakietu przesłanego z EDP informacje i wstawia je do
    // odpowiednich składowych obrazu robota wykorzystywanych przez generator
    // Ten bufor znajduje sie w robocie

  virtual void ecp_termination_notice (void) = 0;
    // Informacja dla MP o zakonczeniu zadania uzytkownika

  virtual BOOLEAN ecp_wait_for_start (void) = 0;
    // Oczekiwanie na polecenie START od MP

  virtual void ecp_wait_for_stop (void) = 0;
    // Oczekiwanie na STOP

  virtual void get_mp_command (void) = 0;
    // Oczekiwanie na polecenie od MP

  virtual MP_COMMAND mp_command_type (void) = 0;
    // Badanie typu polecenia z MP

  virtual void set_ecp_reply ( ECP_REPLY ecp_r) = 0;
    // Ustawienie typu odpowiedzi z ECP do MP

  virtual void copy_mp_to_edp_buffer (void) = 0;
    // Kopiowanie bufora przesylanego z MP do bufora wysylanego do EDP

  class ECP_error {  // Klasa obslugi bledow robota
    public:
      unsigned word32 error_class;
      unsigned word32 error_no;
      edp_error error;

       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no)
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = 0; error.error1 = 0; };
       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no,
                   unsigned long int err0, unsigned long int err1 )
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = err0; error.error1 =err1; };
  }; //end: class ECP_error

}; // end: class robot
```

### 2.4.2 Czujniki

**Klasa sensor** jest bazową klasą abstrakcyjną, z której wywodzi się klasy konkretne. Obiekt klasy pochodnej reprezentuje w procesie ECP konkretny czujnik virtualny, z którego dane są wykorzystywane w instrukcjach sterujących (`Move`, `Wait`) tego procesu. Obiekt ten zawiera zarówno obraz czujnika wirtualnego po stronie procesu ECP, jak i bufory przesyłowe do komunikacji z procesem VSP.

Abstrakcyjna klasa bazowa ma, dla reprezentacji czujników wirtualnych po stronie procesu ECP, następującą postać:

```
class sensor {
   // Klasa bazowa dla czujnikow (klasa abstrakcyjna)
   // Czujniki konkretne wyprowadzane sa z klasy bazowej
 public:
   ECP_VSP_MSG msg_ecp_vsp;   // Wiadomosc przesylana pomiedzy ECP - VSP
   VSP_ECP_MSG msg_vsp_ecp;   // Wiadomosc przesylana pomiedzy VSP - ECP
   sensor_image image;        // Obraz czujnika (uzywany przez uzytkownika)

    virtual void initiate_reading (void) = 0;
       // zadanie odczytu od VSP
       // (realizowane przez klase konkretna)

    virtual void get_reading (void) = 0;
       // odebranie odczytu od VSP
       // (realizowane przez klase konkretna)

    virtual void create_sensor_command(void) = 0;
       // przepisuje rozkaz dla VSP z obrazu czujnika do bufora

    virtual void get_sensor_reply(void) = 0;
       // przepisuje odpowiedz VSP z bufora do obrazu czujnika

   virtual void terminate(void) = 0;
     // rozkaz zakonczenia procesu VSP

   class ECP_error {  // Klasa obslugi bledow czujnikow
     public:
       unsigned word32 error_class;
       unsigned word32 error_no;
       edp_error error;

       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no)
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = 0; error.error1 = 0; };
       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no,
                   unsigned long int err0, unsigned long int err1 )
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = err0; error.error1 =err1; };
   }; //end: class ECP_error
}; // end: class sensor
```

Każdorazowo, gdy do systemu wprowadzane są nowe czujniki, należy stworzyć odpowiednią klasę wywiedzona z klasy bazowej `sensor`. Klasa `sensor` ma w części `public` dwa bufory na pakiet komunikacyjne oraz definicje obrazu czujnika typu `sensor_image`. Jeden z buforów – typu `ECP_VSP_MSG` przeznaczony jest do przesyłania informacji do VSP, drugi – typu `VSP_ECP_MSG` przeznaczony jest do przesyłania odczytów czujników i innych informacji przez VSP. Programista, nie korzysta bezpośrednio z tych buforów, korzysta on z **obrazu czujnika** (czujników). Tutaj, po stronie ECP, należy umieścić w odpowiednich polach polecenia dla VSP.

Klasa `sensor` zawiera pięć metod. Metoda `initiate_reading` przewidziana jest do inicjacji pracy czujnika i może być wykorzystywana przy takich specyficznych czujnikach, gdzie potrzebna jest ich incjacja przez żądaniem odczytu (też: przy nieinteraktywnej pracy z czujnikami).

Metoda `get_reading` przesyła bufor `msg_ecp_vsp` do VSP i odbiera przesyłkę `msg_vsp_ecp`.

Metoda `create_sensor_command` wypełnia bufor wysyłkowy `msg_ecp_vsp` wysyłany do VSP zgodnie z informacją zawartą w polu kodu instrukcji obrazu czujnika przechowywanego w ECP: `image.instruction_code` .

Metoda `get_sensor_reply` przetwarza bufor `msg_vsp_ecp` otrzymany z VSP. W zależności od informacji o poprawności odczytu i od typu danych wypełniane są odpowiednie pola obrazu czujnika.

Metoda `terminate` służy do przygotowania i przesłania rozkazu zakończenia pracy czujnika (czujników).

### 2.4.3 Generatory

Ostatnim argumentem procedury `Move` jest obiekt stanowiący generator trajektorii. Użytkownik musi wskazać taki generator dla każdego ruchu. Oczywiście ten sam generator może być wykorzystywany w wielu ruchach. Zdefiniowano bazową klasę abstrakcyjną `generator`. Wszystkie generatory wykorzystywane przez instrukcję `Move` muszą być wywiedzione z tej klasy. Ma ona następującą postać:

```
class generator {
    // Klasa bazowa dla generatorów trajektorii (klasa abstrakcyjna)
    // Służy zarówno do wyznaczania następnej wartości zadanej jak i
    // sprawdzania spełnienia warunku końcowego
 public:
   class ECP_error {  // Klasa obslugi bledow generatora
     public:
       unsigned word32 error_class;
       unsigned word32 error_no;
       edp_error error;

       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no)
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = 0; error.error1 = 0; };
       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no,
                   unsigned long int err0, unsigned long int err1 )
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = err0; error.error1 =err1; };
   }; //end: class ECP_error

  virtual BOOLEAN first_step (list<sensor>* sensor_list, robot& the_robot) = 0;
      // generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)

  virtual BOOLEAN next_step (list<sensor>* sensor_list, robot& the_robot) = 0;
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)

}; // end: class generator
```

Jedynie obiekty klas pochodnych (generatory konkretne) mogą być użyte w programie użytkowym. Metody generatora konkretnego, na podstawie informacji zawartej w liście czujników oraz obiekcie robot, sprawdzą warunek końcowy, a jeżeli nie będzie on spełniony,

to wygenerują nowe wartości zadane, które zapiszą w wewnętrznych strukturach danych robota (obrazie robota). Obraz robota zawiera informacje zarówno o położeniu aktualnym jak i tym, które ma być osiągnięte.

**Warunek końcowy** określa, czy wykonanie instrukcji osiągnęło stan końcowy – wtedy ruch jest przerywany, czy też, przy nie spełnieniu tego warunku, ruch należy kontynuować. Spełnienie warunku końcowego powoduje zwrócenie wartości 0 przez `next_step`, w przeciwnym przypadku, tzn. kontynuacji ruchu, zwracane jest 1. Warunek końcowy powinien być funkcją prywatną konkretnego generatora.

Poleceniami MP dla ECP są:

- `NEXT_POSE` – zlecające wykonanie kolejnego ruchu (jego efekt zależy od generatora na poziomie ECP),
- `START_TASK` – rozpoczynające wykonanie programu użytkowego w ECP,
- `END_MOTION` – kończące wykonanie ruchu w ECP,
- `STOP` – kończące wykonanie programu użytkowego w ECP na polecenie operatora.

Dla ściśle współpracujących robotów jedynym zadaniem generatora na poziomie ECP jest przekazywanie poleceń MP do EDP. W przypadku robotów działających niezależnie lub luźno współpracujących generator na poziomie ECP musi samodzielnie określić trajektorię ruchu. Robi on to poprzez sformowanie, a następnie wysyłanie przez instrukcję `Move`, rozkazu do EDP. Program użytkowy w ECP może być kończony na polecenie MP zleceniem `STOP` albo z własnej iniciatywy, wtedy ECP przysyła do MP `TASK_TERMINATED`. Oczywiście wybór wariantu zależy od programisty piszącego program użytkowy. Po obu stronach, tzn. ECP i MP, musi być przyjęty ten sam wariant komunikacji.

| **Współpraca** | **Informacja** ($c_0$) | |
|---|---|---|
| luźna | decyzyjna | (booleowska) |
| ścisła | obliczeniowa | (liczbowa) |

**Tabela 2.1:** Wpływ rodzajów współpracy robotów na generatory

| **Zadanie: \ Proces:** | MP | ECP |
|---|---|---|
| Niezależne działanie robotów | inicjuje zadanie | generuje trajektorię |
| Luźna współpraca robotów | synchronizuje roboty | generuje segmenty trajektorii |
| Ścisła współpraca robotów | generuje trajektorię | przeźroczysty |

**Tabela 2.2:** Funkcje generatorów w zależności od realizowanego zadania.

Stworzenie nowego generatora sprowadza się do napisania dwóch jego metod `first_step` oraz `next_step`. Zazwyczaj obliczenia dla pierwszego kroku realizacji trajektorii ruchu różnią się nieco od tych dla każdego następnego, dlatego wprowadzono dwie różne metody. Obie metody mogą zlecać dowolne polecenia do realizacji przez EDP. Są to zazwyczaj zlecenia ruchu, ale mogą to również być polecenia: odczytania aktualnego położenia ramienia w dowolnej reprezentacji, zmiany parametrów lub korektora modelu kinematycznego, zmiany stosowanych algorytmów regulacji lub przełączania wyjść i odczytu wejść binarnych.

### 2.4.4 Warunki wstępne

```
class condition {
   // Klasa bazowa dla warunków oczekiwania (klasa abstrakcyjna)
   // W ramach sprawdzania spełnienia warunku początkowego
   // może służyc zarówno do odczytania aktualnego położenia robota
   // jak i określenia reakcji operatora.
   // Spełnienie warunku początkowego kończy oczekiwanie (wait).
   // Stanowi ono warunek początkowy, którego spełnienie umożliwia
   // rozpoczęcie wykonania następnej instrukcji Move.

 public:
   class ECP_error {  // Klasa obsługi błędów warunku
     public:
       unsigned word32 error_class;
       unsigned word32 error_no;
       edp_error error;

       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no)
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = 0; error.error1 = 0; };
       ECP_error ( unsigned word32 err_cl, unsigned word32 err_no,
                   unsigned long int err0, unsigned long int err1 )
                 { error_class = err_cl; error_no = err_no;
                   error.error0 = err0; error.error1 =err1; };
   }; //end: class ECP_error

  virtual BOOLEAN condition_value (list<sensor>* sensor_list,
                                   robot& the_robot) = 0;
     // bada wartość warunku początkowego
     // (zadanie realizowane przez klasę konkretną)
     // TRUE - kończy czekanie (funkcja wait)
     // FALSE - kontynuuje oczekiwanie

}; // end: class condition
```

## 2.5 Obsługa sytuacji awaryjnych

Wystąpienie błędu sygnalizuje się instrukcją `C++ throw`. Wyjątek przechwytywany i analizowany jest przez funkcję `main` procesu ECP. Istnieją cztery klasy obsługi błędów: `ECP_main_error`, `robot::ECP_error`, `generator::ECP_error` oraz `condition::ECP_error`. Trzy ostatnie służą do sygnalizacji błędów powstałych w metodach klas wywiedzionych z `robot`, `generator` i `condition`, natomiast pierwsza do sygnalizacji błędów powstałych poza nimi. Definicja nowego błędu wymaga następujących czynności:

- wprowadzenia do kodu programu użytkowego instrukcji `throw` z argumentem będącym obiektem jednej z powyższych klas,
- rozszerzenia listy warunków instrukcji `switch` w instrukcji `catch` w funkcji `main`,

Fakt wystąpienia błędu zostanie zasygnalizowany procesowi SRP, który wyświetli na ekranie monitora, w odpowiednim okienku, komunikat o rodzaju błędu.

# Rozdział 3

# Klasy konkretne

Obiekty klas pochodnych zarówno dla MP jak procesów ECP omawiane w niniejszym rozdziale wywodzą się z odpowiednich klas bazowych opisanych powyżej.

## 3.1 Klasy konkretne dla procesu MP

### 3.1.1 Roboty

```
//-----------------------------------------------------------------
class rnt_robot: public robot {
// Klasa dla robota rzeczywistego

 public:
  real_robot (char* name) : robot(name) { } // Konstruktor

  virtual void execute_motion (void); // Zlecenie wykonania ruchu przez robota
                                      // na poziomie MP jest to polecenie dla ECP
  virtual void terminate_ecp (void);  // Zlecenie STOP
  virtual void start_ecp ( void );    // Zlecenie START

  virtual void create_command (void);
    // Wypełnia bufor wysyłkowy do EDP na podstawie danych zawartych w swych składowych
    // Ten bufor znajduje się w robocie

  virtual void get_reply (void);
    // Pobiera z pakietu przesłanego z EDP informacje (aktualnie znajdujące się
    // w klasie robot) i wstawia je do odpowiednich swoich składowych
    // Ten bufor znajduje się w robocie

}; // end: class rnt_robot
//----------------------------------------------------------------------------
```

### 3.1.2 Generatory

Zasadniczym elementem tworzonym przez użytkownika przy pisaniu sterowników dedykowanych konkretnym zadaniom są generatory. Poniżej przedstawiono kilka wybranych klas generatorów realizujących różne zadania.

```
//----------------------------------------------------------------------------
```

```
class empty_generator : public robot_generator {
    // Klasa dla generatorów trajektorii:
    // służy zarówno do wyznaczania następnej wartości zadanej jak i
    // sprawdzania spełnienia warunku końcowego
 public:
  empty_generator (void) { }
  ~empty_generator(){ };

  virtual BOOLEAN first_step (list<sensor>* sensor_list, list<robot>* robot_list);
      // generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)
  virtual BOOLEAN next_step (list<sensor>* sensor_list,
                             list<robot>* robot_list);
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)

}; // end: class empty_generator
//-----------------------------------------------------------------
```

Zadaniem tego generatora jest wyłącznie pobudzanie procesów ECP do działania. Trajektorie ruchu generowane są przez procesy ECP (roboty działają niezależnie). Generator `empty_generator` przygotowuje do wysłania do ECP polecenia `NEXT_POSE`. Ponieważ jest to polecenie stałe, metoda `empty_generator::next_step` nie tworzy swoich poleceń.

```
//-----------------------------------------------------------------
class tight_coop_generator : public robot_generator {
    // Klasa dla generatora trajektorii
    // Służy zarówno do wyznaczania następnej wartości zadanej jak i
    // sprawdzania spełnienia warunku końcowego
  trajectory_description td;
  int node_counter; // Licznik zrealizowanych wezlow
  int idle_step_counter; // Licznik jalowych krokow sterowania
                         // (bez wykonywania ruchu)
 public:
  // konstruktor
  tight_coop_generator (trajectory_description tr_des) {
    td = tr_des;
  };
  // destruktor
  ~tight_coop_generator(){ };
  virtual BOOLEAN first_step (list<sensor>* sensor_list, list<robot>* robot_list);
      // generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)
  virtual BOOLEAN next_step (list<sensor>* sensor_list,
                             list<robot>* robot_list);
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)

}; // end: class tight_coop_generator
//-----------------------------------------------------------------
```

Klasa `tight_coop_generator` służy do generacjii ściśle skoordynowanych trajktorii ruchu dla wielu robotów. Jest to zatem przykład ścisłej współpracy dwóch lub więcej robotów. Generator `tight_coop_generator` wytwarza trajektorię prostoliniową przy zadanym

przyroście położenia i orientacji wyrażonych we współrzędnych kartezjańsko-eulerowskich (ZYZ).

### 3.1.3 Warunki

```
//--------------------------------------------------------------
class a_condition: public condition {
   // Klasa konkretnych warunków początkowych
   // Spełnienie warunku początkowego kończy wykonanie instrukcji WAIT.
   // Stanowi ono warunek początkowym, którego spełnienie umożliwia
   // rozpoczęcie wykonania następnej instrukcji MOVE.
  public:
  virtual BOOLEAN condition_value (list<robot>* robot_list,
                                   list<sensor>* sensor_list);
      // obliczenie wartości warunku początkowego
}; // end: class condition
//--------------------------------------------------------------
```

Klasa `a_condition` umożliwia implementację prostego warunku dla funkcji `Wait`.

## 3.2 Klasy konkretne dla procesów ECP

### 3.2.1 Robot

```
//--------------------------------------------------------------
class rnt_robot: public robot {
// Klasa dla przykładowego rzeczywistego robota
  word08 EDP_name[30];  // Nazwa procesu EDP
  pid_t EDP_MASTER_Pid; // Identyfikator procesu driver'a edp_m
  pid_t MP_pid;
  BOOLEAN synchronised; // Flaga synchronizacji robota
                      // (TRUE - zsynchronizowany, FALSE - nie)

 public:
  rnt_robot(const word08 *edp_name); // Konstruktor

  virtual void execute_motion (void); // Zlecenie wykonania ruchu przez robota:
                                     // jest to polecenie dla EDP

  virtual void set_edp_master_pid ( pid_t edppid ) {EDP_MASTER_Pid = edppid;};
                                    // Przekazanie identyfikatora procesu EDP_MASTER
  virtual void set_mp_pid ( pid_t mp_pid)  {MP_pid = mp_pid;};
                                   // Przekazanie identyfikatora procesu MP
  virtual void synchronise ( void ); // Zlecenie synchronizacji robota

  // Czy robot zsynchronizowany?
  virtual BOOLEAN is_synchronised ( void ) {return synchronised;};

  virtual void create_command (void);
    // Wypełnia bufor wysyłkowy do EDP na podstawie danych zawartych w obrazie
    // robota wykorzystywanych przez generator
    // Ten bufor znajduje sie w robocie

  virtual void get_reply (void);
```

```
    // Pobiera z pakietu przesłanego z EDP informacje i wstawia je do
    // odpowiednich składowych obrazu robota wykorzystywanych przez generator
    // Ten bufor znajduje sie w robocie
    // Informacja dla MP o zakonczeniu zadania uzytkownika
  virtual void ecp_termination_notice (void);

  // Oczekiwanie na polecenie START od MP
  virtual BOOLEAN ecp_wait_for_start (void);

  // Oczekiwanie na STOP
  virtual void ecp_wait_for_stop (void);

  // Oczekiwanie na polecenie od MP
  virtual void get_mp_command (void);

  // Badanie typu polecenia z MP
  virtual MP_COMMAND mp_command_type (void) { return mp_command.command; };

  // Ustawienie typu odpowiedzi z ECP do MP
  virtual void set_ecp_reply ( ECP_REPLY ecp_r) { ecp_reply.reply = ecp_r; };

  // Kopiowanie bufora przesylanego z MP do bufora wysylanego do EDP
  virtual void copy_mp_to_edp_buffer (void) {
    memcpy( &EDP_command_and_reply_buffer.instruction,
            &mp_command.mp_package.instruction, sizeof(c_buffer));
  }
}; // end: class rnt_robot
//-----------------------------------------------------------------
```

Jest to uniwersalna klasa robotów konkretnych.

### 3.2.2 Generatory

Generatory są zasadniczym elementem pisanym przez użytkownika tworzeniu sterownika realizującego rzeczywiste zadanie. Poniżej przedstawiono kilka wybranych klas generatorów dla procesów ECP realizujących różne rodzaje ruchów.

```
//------------------------------------------------------------------------
// Generator czyniący ECP przezroczystym. Faktyczna generacja trajektorii
// odbywa sie w MP
class generator_t : public rnt_generator {

public:
  virtual BOOLEAN first_step (list<sensor>* sensor_list, robot& the_robot);
      // generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)
  virtual BOOLEAN next_step (list<sensor>* sensor_list, robot& the_robot);
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)

}; //end: class generator_t
//------------------------------------------------------------------------
```

Zadaniem tego generatora jest jedynie przesyłanie punktów trajektorii obliczonych w generatorze działającym w procesie MP (patrz `tight_coop_generator`).

```
//----------------------------------------------------------------------------
// Generator odtwarzajacy nauczone pozycje
class teach_in_generator : public rnt_generator {
protected:
   list <taught_in_pose>* pose_list_ptr;    // wskaźnik na początek listy
                                            // nauczonych pozycji
   list <taught_in_pose>* current_pose_ptr; // wskaźnik na aktualnie
                                            // odtwarzaną pozycję

public:

  // konstruktor
  teach_in_generator (void) { pose_list_ptr = NULL; current_pose_ptr = NULL; };

  // destruktor
  ~teach_in_generator (void) { flush_pose_list(); }

  void flush_pose_list ( void ) {
    // niszczy  liste nauczonych pozycji
    if (pose_list_ptr != NULL) {
      pose_list_ptr->delete_list_elements();
      delete pose_list_ptr;
      pose_list_ptr = NULL;
      current_pose_ptr = NULL;
    }
  }; // end: flush_pose_list

  void initiate_pose_list(void) { current_pose_ptr = pose_list_ptr; };

  void next_pose_list_ptr (void) {
    if (current_pose_ptr != NULL)
      current_pose_ptr = current_pose_ptr->next;
  }

  void get_pose (taught_in_pose& tip) {
    tip.arm_type = current_pose_ptr->E_ptr->arm_type;
    tip.motion_time = current_pose_ptr->E_ptr->motion_time;
    memcpy(tip.coordinates, current_pose_ptr->E_ptr->coordinates,
           6*sizeof(double));
  }
// Pobierz nastepna pozycje z listy
  void get_next_pose (double next_pose[6])  {
    memcpy(next_pose, current_pose_ptr->next->E_ptr->coordinates, 6*sizeof(double));
  };

  void set_pose (POSE_SPECIFICATION ps, double motion_time, double coordinates[6]) {
    current_pose_ptr->E_ptr->arm_type = ps;
    current_pose_ptr->E_ptr->motion_time = motion_time;
    memcpy(current_pose_ptr->E_ptr->coordinates, coordinates, 6*sizeof(double));
  }

  BOOLEAN is_pose_list_element ( void ) {
    // sprawdza czy aktualnie wskazywany jest element listy, czy lista sie skonczyla
    if ( current_pose_ptr )
      return TRUE;
    else
      return FALSE;
  }
```

```
  BOOLEAN is_last_list_element ( void ) {
    // sprawdza czy aktualnie wskazywany element listy ma nastepnik
    if ( current_pose_ptr )
      if ( current_pose_ptr->next )
        return TRUE;
    return FALSE;
  }

  void create_pose_list_head (POSE_SPECIFICATION ps, double motion_time,
                             double coordinates[6]) {
    pose_list_ptr = new list<taught_in_pose>(new taught_in_pose(ps,
                                            motion_time, coordinates) );
    current_pose_ptr = pose_list_ptr;
  }

  void insert_pose_list_element (POSE_SPECIFICATION ps, double motion_time,
                                                 double coordinates[6]) {
    current_pose_ptr->insert_list_element (new taught_in_pose(ps,
                                                 motion_time, coordinates));
    current_pose_ptr = current_pose_ptr->next;
  }

  int pose_list_length(void) { return pose_list_ptr->list_length(); };

  virtual BOOLEAN first_step (list<sensor>* sensor_list, robot& the_robot);
      // generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)
  virtual BOOLEAN next_step (list<sensor>* sensor_list, robot& the_robot);
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)

}; //end: class teach_in_generator
//--------------------------------------------------------------------------


//--------------------------------------------------------------------------
// Generator odtwarzający nauczone pozycje z rozpędzaniem i hamowaniem między
// nauczonymi pozycjami z dokładną zadaną pozycją końcową
class parabolic_teach_in_generator : public teach_in_generator {

protected:
  taught_in_pose tip;      // Kolejna z listy nauczonych pozycji.
                           // Trajektoria między kolejnymi nauczonymi
                           // pozycjami dzielona jest na przedziały interpolacji.
                           // Każdy przedzial interpolacji jest równoważny jednemu
   // makrokrokowi wykonywanemu przez EDP.
  BOOLEAN first_interval;  // TRUE  - jeżeli ruch odbywa się w pierwszym przedziale
                           // interpolacji
                           // FALSE - w przeciwnym przypadku
  int node_counter;        // Bieżący węzeł interpolacji
  int number_of_intervals; // Liczba przedziałów interpolacji - wynika z
                           // całkowitego czasu ruchu
  int half_number_of_intervals; // Połowa liczby przedziałów interpolacji

  double next_pose[6];     // Nastepna pozycja na liście - punkt końcowy
                           // trajektorii parabolicznej
  double a[6];             // Tablica faktycznie realizowanych przyspieszeń
                           // dla kolejnych osi/współrzednych
```

```
  double INTERVAL;         // Długość okresu interpolacji w [sek]
  double a_max_motor[6];   // Tablica dopuszczalnych przyspieszeń dla kolejnych
                           // osi/współrzednych
  double v_max_motor[6];   // Tablica dopuszczalnych predkości
   // dla kolejnych osi/współrzednych
  double a_max_joint[6];   // Tablica dopuszczalnych przyspieszeń
   // dla kolejnych osi/współrzednych

  double v_max_joint[6];   // Tablica dopuszczalnych predkości
   // dla kolejnych osi/współrzednych

  double a_max_zyz[6];     // Tablica dopuszczalnych przyspieszeń
   // dla kolejnych osi/współrzednych
  double v_max_zyz[6];     // Tablica dopuszczalnych predkości
   // dla kolejnych osi/współrzednych

  double a_max_aa[6];      // Tablica dopuszczalnych przyspieszeń
   // dla kolejnych osi/współrzednych
  double v_max_aa[6];      // Tablica dopuszczalnych predkości
   // dla kolejnych osi/współrzednych

public:
  parabolic_teach_in_generator (void); // konstruktor
  virtual BOOLEAN first_step (list<sensor>* sensor_list, robot& the_robot);
      // Generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)
  virtual BOOLEAN next_step (list<sensor>* sensor_list, robot& the_robot);
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)

}; //end: parabolic_teach_in_generator
//--------------------------------------------------------------------------


//--------------------------------------------------------------------------
// Generator trajektorii prostoliniowej
class linear_generator : public rnt_generator {
protected:
int node_counter;

public:
  trajectory_description td;
  // konstruktor
  linear_generator (trajectory_description tr_des) {
    td = tr_des;
  };

  virtual BOOLEAN first_step (list<sensor>* sensor_list, robot& the_robot);
      // generuje pierwszy krok ruchu -
      // pierwszy krok często różni się od pozostałych,
      // np. do jego generacji nie wykorzystuje się czujników
      // (zadanie realizowane przez klasę konkretną)
  virtual BOOLEAN next_step (list<sensor>* sensor_list, robot& the_robot);
     // generuje każdy następny krok ruchu
     // (zadanie realizowane przez klasę konkretną)
}; //end: class linear_generator
//--------------------------------------------------------------------------
```

### 3.2.3 Warunki

```
// --------------------------------------------------------------------------
class operator_reaction_condition: public rnt_condition {
   // Klasa konkretnych warunków początkowych dla robota RNT
   // Spełnienie warunku początkowego kończy wykonanie instrukcji WAIT.
   // Stanowi ono warunek początkowy, którego spełnienie umożliwia
   // rozpoczęcie wykonania następnej instrukcji MOVE.
protected:
  // Wskaźnik na początek listy odczytanych pozycji
  list <taught_in_pose>* supplementary_list_ptr;

  // Wskaźnik na aktualnie odczytaną pozycję
  list <taught_in_pose>* current_supplementary_list_ptr;

public:

  // konstruktor
  operator_reaction_condition (void) { supplementary_list_ptr = NULL;
                                     current_supplementary_list_ptr = NULL; };

  // destruktor
  ~operator_reaction_condition (void) { flush_supplementary_list(); }

  void flush_supplementary_list ( void ) {
    // niszczy  liste odczytanych pozycji
    if (supplementary_list_ptr != NULL) {
      supplementary_list_ptr->delete_list_elements();
      delete supplementary_list_ptr;
      supplementary_list_ptr = NULL;
      current_supplementary_list_ptr = NULL;
    }
  }; // end: flush_supplementary_list

  void initiate_supplementary_list(void)
    { current_supplementary_list_ptr = supplementary_list_ptr; };

  void next_supplementary_list_ptr (void) {
    if (current_supplementary_list_ptr != NULL)
      current_supplementary_list_ptr = current_supplementary_list_ptr->next;
  }

  void get_supplementary (taught_in_pose& tip) {
    tip.arm_type = current_supplementary_list_ptr->E_ptr->arm_type;
    tip.motion_time = current_supplementary_list_ptr->E_ptr->motion_time;
    memcpy(tip.coordinates, current_supplementary_list_ptr->E_ptr->coordinates,
           6*sizeof(double));
  }

  void set_supplementary (POSE_SPECIFICATION ps, double motion_time,
                          double coordinates[6]) {
    current_supplementary_list_ptr->E_ptr->arm_type = ps;
    current_supplementary_list_ptr->E_ptr->motion_time = motion_time;
    memcpy(current_supplementary_list_ptr->E_ptr->coordinates, coordinates,
           6*sizeof(double));
  }

  void create_supplementary_list_head (POSE_SPECIFICATION ps, double motion_time,
                                     double coordinates[6]) {
```

```
    supplementary_list_ptr = new list<taught_in_pose>(new taught_in_pose(ps,
                                           motion_time, coordinates) );
    current_supplementary_list_ptr = supplementary_list_ptr;
  }

  void insert_supplementary_list_element (POSE_SPECIFICATION ps,
                              double motion_time, double coordinates[6]) {
    current_supplementary_list_ptr->insert_list_element (new taught_in_pose(ps,
                                           motion_time, coordinates));
    current_supplementary_list_ptr = current_supplementary_list_ptr->next;
  }

  BOOLEAN is_supplementary_list_element ( void ) {
    // Sprawdza czy aktualnie wskazywany jest element listy,
    // czy lista już skonczyla się

    if ( current_supplementary_list_ptr )
      return TRUE;
    else
      return FALSE;
  }

  BOOLEAN is_supplementary_list_head ( void ) {
    // sprawdza czy aktualnie wskazywany jest początek listy,
    // czy lista jest pusta?
    if ( supplementary_list_ptr )
      return TRUE;
    else
      return FALSE;
  }

  int supplementary_list_length(void)
    { return supplementary_list_ptr->list_length(); };

  virtual BOOLEAN condition_value (list<sensor>* sensor_list, robot& the_robot);
     // Bada wartośś warunku poczatkowego
     // TRUE - konczy czekanie (funkcja wait)
     // FALSE - kontynuuje oczekiwanie
}; // end: class operator_reaction_condition
// ---------------------------------------------------------------------------
```

### 3.2.4 Czujniki

Rozważany jest dwustanowy czujnik klawiszowy połączony z systemem sterującym robota. Pomiar stanu klawiszy dostarcza tu ośmiu wartości. Odczytami są wartości binarne `0` albo `1` określące stan wciśnięcia klawiszy. Oczyt z każdego kanału dostarczany jest do systemu sterowania przez przetwornik A-C.

Po stronie ECP zdefiniowana jest struktura stanowiąca **obraz czujnika**:

```
typedef struct {
   VSP_COMMAND instruction_code;
   int reply_code;
   unsigned int reading_type;
   int int_parameter;
   float float_parameter;
   scalled force;       //dane "nowe" przeskalowane
```

```
   u8_data binary_data; // dane w postaci binarnej
   int all_bin_data;    //dane skumulowane z czujnika binarnego
   int_8 bin_sensor;
} sensor_image;
```

Pierwsze pole zawiera kod instrukcji przesyłanej do procesu VSP przez ECP (dla rozważanego przykładu jest to: `VSP_INIT, VSP_GET_READING`, lub `VSP_TERMINATE`). Kolejne pole (`reply_code`) jest kodem odpowiedzi procesu VSP. Dana `reading_type` jest informacją procesu VSP o typie odczytanych danych. Kolejne dwa pola: `int_parameter`, `float_parameter` są opcjonalnym parametrem instrukcji przesyłanej z ECP (może to byc parametr wymagany przez czujnik, lub inna dana). Parametr ten może być typu zmiennoprzecinkowego albo typu całkowitego. Dwa kolejne pola przeznaczone są na dane odebrane z VSP dla opracowywanego też czujnika sił. Dane typu `scalled` stanowią zbiór sześciu wartości sił odczytanych z czujników tensometrycznych i przeskalowanych zgodnie z zakresem stosowanego przetwornika A-C (8 bitów: zakres 0–255) oraz zakresem napięć toru pomiarowego (5000mV). Dane typu `binary_data` to dane nie przeskalowane, a więc z zakresu 0–255. W zależności od informacji o typie danych – `reading_type`, proces ECP "wie" jakiego typu dane powinien odebrać. Typ danych definioway jest przez programistę po stronie procesu VSP, co zostanie jeszcze omówione.

Dla komentowanego tu czujnika klawiszowego dane są zapisywane jako liczba całkowita `all_bin_data` zawierająca dane skumulowane z czujnika klawiszowego (osiem klawiszy - osiem bitów) albo jako osiem liczb strutury `bin_sensor` typu int_8.

Poniżej pokazano **klasę konkretną czujnika** odpowiadajacą omawianemu tu przykładowi:

```
// KLASA sensor po stronie ECP dla czujnika dotykowego
class bin_sensor : public sensor {
  // Obraz czujnika po stronie ECP
  //   - odczytane dane + ewentualnie dane historyczne + błędy
  // W obrazie można zgromadzic dane o poleceniu dla VSP oraz dane odczytane z VSP

pid_t VSPpid; // identyfikator procesu VSP

public:
    bin_sensor ( nid_t VSP_node, char *VSP_program );
       //konstruktor: (nr wezla, sciezka i nazwa VSP)

    ~bin_sensor ( void ) {};
       // destruktor

    virtual void initiate_reading ( void ) { };
       // zadanie odczytu od VSP

    virtual void get_reading ( void );
       // odebranie odczytu od VSP

   virtual void create_sensor_command ( void );
     // to przepisuje rozkaz utworzony w obrazie do bufora to_vsp

   virtual void get_sensor_reply ( void );
     // to przepisuje odczyt z bufora from_vsp do obrazu czujnika

   virtual void terminate ( void );
```

```
    // rozkaz zakonczenia procesu VSP

}; // end: bin_sensor
```

Konstruktor `bin_sensor::bin_sensor` tworzy i incjuje obiekt klasy `bin_sensor` co oznacza odczytanie programu procesu VSP wskazywanego przez `*VSP_program` i utworzenie procesu wykonującego ten program. Metoda `bin_sensor::get_reading` korzysta z metody `bin_sensor::create_sensor_command` w celu wypełnienia bufora wysyłkowego `msg_ecp_vsp` przesyłanego przez proces ECPdo procesu VSP, następnie przesyła ten bufor i odbiera bufor `msg_vsp_ecp`. W przypadku wystąpienia błędu przesłania generowany jest wyjątek `ECP_SYSTEM_ERROR`. Na koniec, w celu przetworzenia otrzymanego bufora, wywoływana jest metoda `bin_sensor::get_sensor_reply`.

Metody `bin_sensor::create_sensor_command` i `bin_sensor::get_sensor_reply` są omówione dokładniej poniżej.

Metoda `bin_sensor::create_sensor_command` wypełnia bufor wysyłkowy `msg_ecp_vsp` wysyłany do VSP zgodnie z informacją zawartą w polu kodu instrukcji obrazu czujnika przechowywanego w ECP: `image.instruction_code` .

Metoda `bin_sensor::get_sensor_reply` przetwarza bufor `msg_vsp_ecp` otrzymany z VSP. Jeżeli w polu `msg_vsp_ecp.VSP_report` zawarta jest informacja o prawidłowym odczycie (`VSP_reply_OK`) to do pola `image.reading_type` obrazu przepisywana jest informacja o typie odebranych danych a do odpowiednich pól danych: `image.all_bin_data` albo `image.bin_sensor` przepisywane są odczyty z czujników klawiszowych zawarte w odpowiednich polach bufora odebranego z VSP. Jest to `vsp_reply_buffer`. W przypadku braku informacji o poprawnym odczycie lub przy innych błędach generowane sa odpowiednie wyjątki.

## Rozdział 4

# Funkcje i metody procesu MP

W tym rozdziale zgromadzono wszystkie funkcje i metody, które mogą być przydatne dla użytkownika (programisty systemu MRROC++) przy tworzeniu części programu użytkowego znajdującego się w procesie ECP. Zamieszczono zarówno metody, które użytkownik będzie stosował konstruując sterownik dedykowany swemu specyficznemu zadaniu, jak i metody, które nie będą bezpośrednio przez użytkownika wykorzystywane w jego programach, ale są niezbędne dla zrozumienia sposobu działania tych pierwszych. Większość z zamieszczonych metod klas wywiedzionych z klasy `generator` stanowi przykład (wzór), który można wykorzystać przy tworzeniu swych własnych generatorów. Niemniej jednak były one stosowane do realizacji wielu zadań, więc mogą być także wykorzystane przez użytkownika dokładnie w przedstawionej postaci.

Funkcje i metody zostały wymienione w porządku alfabetycznym. W przypadku metod wpierw uwzględniono nazwę klasy, z której pochodzą.

**Definicja:** `empty_generator::empty_generator (void)`

`#include "mp.h"`

**Opis:** Metoda `empty_generator::empty_generator` tworzy i inicjuje obiekt klasy `empty_generator`.

**Rezultat:** Metoda `empty_generator::empty_generator` nie zwraca żadnej wartości (jest konstruktorem obiektu).

**Błędy:** Metoda `empty_generator::empty_generator` nie zgłasza wyjątków.

**Zobacz:** `empty_generator.first_step`, `empty_generator.next_step`

**Definicja:**

```
BOOLEAN empty_generator::first_step (
                          list<sensor>* sensor_list,
                          list<robot>* robot_list)
```

```
#include "mp.h"
```

**Opis:** Metoda `empty_generator::first_step` określa czynności związane z generacją pierwszego kroku ruchu. Zadaniem tego generatora jest pobudzanie procesów ECP do działania. Faktyczne trajektorie ruchu generowane są przez procesy ECP (roboty działają niezależnie). W związku z tym generator ten przygotowuje do wysłania do ECP polecenia `NEXT_POSE`. Ponieważ jest to polecenie stałe, metoda `empty_generator::next_step` tego generatora nie musi tworzyć swoich poleceń. Następnie metoda `empty_generator::first_step` wywołuje `copy_generator_command`, aby skopiować polecenia przygotowane przez generator do obrazów robotów znajdujących się na liście. Rozkazy te będą wysłane (do wykonania przez ECP) przez odpowiednie metody `execute_motion` znajdujące się w funkcji `Move`. Chwilowo założono, że lista robotów jest jednoelementowa.

**Rezultat:** Metoda `empty_generator::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

`true` – warunek końcowy nie jest spełniony (kontynuować ruch)
`false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `empty_generator::first_step` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `empty_generator::first_step`.

**Zobacz:** `empty_generator.next_step`

**Przykład:** Patrz tekst `Move` dla procesu MP.

**Definicja:**

```
BOOLEAN empty_generator::next_step (
                            list<sensor>* sensor_list,
                            list<robot>* robot_list)

#include "mp.h"
```

**Opis:** Metoda `empty_generator::next_step` określa czynności związane z generacją kolejnych kroków ruchu. Zadaniem tego generatora jest pobudzanie procesów ECP do działania. Faktyczne trajektorie ruchu generowane są przez procesy ECP (roboty działają niezależnie). W związku z tym metoda `empty_generator::next_step` nie przygotowuje nowego polecenia dla ECP. Ponieważ jest to polecenie stałe, metoda `empty_generator::first_step` już to zrobiła. Następnie metoda `empty_generator::next_step` wywołuje `copy_data`, aby skopiować dane przesłane przez proces ECP z buforów przesyłowych do obrazów robotów znajdujących się na liście. Przy okazji zostanie sprawdzone, czy z ECP przysłano `TASK_TERMINATED`. Jeżeli tak, to generowanie trajektorii jest kończone. W ten sposób zostanie uaktualniony stan obrazów robotów. Rozkazy przygotowane przez `empty_generator::first_step` będą wysłane (do wykonania przez ECP) przez odpowiednie metody `execute_motion` znajdujące się w funkcji `Move`. Chwilowo założono, że lista robotów jest jednoelementowa.

**Rezultat:** Metoda `empty_generator::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

- `true` – warunek końcowy nie jest spełniony (kontynuować ruch)
- `false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `empty_generator::next_step` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `empty_generator::next_step`.

**Zobacz:** `empty_generator.first_step`

**Przykład:** Patrz tekst `Move` dla procesu MP.

**Definicja:**

```
BOOLEAN Move (list<robot>* robot_list,
              list<sensor>* sensor_list,
              generator& the_generator)
```

```
#include "mp.h"
```

**Opis:** Funkcja `Move` realizuje ruch effektorów umieszczonych na liście `robot_list` z wykorzystaniem informacji pochodzacych w czujników wirtualnych umieszczonych na liście `sensor_list`. Ruch jest generowany zgodnie ze specyfikacją określoną przez `the_generator`. Na wstępie sprawdza się czy nie zostało przysłane jakieś polecenie od operatora. Jeżeli tak, to jest ono odbierane. Pierwszy kroku ruchu określony jest przez metodę `the_generator.first_step`. Często sprowadza się on do odczytu aktualnego położenia efektorów (robotów). Następnie żądane są dane od wszystkich czujników umieszczonych na liście `sensor_list`. Po tym cyklicznie zlecane jest wykonanie kroku ruchu przez metody `the_robot.execute_motion`. Powoduje to wykonanie ruchu przez wszystkie efektory umieszczone na liście `robot_list`. W kolejnej fazie wykonania funkcji `Move` odczytywane są dane uzyskane przez wszystke czujniki umieszczone na liście `sensor_list`. Na koniec metoda `the_generator.next_step` sprawdza warunek końcowy oraz jeżeli nie jest on spełniony oblicza kolejny krok ruchu, po czym powyższe operacje są ponawiane. W przeciwnym przypadku `Move` kończy swe działanie. Sieć działań funkcji `Move` przedstawiono na rysunku 4.1.

**Rezultat:** Funkcja `Move` zwraca wartość `BOOLEAN`. Wartość ta określa tryb zakończenia ruchu, tzn.:

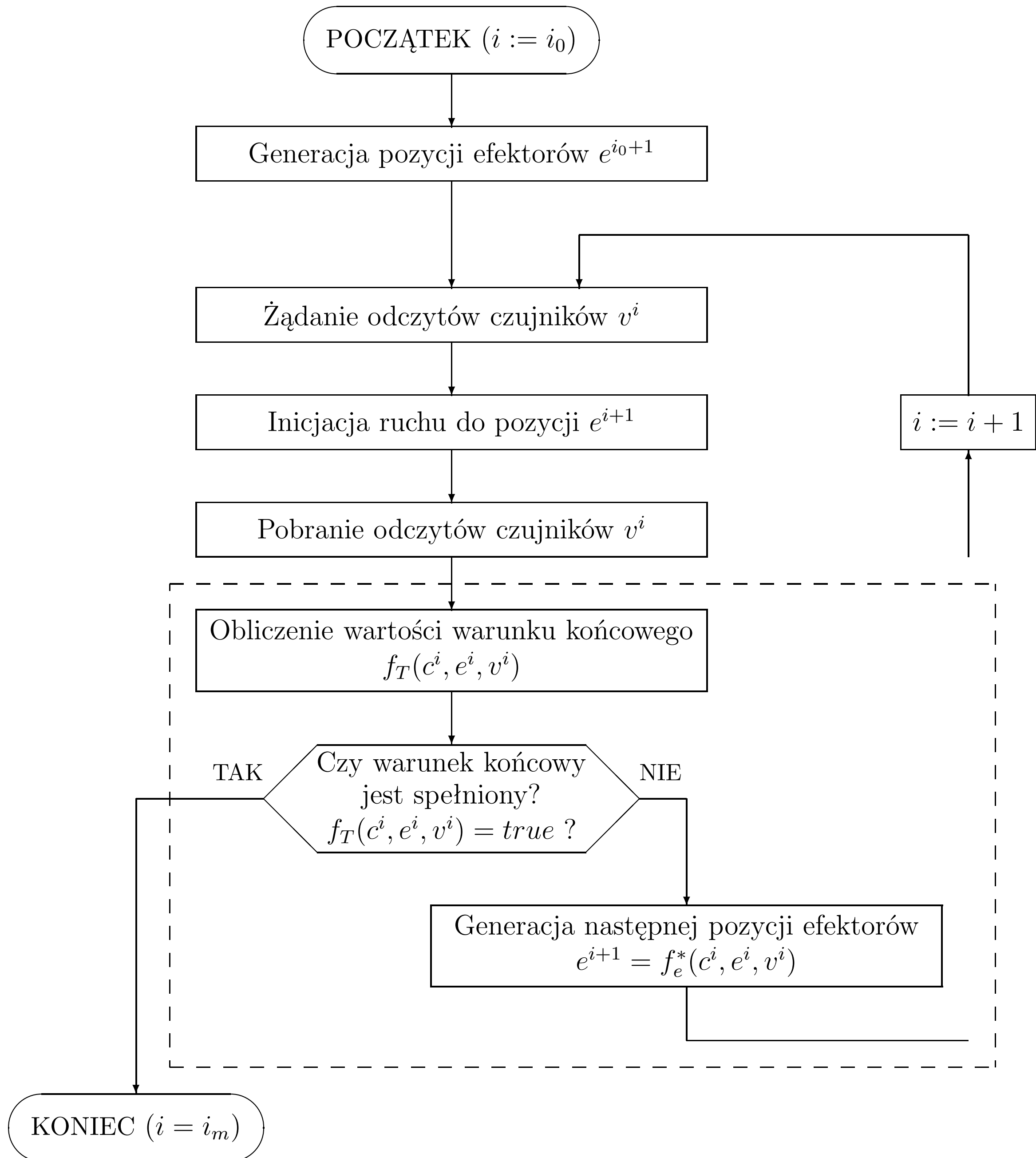
| | | |
|---|---|---|
| `true` | – | przerwano ruch na skutek polecenia operatora (`STOP`) |
| `false` | – | przerwano ruch na skutek spełnienia warunku końcowego |

**Błędy:** Funkcja `Move` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są błędy systemowe `SYSTEM_ERROR`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez funkcję `Move`.

**Tekst:**

```
BOOLEAN Move(list<robot>*  robot_list, list<sensor>* sensor_list,
             generator& the_generator) {

 // Funkcja zwraca FALSE gdy samoistny koniec ruchu
 // Funkcja zwraca TRUE gdy koniec ruchu wywołany jest przez STOP

 list<sensor>*  sensor_lptr;          // sensor list pointer
 list<robot>*   robot_lptr;           // robot list pointer
 pid_t pid;                           // identyfikator proxy
 unsigned long int e;

 // (Inicjacja) generacja pierwszego kroku ruchu
 if (!the_generator.first_step(sensor_list, robot_list) )
   return FALSE;

 do { // realizacja ruchu
    // Sprawadzenie czy przyszło polecenie operatora
    if((pid = Creceive(MPProxyPids.stop, NULL, 0 ) ) != -1 ) {
```

```
        // Przesłanie polecenia STOP do ECP
        for ( robot_lptr = robot_list;  robot_lptr;
              robot_lptr = robot_lptr->next ) {
          robot_lptr->E_ptr->terminate_ecp();
        } // end: for
        return TRUE;
      }
      else if (errno != ENOMSG) {
        // Błąd komunikacji miedzyprocesowej - wyjatek
        e = errno;
        perror("Creceive STOP proxy from UI failed ?\n");
        msg->message(SYSTEM_ERROR, e,
                     "MP: Creceive STOP proxy from UI failed");
        throw MP_main_error (SYSTEM_ERROR, (unsigned word32) 0);
      }

      if ( ( pid = Creceive(MPProxyPids.pause, NULL, 0) ) != -1 ) {
        msg->message("To resume MP click RESUME icon");
        //Oczekiwanie na RESUME
        if (Receive (MPProxyPids.resume, NULL, 0) == -1) {
          // Wystapił błąd systemowy przy Receive
        e = errno;
        perror("Creceive RESUME proxy from UI failed ?\n");
        msg->message(SYSTEM_ERROR, e,
                     "MP: Creceive RESUME proxy from UI failed");
        throw MP_main_error (SYSTEM_ERROR, (unsigned word32) 0);
        }
        msg->message("MP user program is running");
      }
      else if (errno != ENOMSG) {
        // Błąd komunikacji miedzyprocesowej - wyjatek
        e = errno;
        perror("Creceive PAUSE proxy from UI failed ?\n");
        msg->message(SYSTEM_ERROR, e,
                     "MP: Creceive PAUSE proxy from UI failed");
        throw MP_main_error (SYSTEM_ERROR, (unsigned word32) 0);
      }

     // żądanie danych od wszystkich czujników
     for (sensor_lptr = sensor_list; sensor_lptr;
          sensor_lptr = sensor_lptr->next)
       sensor_lptr->E_ptr->initiate_reading();

     // wykonanie kroku ruchu przez wszystkie roboty
     for (robot_lptr = robot_list; robot_lptr;
          robot_lptr = robot_lptr->next) {
       robot_lptr->E_ptr->execute_motion();
     } // end: for

     // odczytanie danych z wszystkich czujników
     for (sensor_lptr = sensor_list; sensor_lptr;
          sensor_lptr = sensor_lptr->next)
       sensor_lptr->E_ptr->get_reading();

   } while ( the_generator.next_step( sensor_list, robot_list) );
   // end: do

   return FALSE;
  }; // end: Move()
```




**Rys. 4.1:** Sieć działań instrukcji `Move` w procesie MP, z zaznaczonym zakresem działania metody `generator.next_step`

**Definicja:** `void rnt_robot::create_command (void)`

`#include "mp.h"`

**Opis:** Metoda `rnt_robot::create_command` wypełnia bufor wysyłkowy do ECP na podstawie danych zawartych w składowych obrazu robota. Dane te zostały tam umieszczone przez generator. Obraz robota stanowi `ecp_td` typu `robot_ECP_transmission_data` i jest składową klasy `robot`. Buforem wysyłowym jest `mp_command` typu `MP_COMMAND_PACKAGE`. Aby poprawnie wypełnić bufor analizowana jest zawartość składowych obrazu robota. Tak skompletowane polecenie zostanie przesłane do ECP przez metodę `execute_motion` (właściwą dla danego robota). Metoda `execute_motion` jest wywoływana w funkcji `Move`.

**Rezultat:** Metoda `rnt_robot::create_command` nie zwraca żadnej wartości (jest procedurą). Jest to negacja warunku końcowego, tzn.:

`true` – warunek końcowy nie jest spełniony (kontynuować ruch)
`false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `rnt_robot::create_command` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne:

`INVALID_POSE_SPECIFICATION`,
`INVALID_RMODEL_TYPE`,
`INVALID_ECP_COMMAND`.

Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `rnt_robot::create_command`.

**Definicja:** `void rnt_robot::execute_motion (void)`

`#include "mp.h"`

**Opis:** Metoda `rnt_robot::execute_motion` zleca wykonanie ruchu przez robota. Jest to polecenie dla ECP. Przesyłane jest funkcją `Send`.

**Rezultat:** Metoda `rnt_robot::execute_motion` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `rnt_robot::execute_motion` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `ECP_ERRORS`. Ponadto przy przesyłaniu polecenia do ECP mogą wystąpić błędy systemowe `SYSTEM_ERROR`.

**Zobacz:** `Move`

**Przykład:** Patrz tekst `Move` dla procesu MP.

**Definicja:** `void rnt_robot::get_reply (void)`

`#include "mp.h"`

**Opis:** Metoda `rnt_robot::get_reply` pobiera z pakietu przesłanego z ECP informacje i wstawia je do odpowiednich składowych obrazu robota. Obraz robota stanowi `ecp_td` typu `robot_ECP_transmission_data` i jest składową klasy `robot`. Bufor odbierany od ECP to `ecp_reply` typu `ECP_REPLY_PACKAGE`, stanowiący składową klasy `robot`. Aby prawidłowo wypełnić obraz robota danymi informacje zawarte w buforze muszą zostać przeanalizowane.

**Rezultat:** Metoda `rnt_robot::get_reply` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `rnt_robot::get_reply` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne:
`INVALID_POSE_SPECIFICATION`,
`INVALID_RMODEL_TYPE`,
`INVALID_EDP_REPLY`.

**Definicja:** `void rnt_robot::start_ecp(void)`

```
#include "mp.h"
```

**Opis:** Metoda `rnt_robot::start_ecp` realizuje wysłanie polecenia START do procesu pojedynczego procesu ECP. Jeśli odebrana odpowiedź jest ECP_ACKNOWLEDGE to polecenie zostało przyjęte, w przeciwnym przypadku zgłaszany jest błąd.

**Rezultat:** Metoda `rnt_robot::start_ecp` nie zwraca żadnej wartości.

**Błędy:** Funkcja `rnt_robot::start_ecp` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Generowany jest błąd niefatalny:

`ECP_ERRORS` – wysłane polecenie było niedozwolone w tym momencie lub wykryto inny błąd w ECP.

Ponadto jest wykrywany błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem MP a ECP

**Zobacz:** `rnt_robot::terminate_ecp, start_all, terminate_all`

**Definicja:** `void rnt_robot::terminate_ecp(void)`

```
#include "mp.h"
```

**Opis:** Metoda `rnt_robot::terminate_ecp` realizuje wysłanie polecenia STOP do procesu pojedynczego procesu ECP. Jeśli odebrana odpowiedź jest ECP_ACKNOWLEDGE to polecenie zostało przyjęte, w przeciwnym przypadku zgłaszany jest błąd.

**Rezultat:** Metoda `rnt_robot::terminate_ecp` nie zwraca żadnej wartości.

**Błędy:** Funkcja `rnt_robot::terminate_ecp` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Generowany jest błąd niefatalny:

`ECP_ERRORS` – wysłane polecenie było niedozwolone w tym momencie lub wykryto inny błąd w ECP.

Ponadto jest wykrywany błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem MP a ECP

**Zobacz:** `rnt_robot::start_ecp, start_all, terminate_all`

**Definicja:** `void copy_data (list<robot>* robot_list)`

`#include "mp.h"`

**Opis:** Metoda `robot_generator::copy_data` kopiuje dane z buforów przesyłowych otrzymanych z procesów ECP do odpowiednich obrazów robotów. Dla każdego robota używa odpowiedniej metody `get_reply`. Czyni to dla wszystkich robotów umieszczonych na liście.

**Rezultat:** Metoda `robot_generator::copy_data` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `robot_generator::copy_data` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `robot_generator::copy_data`.

**Zobacz:** `rnt_robot::get_reply`

**Definicja:** `void copy_generator_command (list<robot>* robot_list)`

`#include "mp.h"`

**Opis:** Metoda `robot_generator::copy_generator_command` kopiuje polecenia stworzone przez generator, a umieszczone w obrazach robotów, do buforów przesyłowych odpowiednich procesów ECP. Dla każdego robota używa odpowiedniej metody `create_command`. Czyni to dla wszystkich robotów umieszczonych na liście.

**Rezultat:** Metoda `robot_generator::copy_generator_command` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `robot_generator::copy_generator_command` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `robot_generator::copy_data`.

**Zobacz:** `rnt_robot::create_command`

**Definicja:** `void start_all(list<robot>& head )`

```
#include "mp.h"
```

**Opis:** Funkcja `start_all` umożliwia wysłanie polecenia START do wszystkich procesów ECP umieszczonych na liście `head`.

**Rezultat:** Funkcja `start_all` nie zwraca żadnej wartości.

**Błędy:** Funkcja `start_all` nie wykrywa żadnych błędów.

**Zobacz:** `terminate_all, start_ecp`

**Definicja:**

```
void terminate_all(list<robot>& head )

#include "mp.h"
```

**Opis:** Funkcja `terminate_all` umożliwia wysłanie polecenia STOP do wszystkich procesów ECP umieszczonych na liście `head`.

**Rezultat:** Funkcja `terminate_all` nie zwraca żadnej wartości.

**Błędy:** Funkcja `terminate_all` nie wykrywa żadnych błędów.

**Zobacz:** `start_all`, `terminate_ecp`

**Definicja:**

```
BOOLEAN tight_coop_generator::first_step (
                                        list<sensor>* sensor_list,
                                        list<robot>* robot_list)
```

```
#include "mp.h"
```

**Opis:** Metoda `tight_coop_generator::first_step` określa czynności związane z generacją pierwszego kroku ruchu dla generatora wytwarzającego trajektorię prostoliniową przy zadanym przyroście położenia i orientacji wyrażonych we współrzędnych kartezjańsko-eulerowskich (ZYZ). W związku z tym generator ten przygotowuje do wysłania do ECP polecenia `NEXT_POSE` oraz wszelkie parametry ruchu. Niemniej jednak, dla celów interpolacji, wpierw trzeba odczytać aktualne położenie końcówki ramienia. Dlatego `first_step` formuje rozkaz odczytu `GET`. Następnie wywołuje `copy_generator_command`, aby skopiować polecenia przygotowane przez siebie do obrazów robotów znajdujących się na liście. Rozkazy te będą wysłane (do wykonania przez ECP) przez odpowiednie metody `execute_motion` znajdujące się w funkcji `Move`. Ponadto `first_step` ustawia `idle_step_counter` na 2. W wyniku tego `next_step` dwukrotnie powtórzy zlecenie odczytu, a tymczasem informacja o aktualnym stanie ramienia dotrze na poziom MP. Dopiero wtedy będzie można wyznaczyć parametry interpolacji.

**Rezultat:** Metoda `tight_coop_generator::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

- `true` – warunek końcowy nie jest spełniony (kontynuować ruch)
- `false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `tight_coop_generator::first_step` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `tight_coop_generator::first_step`.

**Zobacz:** `tight_coop_generator.next_step`

**Przykład:** Patrz tekst `Move` dla procesu MP.

**Definicja:**

```
BOOLEAN tight_coop_generator::next_step (
                                   list<sensor>* sensor_list,
                                   list<robot>* robot_list)
```

```
#include "mp.h"
```

**Opis:** Metoda `tight_coop_generator::next_step` określa czynności związane z generacją kolejnych kroków ruchu dla generatora wytwarzającego trajektorię prostoliniową przy zadanym przyroście położenia i orientacji wyrażonych we współrzędnych kartezjańsko-eulerowskich (ZYZ). W związku z tym generator ten przygotowuje do wysłania do ECP polecenia `NEXT_POSE` i `SET` oraz wszelkie parametry ruchu. Wpierw, dzięki zastosowaniu licznika `idle_step_counter` (ustawionego na 2 przez `first_step`) następuje oczekiwanie na odczyt aktualnego położenia końcówki. Potem sprawdza się, czy końcówka przeszła przez wszystkie węzły interpolacji. Jeżeli tak, to generowanie trajektorii jest kończone. Jeżeli nie, to następuje kopiowanie danych z buforów przysłowych (z ECP) do obrazów robotów wykorzystywanych przez generator. Sprawdza się również, czy aktualny przedział interpolacji jest ostatnim. Jeżeli tak, to zamiast polecenia `NEXT_POSE` wysłane zostanie polecenie `END_MOTION` do procesów ECP. Na koniec wywołuje się `copy_generator_command`, aby skopiować polecenia przygotowane przez `next_step` do obrazów robotów znajdujących się na liście. Rozkazy te będą wysłane (do wykonania przez ECP) przez odpowiednie metody `execute_motion` znajdujące się w funkcji `Move`.

**Rezultat:** Metoda `tight_coop_generator::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

`true` – warunek końcowy nie jest spełniony (kontynuować ruch)
`false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `tight_coop_generator::next_step` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `tight_coop_generator::next_step`.

**Zobacz:** `tight_coop_generator.first_step`

**Przykład:** Patrz tekst `Move` dla procesu MP.

**Definicja:** `tight_coop_generator::tight_coop_generator (void)`

`#include "mp.h"`

**Opis:** Metoda `tight_coop_generator::tight_coop_generator` tworzy i inicjuje obiekt klasy `tight_coop_generator`.

**Rezultat:** Metoda `tight_coop_generator::tight_coop_generator` nie zwraca żadnej wartości (jest konstruktorem obiektu).

**Błędy:** Metoda `tight_coop_generator::tight_coop_generator` nie zgłasza wyjątków.

**Zobacz:** `tight_coop_generator.first_step`, `tight_coop_generator.next_step`

**Definicja:**

```
BOOLEAN Wait (list<robot>* robot_list,
              list<sensor>* sensor_list,
              condition& the_condition)
```

```
#include "mp.h"
```

**Opis:** Funkcja `Wait` powoduje oczekiwanie na spełnienie warunku początkowego `the_condition`. Korzysta z informacji pochodzacych w czujników wirtualnych umieszczonych na liście `sensor_list` oraz aktualnego stanu efektorów umieszczonych na liście `robot_list`. Warunek początkowy określony jest specyfikacją zawartą w metodzie `the_condition.condition_value`. Na wstępie żądane są dane od wszystkich czujników umieszczonych na liście `sensor_list`. W kolejnej fazie wykonania funkcji `Wait` odczytywane są dane uzyskane przez wszystke czujniki umieszczone na liście `sensor_list`. Na koniec metoda `the_generator.condition_value` sprawdza warunek końcowy oraz jeżeli nie jest on spełniony ponawia powyższe operacje. W przeciwnym przypadku `Wait` kończy swe działanie. Sieć działań funkcji `Wait` przedstawiono na rysunku 4.2.

**Rezultat:** Funkcja `Wait` zwraca wartość `BOOLEAN`. Wartość ta określa tryb zakończenia ruchu, tzn.:

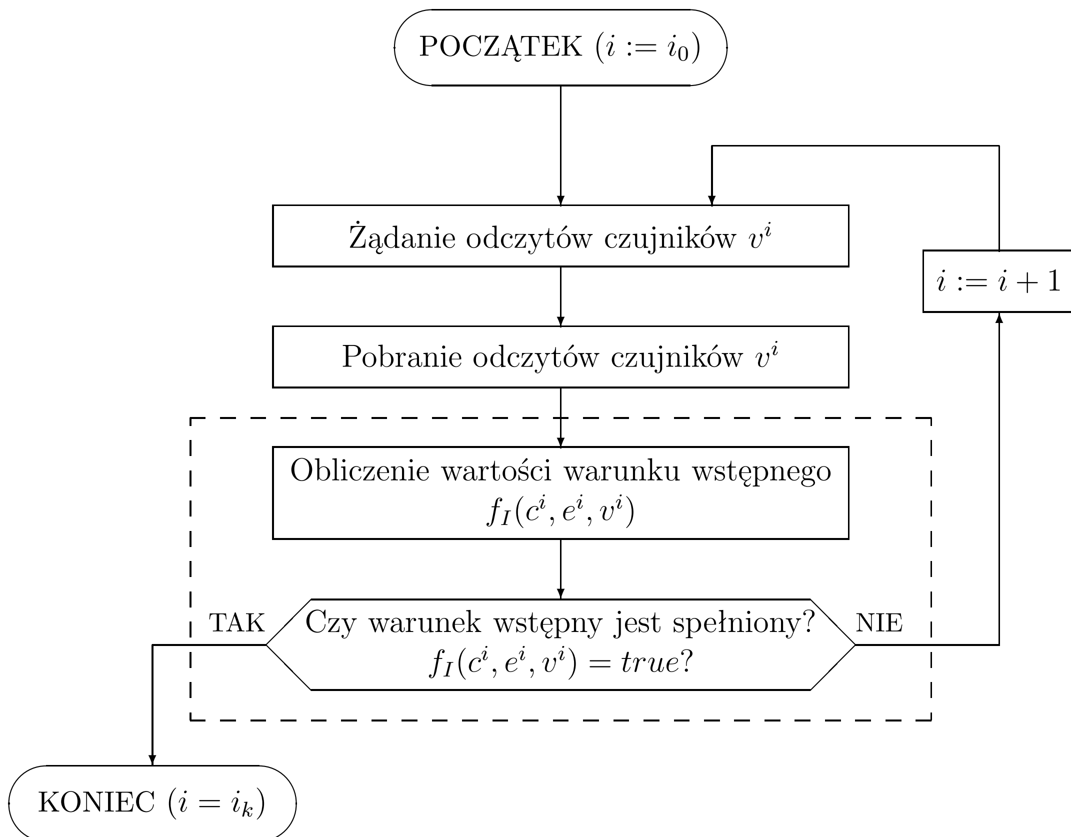
| | | |
|---|---|---|
| `true` | – | przerwano ruch na skutek polecenia operatora (`STOP`) |
| `false` | – | przerwano ruch na skutek spełnienia warunku końcowego |

**Błędy:** Funkcja `Wait` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są błędy systemowe `SYSTEM_ERROR`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez funkcję `Wait`.

**Tekst:**

```
BOOLEAN Wait(list<robot>* robot_list, list<sensor>* sensor_list,
             condition& the_condition)

 // Funkcja oczekiwania na spełnienie warunku początkowego
 // Funkcja zwraca FALSE, gdy samoistny koniec oczekiwania
 // Funkcja zwraca TRUE, gdy zakończenie wywołane jest przez STOP

 list<sensor>*  sensor_lptr;          // sensor list pointer
 list<robot>*   robot_lptr;           // robot list pointer
 pid_t pid;                           // identyfikator proxy
 unsigned long int e;

 do { // oczekiwanie
   // Sprawadzenie czy nie przyszło polecenie operatora
    if((pid = Creceive(MPProxyPids.stop, NULL, 0 ) ) != -1 ) {
      // Przesłanie polecenia STOP do ECP
      for ( robot_lptr = robot_list; robot_lptr; robot_lptr =
            robot_lptr->next ) {
        robot_lptr->E_ptr->terminate_ecp();
      } // end: for
      return TRUE;
    }
    else if (errno != ENOMSG) {
      // Błąd komunikacji miedzyprocesowej - wyjatek
      e = errno;
```

```
        perror("Creceive STOP proxy from UI failed ?\n");
        msg->message(SYSTEM_ERROR, e,
                     "MP: Creceive STOP proxy from UI failed");
        throw MP_main_error (SYSTEM_ERROR, (unsigned word32) 0);
      }

      if ( ( pid = Creceive(MPProxyPids.pause, NULL, 0) ) != -1 ) {
        msg->message("To resume MP click RESUME icon");
        // Oczekiwanie na RESUME
        if (Receive (MPProxyPids.resume, NULL, 0) == -1) {
          // Wystapił błąd systemowy przy Receive
          e = errno;
          perror("Creceive RESUME proxy from UI failed ?\n");
          msg->message(SYSTEM_ERROR, e,
                       "MP: Creceive RESUME proxy from UI failed");
          throw MP_main_error (SYSTEM_ERROR, (unsigned word32) 0);
        }
        msg->message("MP user program is running");
      }
      else
        if (errno != ENOMSG) {
          // Błąd komunikacji miedzyprocesowej - wyjatek
          e = errno;
          perror("Creceive PAUSE proxy from UI failed ?\n");
          msg->message(SYSTEM_ERROR, e,
                       "MP: Creceive PAUSE proxy from UI failed");
          throw MP_main_error (SYSTEM_ERROR, (unsigned word32) 0);
        }

     // żądanie danych od wszystkich czujników
     for (sensor_lptr = sensor_list; sensor_lptr;
          sensor_lptr = sensor_lptr->next)
       sensor_lptr->E_ptr->initiate_reading();

     // odczytanie danych z wszystkich czujników
     for (sensor_lptr = sensor_list; sensor_lptr;
          sensor_lptr = sensor_lptr->next)
       sensor_lptr->E_ptr->get_reading();

   } while ( !the_condition.condition_value(robot_list,sensor_list) );
   // end: do

   return FALSE;
  }; // end: Wait()
```




**Rys. 4.2:** Sieć działań instrukcji `Wait` z zaznaczonym zakresem działania obiektu `condition`

**Definicja:** `void wait_for_start(mpid_t StartProxy)`

```
#include "mp.h"
```

**Opis:** Funkcja `wait_for_start` realizuje oczekiwanie na polecenie operatora START przesyłane z poziomu UI. Polecenie jest przkazywane za pośrednictwem *proxy* `StartProxy`.

**Rezultat:** Funkcja `wait_for_start` nie zwraca żadnej wartości.

**Błędy:** Funkcja `wait_for_start` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem MP a UI

**Zobacz:** `wait_for_stop`

**Definicja:** `void wait_for_stop(mpid_t StopProxy)`

```
#include "mp.h"
```

**Opis:** Funkcja `wait_for_stop` realizuje oczekiwanie na polecenie operatora STOP przesyłane z poziomu UI. Polecenie jest przkazywane za pośrednictwem *proxy* `StopProxy`.

**Rezultat:** Funkcja `wait_for_stop` nie zwraca żadnej wartości.

**Błędy:** Funkcja `wait_for_stop` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem MP a UI

**Zobacz:** `wait_for_start`

## Rozdział 5

# Funkcje i metody procesu ECP

W tym rozdziale zgromadzono wszystkie funkcje i metody, które mogą być przydatne dla użytkownika (programisty systemu MRROC++) przy tworzeniu części programu użytkowego znajdującego się w procesie ECP. Zamieszczono zarówno metody, które użytkownik będzie stosował konstruując sterownik dedykowany swemu specyficznemu zadaniu, jak i metody, które nie będą bezpośrednio przez użytkownika wykorzystywane w jego programach, ale są niezbędne dla zrozumienia sposobu działania tych pierwszych. Większość z zamieszczonych metod klas wywiedzionych z klasy `generator` stanowi przykład (wzór), który można wykorzystać przy tworzeniu swych własnych generatorów. Niemniej jednak były one stosowane do realizacji wielu zadań, więc mogą być także wykorzystane przez użytkownika dokładnie w przedstawionej postaci.

Funkcje i metody zostały wymienione w porządku alfabetycznym. W przypadku metod wpierw uwzględniono nazwę klasy, z której pochodzą.

**Definicja:** `bin_sensor::bin_sensor( nid_t VSP_node, char *VSP_program)`

```
#include "ecp.cc"
```

**Opis:** Konstruktor `bin_sensor::bin_sensor` tworzy i incjuje obiekt klasy `bin_sensor` co oznacza odczytanie programu procesu VSP z pliku `*VSP_program` i utworzenie procesu wykonującego ten program. Po zakończeniu działania proces potomny VSP nie będzie generował sygnału `SIGHOLD`.

**Rezultat:** Metoda `bin_sensor::bin_sensor` nie zwraca żadnej wartości.

**Błędy:** Funkcja `bin_sensor::bin_sensor` reaguje na wystąpienie błędu utworzenia procesu potomnego zgłoszeniem wyjątku.

Generowany jest błąd systemowy:
`SYSTEM_ERROR` – błąd: brak połączenia ECP -VSP.

**Definicja:** `void bin_sensor::create_sensor_command (void)`

```
#include "ecp.cc"
```

**Opis:** Metoda `bin_sensor::create_sensor_command` wypełnia bufor wysyłkowy `msg_ecp_vsp` wysyłany do VSP zgodnie z informacją zawartą w polu kodu instrukcji "obrazu" czujnika przechowywanego w ECP, to znaczy w polu `image.instruction_code`.

**Rezultat:** Metoda `create_sensor_command` nie zwraca żadnej wartości.

**Błędy:** Funkcja `create_sensor_command` reaguje na wystąpienie błędu w polu kodu instrukcji (nierozpoznawalny kod) zgłoszeniem wyjątku.

Generowany jest błąd niefatalny:

`NON_FATAL_ERROR` – błąd `INVALID_VSP_COMMAND` informuje, że przygotowano błędny rozkaz dla VSP.

**Zobacz:** `bin_sensor::get_reading`, `bin_sensor::get_sensor_reply`

**Definicja:** `void bin_sensor::get_reading (void)`

```
#include "ecp.cc"
```

**Opis:** Metoda `get_reading` korzysta z metody `create_sensor_command` w celu wypełnienia bufora wysyłkowego `msg_ecp_vsp` przesyłanego do VSP, następnie przesyła ten bufor i odbiera bufor `msg_vsp_ecp`. W przypadku wystąpienia niepowodzenia przesłania generowany jest wyjątek `ECP_SYSTEM_ERROR`. Na koniec, w celu przetworzenia otrzymanego bufora, wywoływana jest metoda `get_sensor_reply`.

**Rezultat:** Metoda `get_reading` nie zwraca żadnej wartości.

**Błędy:** Funkcja `get_reading` reaguje na wystąpienie błędu związanego z przesłaniem do procesu potomnego VSP zgłoszeniem wyjątku.

Generowany jest błąd:

`ECP_SYSTEM_ERROR` – w tym przypadku przesyłka ECP–VSP nie powiodła się.

**Zobacz:** `bin_sensor::get_sensor_reply`, `bin_sensor::create_sensor_command`

**Definicja:** `void bin_sensor::get_sensor_reply (void)`

```
#include "ecp.cc"
```

**Opis:** Metoda `get_sensor_reply` przetwarza bufor `msg_vsp_ecp` otrzymany z VSP. Jeżeli w polu `msg_vsp_ecp.vsp_report` zawarta jest informacja o prawidłowym odczycie (`VSP_reply_OK`) to do pola `image.reading_type` obrazu czujnika (czujników) przepisywana jest informacja o typie odebranych danych a do odpowiednich pól danych obrazu czujnika przepisywane są odczyty z czujników zawarte w odpowiednich polach bufora odebranego z VSP. Jest to `vsp_reply_buffer`. W przypadku braku informacji o poprawnym odczycie lub przy innych błędach generowane sa odpowiednie wyjątki.

**Rezultat:** Metoda `get_sensor_reply` nie zwraca żadnej wartości.

**Błędy:** Funkcja `get_sensor_reply` reaguje na wystąpienie danych o nierozpoznawalnym typie (w polu bufora określającym typ danych jest nierozpoznawalny kod typu), obsługiwane są też błędy związane z czujnikami i zgłoszone w polu `msg_vsp_ecp.vsp_report`. We wszystkich tych przypadkach generowane są wyjątki. Jeżeli w polu wystąpi `msg_vsp_ecp.vsp_report` nierozpoznawalny kod, generowany jest też odpowiedni wyjątek.

Generowane są wyjątki:

| | | |
|---|---|---|
| `NON_FATAL_ERROR` | – | w pierwszym omówionym przypadku jest to `INVALID_VSP_READING_TYPE`, czyli błędny typ danych, |
| `NON_FATAL_ERROR` | – | w drugim omówionym przypadku są to błędy związane z czujnikami zgłoszone przez VSP: `SENSOR_ERROR_1, SENSOR_ERROR_2, INVALID_VSP_READING_TYPE`, |
| `NON_FATAL_ERROR` | – | w trzecim przypadku jest to nierozpoznawalny kod zgłoszony jako błąd: `INVALID_VSP_REPLY`. |

**Zobacz:** `bin_sensor::get_reading, bin_sensor::create_sensor_command`

**Definicja:** `void bin_sensor::terminate (void)`

```
#include "ecp.cc"
```

**Opis:** Metoda `bin_sensor::terminate` wpisuje do pola obrazu czujnika `image.instruction_code` kod `VSP_TERMINATE`. Kod ten jest w metodzie `create_sensor_command` przepisywany do bufora wysyłkowego do VSP. Proces VSP, po odebraniu tego kodu, kończy swoje działanie.

**Rezultat:** Metoda `terminate` nie zwraca żadnej wartości.

**Zobacz:** `bin_sensor::create_sensor_command`

**Definicja:**

```
BOOLEAN generator_t::first_step (
                        list<sensor>* sensor_list,
                        robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `generator_t::first_step` określa czynności związane z generacją pierwszego kroku ruchu. Generator `generator_t` czyni ECP przeźroczystym. Faktyczna generacja trajektorii odbywa się w MP. Dlatego metoda ta ogranicza się do wywołania `generator_t::next_step`.

**Rezultat:** Metoda `generator_t::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

- `true` – warunek końcowy nie jest spełniony (kontynuować ruch)
- `false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `generator_t::first_step` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `generator_t::first_step`.

**Zobacz:** `generator_t::next_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `generator_t::generator_t (void)`

`#include "ecp.h"`

**Opis:** Metoda `generator_t::generator_t` tworzy i inicjuje obiekt klasy `generator_t`.

**Rezultat:** Metoda `generator_t::generator_t` nie zwraca żadnej wartości (jest konstruktorem obiektu).

**Błędy:** Metoda `generator_t::generator_t` nie zgłasza wyjątków.

**Zobacz:** `generator_t.first_step`, `generator_t.next_step`

**Definicja:**

```
BOOLEAN generator_t::next_step (
                     list<sensor>* sensor_list,
                     robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `generator_t::next_step` określa czynności związane z generacją kolejnych kroków ruchu. Generator `generator_t` czyni ECP przeźroczystym. Faktyczna generacja trajektorii odbywa się w MP. Na wstępie odbywa się kopiowanie danych z bufora przysłanego z EDP do obrazu robota wykorzystywanego przez generator. Następnie pobierane jest polecenie otrzymane z MP. Jeżeli jest to `NEXT_POSE`, to następuje przepisanie do bufora poleceń przesyłanego do EDP przysłanej z MP kolejnej pozycji oraz zwrócenie `true`. Jeżeli jest to `END_MOTION`, to zwracane jest `false`. Gdy jest to `STOP` lub wykryto sytuację awaryjną, zgłaszany jest odpowiedni wyjątek.

**Rezultat:** Metoda `generator_t::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

| | | |
|---|---|---|
| `true` | – | warunek końcowy nie jest spełniony (kontynuować ruch) |
| `false` | – | warunek końcowy jest spełniony (przerwać ruch) |

**Błędy:** Metoda `generator_t::next_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_MP_COMMAND`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `generator_t::next_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `generator_t.first_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `int input_double(char* question )`

```
#include "ecp.h"
```

**Opis:** Funkcja `input_double` służy do wysłania żądania `question` wprowadzenia przez operatora liczby rzeczywistej. Po nawiązaniu połączenia pomiędzy procesem ECP i UI wyświetlane jest okienko umożliwiające wprowadzenie przez operatora tylko liczby rzeczywistej.

**Rezultat:** Funkcja zwraca wprowadzoną przez operatora liczbę całkowitą.

**Błędy:** Funkcja `input_double` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem ECP i UI

**Zobacz:** `operator_reaction`, `input_integer`, `show_message`

**Definicja:** `int input_integer(char* question )`

```
#include "ecp.h"
```

**Opis:** Funkcja `input_integer` służy do wysłania żądania `question` wprowadzenia przez operatora liczby całkowitej. Po nawiązaniu połączenia pomiędzy procesem ECP i UI wyświetlane jest okienko umożliwiające wprowadzenie przez operatora tylko liczby całkowitej.

**Rezultat:** Funkcja zwraca wprowadzoną przez operatora liczbę całkowitą.

**Błędy:** Funkcja `input_integer` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem ECP i UI

**Zobacz:** `operator_reaction`, `input_double`, `show_message`

**Definicja:**

```
BOOLEAN linear_generator::first_step (
                            list<sensor>* sensor_list,
                            robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `linear_generator::first_step` określa czynności związane z generacją pierwszego kroku ruchu dla trajektorii prostoliniowej o zadany przyrost polozenia i orientacji. Wpierw odbierany jest kolejne polecenie MP oraz przesyłane jest zwrotnie `ECP_ACKNOWLEDGE`. Następnie metoda analizuje polecenie przysłane przez MP. Jeżeli jest to `NEXT_POSE`, to formowany jest rozkaz dla EDP. Jest to `GET ARM XYZ_EULER_ZYZ`, czyli żądanie odczytu aktualnego położenia ramienia robota we współrzędnych kartezjańskich oraz kątach Eulera (ZYZ). Na podstawie odczytanej aktualnej pozycji ramienia oraz pozycji zadanej wyznaczone będą przez metodę `linear_generator::next_step` przedziały interpolacji, czyli makrokroki do realizacji przez EDP. Polecenie składowane jest w obrazie robota. Następnie metoda `the_robot.create_command` przepisuje polecenie z obrazu do bufora przesyłkowego do procesu EDP. Polecenie operatora `STOP` oraz sytuacje awaryjne sygnalizowane są przez zgłoszenie odpowiednich wyjątków.

**Rezultat:** Metoda `linear_generator::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

| | | |
|---|---|---|
| `true` | – | warunek końcowy nie jest spełniony (kontynuować ruch) |
| `false` | – | warunek końcowy jest spełniony (przerwać ruch) |

**Błędy:** Metoda `linear_generator::first_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_MP_COMMAND`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `linear_generator::first_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `linear_generator.next_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `linear_generator::linear_generator (void)`

`#include "ecp.h"`

**Opis:** Metoda `linear_generator::linear_generator` tworzy i inicjuje obiekt klasy `linear_generator`.

**Rezultat:** Metoda `linear_generator::linear_generator` nie zwraca żadnej wartości (jest konstruktorem obiektu).

**Błędy:** Metoda `linear_generator::linear_generator` nie zgłasza wyjątków.

**Zobacz:** `linear_generator.first_step`, `linear_generator.next_step`

**Definicja:**

```
BOOLEAN linear_generator::next_step (
                              list<sensor>* sensor_list,
                              robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `linear_generator::next_step` określa czynności związane z generacją kolejnych kroków ruchu dla trajektorii prostoliniowej o zadany przyrost polozenia i orientacji. Przy przechodzeniu do kolejnego węzła interpolacji metoda kontaktuje się z MPprzekazując `ECP_ACKNOWLEDGE`. Ponadto kopiuje danye z bufora przysłanego z EDP do obrazu robota wykorzystywanego przez generator. Po wykonaniu tych czynności następuje obliczenie zadanej pozycji pośredniej w tym kroku ruchu (określenie kolejnego węzła interpolacji) oraz uformowanie nowego polecenia dla EDP. Na zakończenie kopiowany jest tak przygotowany rozkaz do bufora wysyłkowego do EDP.

**Rezultat:** Metoda `linear_generator::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

| | | |
|---|---|---|
| `true` | – | warunek końcowy nie jest spełniony (kontynuować ruch) |
| `false` | – | warunek końcowy jest spełniony (przerwać ruch) |

**Błędy:** Metoda `linear_generator::next_step` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `linear_generator::next_step`.

**Zobacz:** `linear_generator.first_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `BOOLEAN load_file(teach_in_generator& the_generator)`

```
#include "ecp.h"
```

**Opis:** Funkcja realizuje odczyt trajektorii z pliku tekstowego (ASCII). Trajektoria jest zapisana w pliku w postaci ciągu pozycji (patrz opis funkcji `save_file`). Wywołanie funkcji powoduje nawiązanie komunikacji między procesem ECP oraz procesem UI. Następnie z ECP jest wysłane do UI polecenie `LOAD_FILE`, które powoduje wyświetlenie okienka zawierającego nazwy plików z zapisanymi wcześniej trajektoriami. Użytkownik może wybrać plik poprzez wpisanie jego nazwy lub kliknięcie myszką jego nazwy. Domyślnie wyświetlana jest zawartość bieżącego katalogu (zazwyczaj jest to `../mrrocpp/bin`), ale można zmienić ścieżkę dostępu do plików, wybierając inny katalog (opcja w interfejsie okienkowym).

**Rezultat:** Funkcja `load_file` zwraca wartość typu `BOOLEAN`:

`true` – odczytano pomyślnie dane z pliku
`false` – nie wybrano żadnego pliku

**Błędy:** Funkcja `load_file` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne:

| | | |
|---|---|---|
| `NON_EXISTENT_DIRECTORY` | – | nie istnieje katalog o podanej nazwie, z którego ma być odczytany plik |
| `NON_EXISTENT_FILE` | – | w bieżącym katalogu nie ma pliku o podanej nazwie, |
| `READ_FILE_ERROR` | – | błąd odczytu z pliku, |
| `NON_TRAJECTORY_FILE` | – | czytany plik nie zawiera trajektorii (np. plik ma błędną strukturę danych). |

Ponadto jest wykrywany błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem ECP i UI

Wyjątki mogą być również zgłaszane przez funkcje lub metody wywoływane przez funkcję `load_file`.

**Zobacz:** `save_file`

**Przykład:** Patrz przykład dla funkcji `teach`.

**Definicja:**

```
void Move (robot& the_robot,
           list<sensor>* sensor_list,
           generator& the_generator)
```

```
#include "ecp.h"
```

**Opis:** Funkcja `Move` realizuje ruch effektora `the_robot` z wykorzystaniem informacji pochodzacych w czujników wirtualnych umieszczonych na liście `sensor_list`. Ruch jest generowany zgodnie ze specyfikacją określoną przez `the_generator`. Pierwszy kroku ruchu określony jest przez metodę `the_generator.first_step`. Często sprowadza się on do odczytu aktualnego położenia efektora (robota). Następnie żądane są dane od wszystkich czujników umieszczonych na liście `sensor_list`. Po tym zlecane jest wykonanie kroku ruchu przez metodę `the_robot.execute_motion`. W kolejnej fazie wykonania funkcji `Move` odczytywane są dane uzyskane przez wszystke czujniki umieszczone na liście `sensor_list`. Na koniec metoda `the_generator.next_step` sprawdza warunek końcowy oraz jeżeli nie jest on spełniony oblicza kolejny krok ruchu, po czym powyższe operacje są ponawiane. W przeciwnym przypadku `Move` kończy swe działanie. Sieć działań funkcji `Move` przedstawiono na rysunku 5.1.

**Rezultat:** Funkcja `Move` nie zwraca żadnej wartości (jest procedurą).

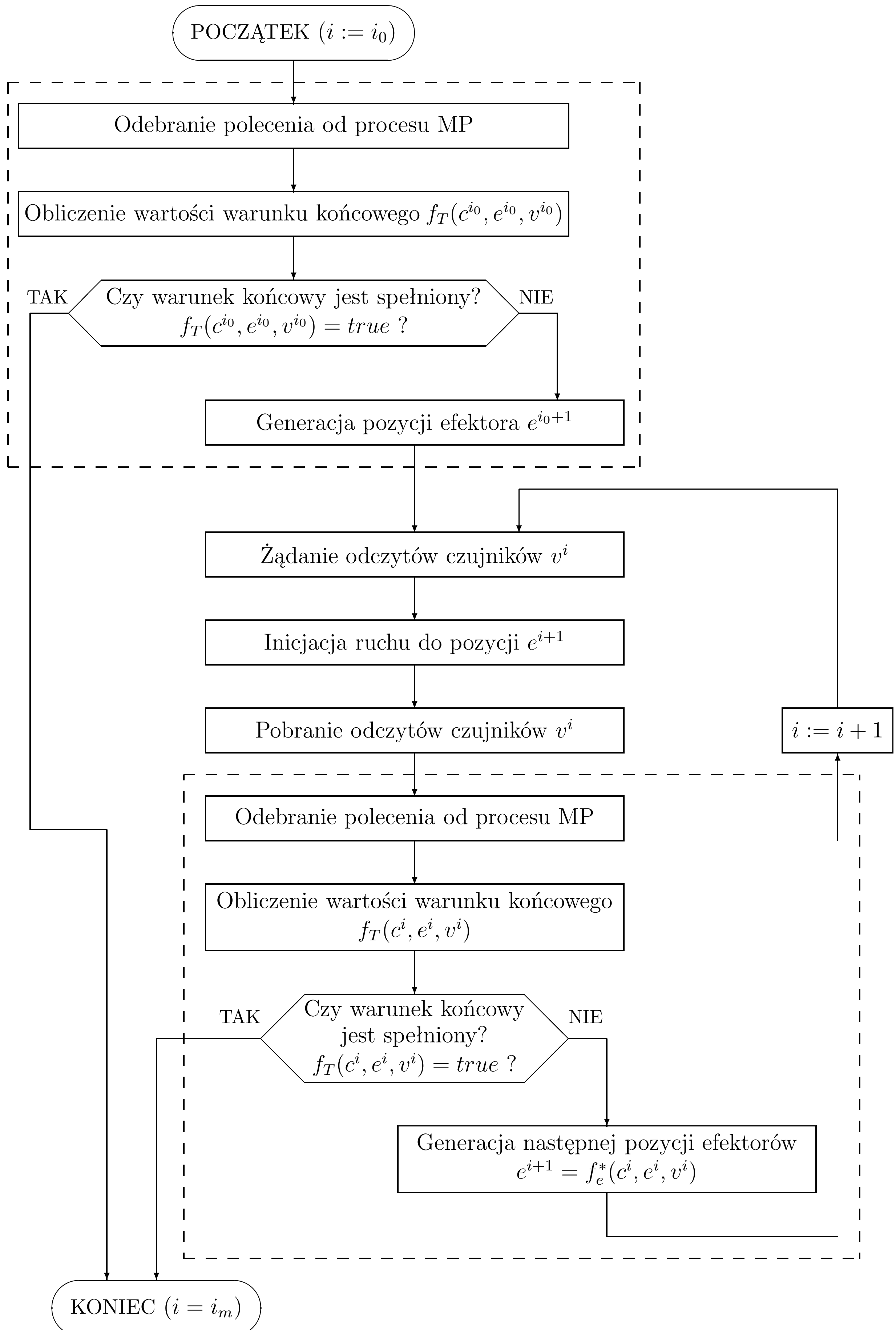
**Błędy:** Funkcja `Move` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez funkcję `Move`.

**Zobacz:** `teach`

**Przykład:** Patrz przykład dla funkcji `teach`.

**Tekst:**

```
void Move(robot& the_robot, list<sensor>* sensor_list,
          generator& the_generator) {
// Instrukcja ruchu dla ECP

 list<sensor>* sensor_lptr = NULL;  // wskazuje aktualnie przetwarzany
                                    // element listy czujników

 // generacja pierwszego kroku ruchu
  if (!the_generator.first_step(sensor_list, the_robot) )
    return; // Warunek końcowy spełniony w pierwszym kroku

 do { // realizacja ruchu

   // żądanie danych od wszystkich czujników
   for (sensor_lptr = sensor_list; sensor_lptr;
        sensor_lptr = sensor_lptr->next)
     sensor_lptr->E_ptr->initiate_reading();

   // wykonanie kroku ruchu,
   // czyli zlecenie ruchu "SET" oraz odczyt stanu robota "GET"
   the_robot.execute_motion();

   // odczytanie danych z wszystkich czujników
   for (sensor_lptr = sensor_list; sensor_lptr;
```




**Rys. 5.1:** Sieć działań instrukcji `Move` w procesie ECP, z zaznaczonym zakresem działania metod `generator.first_step` i `generator.next_step`

```
        sensor_lptr = sensor_lptr->next)
     sensor_lptr->E_ptr->get_reading();

 } while ( the_generator.next_step(sensor_list, the_robot) );

}; // end: Move()
```

**Definicja:**

```
BOOLEAN off_line_generator::first_step (
                                list<sensor>* sensor_list,
                                robot& the_robot)
#include "ecp.h"
```

**Opis:** Metoda `off_line_generator::first_step` określa czynności związane z generacją pierwszego kroku ruchu. Generator ma za zadanie odtwarzać pozycje z listy, która została stworzona na podstawie zawartości uprzednio wczytanego pliku tekstowego. Ruch między kolejnymi pozycjami listy odbywa się ze stałą prędkością. Kontakt z MP nastepuje przed rozpoczęciem ruchu do kolejnej pozycji na liscie. Metoda `off_line_generator::first_step` przygotowuje rozkaz odczytu aktualnej pozycji narzędzia we współrzędnych zewnętrznych w celu aktualizacji danych w `transformer` w EDP. Poprzednie ruchy mogły odbywać się we współrzędnych innych niż zewnętrzne, a więc odpowiednie dane nie były aktualizowane.

**Rezultat:** Metoda `off_line_generator::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

| | | |
|---|---|---|
| `true` | – | warunek końcowy nie jest spełniony (kontynuować ruch) |
| `false` | – | warunek końcowy jest spełniony (przerwać ruch) |

**Błędy:** Metoda `off_line_generator::first_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_MP_COMMAND`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `generator_t::first_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `off_line_generator::load_frames`, `off_line_generator::next_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `BOOLEAN off_line_generator::load_frames ( void )`

`#include "ecp.h"`

**Opis:** Metoda `off_line_generator::load_frames` powoduje wczytanie trajektorii z pliku tekstowego. Każdy wiersz tego pliku zawiera trzynaście liczb rzeczywistych. Pierwszą stanowi czas dojścia do pozycji, która określona jest przez kolejnych dwanaście liczb. Liczby te stanowią definicję macierzy jednorodnej opisującej pozycje narzędzia względem układu odniesienia. Układ ten może być zarówno globalnym jak i lokalnym układem odniesienia – zależy to od tego czy w `off_line_generator::next_step` korzysta się z transformacji wykonywanej przez `transform_frame`. Pierwsze trzy trójki liczb stanowią wersory odpowiednio osi $X$, $Y$ i $Z$ układu związanego z narzędziem wyrażone w układzie odniesienia. Ostatnia trójka liczb to wektor określający początek układu narzędzia względem układu odniesienia.

**Rezultat:** Metoda `off_line_generator::load_frames` zwraca wartość `BOOLEAN`:

- `true` – jeśli wczytanie trajektorii powiodło się
- `false` – w przeciwnym przypadku

**Błędy:** Metoda `off_line_generator::load_frames` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne:
`NON_EXISTENT_DIRECTORY`,
`NON_EXISTENT_FILE`,
`READ_FILE_ERROR`.
W przypadku braku powodzenia w kontakcie z procesem UI zgłaszany jest błąd systemowy `SYSTEM_ERROR`.

**Zobacz:** `off_line.first_step, off_line.next_step`

**Definicja:**

```
BOOLEAN off_line_generator::next_step (
                              list<sensor>* sensor_list,
                              robot& the_robot)

#include "ecp.h"
```

**Opis:** Metoda `off_line_generator::next_step` określa czynności związane z generacją kolejnych kroków ruchu. Generator ma za zadanie odtwarzać pozycje z listy, która została stworzona na podstawie zawartości uprzednio wczytanego pliku tekstowego. Ruch między kolejnymi pozycjami listy odbywa się ze stałą prędkością. Zakłada się, że pozycje na liście umieszczone są dostatecznie gęsto by powstała gładka trajektoria. Kontakt z MP nastepuje przed rozpoczęciem ruchu do kolejnej pozycji na liscie.

Konstruktor obiektu tej klasy definuje macierz jednorodną `local2global` służącą do ewentualnej transformacji wczytanych pozycji z lokalnego do globalnego układu odniesienia. Możliwość ta musi być aktywowana w `off_line_generator::next_step`. W tym celu należy wywołać `off_line_generator::transform_frame`.

**Rezultat:** Metoda `off_line_generator::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:
- `true` – warunek końcowy nie jest spełniony (kontynuować ruch)
- `false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `off_line_generator::next_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywa błąd niefatalny `INVALID_MP_COMMAND`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `off_line_generator::next_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `off_line_generator::load_frames`, `off_line_generator::first_step`, `off_line_generator::off_line_generator`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `off_line_generator::off_line_generator (void)`

`#include "ecp.h"`

**Opis:** Metoda `off_line_generator::off_line_generator` stanowi konstruktor generatora odtwarzajacego pozycje bez rozpedzania i hamowania miedzy pozycjami oraz dodatkowej interpolacji. Pozycje te są wczytywane z pliku tekstowego i umieszczane na liście metodą `off_line_generator::load_frames`. Konstruktor definuje macierz jednorodną `local2global` służącą do ewentualnej transformacji wczytanych pozycji z lokalnego do globalnego układu odniesienia.

**Rezultat:** Metoda `off_line_generator::off_line_generator` nie zwraca żadnej wartości (jest konstruktorem obiektu).

**Błędy:** Metoda `off_line_generator::off_line_generator` nie zgłasza wyjątków.

**Zobacz:** `off_line_generator::first_step`, `off_line_generator::next_step`

**Definicja:** `off_line_generator::∼off_line_generator (void)`

`#include "ecp.h"`

**Opis:** Metoda `off_line_generator::∼off_line_generator` stanowi destruktor generatora odtwarzajacego pozycje bez rozpedzania i hamowania miedzy pozycjami oraz dodatkowej interpolacji. Destrukcja polega na usunięciu z pamięci wczytanej listy pozycji.

**Rezultat:** Metoda `off_line_generator::∼off_line_generator` nie zwraca żadnej wartości (jest destruktorem obiektu).

**Błędy:** Metoda `off_line_generator::∼off_line_generator` nie zgłasza wyjątków.

**Zobacz:** `off_line_generator::first_step`, `off_line_generator::next_step`

**Definicja:** `BOOLEAN operator_reaction(char* question)`

`#include "ecp.h"`

**Opis:** Funkcja `operator_reaction` służy do wywołania reakcji operatora i uzyskania od niego odpowiedzi na pytanie `question`. Odpowiedź jest postaci TAK/NIE (YES/NO). Wywołanie funkcji powoduje nawiązanie komunikacji między procesem ECP i UI, a następnie wyświetlenie okienka z pytaniem i oczekiwanie na odpowiedź opertora. Funkcja jest wykorzystywana, jeśli w trakcie wykonywania procesu ECP potrzebna jest interwencja operatora.

**Rezultat:** Metoda `operator_reaction` zwraca wartość typu `BOOLEAN`:

`true` – odpowiedź TAK (`ANSWER_YES`)
`false` – odpowiedź NIE

**Błędy:** Funkcja `operator_reaction` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem ECP i UI

**Zobacz:** `input_integer, input_double, show_message`

**Przykład:** Patrz przykład dla funkcji `teach`.

**Definicja:**

```
BOOLEAN parabolic_teach_in_generator::first_step (
                                 list<sensor>* sensor_list,
                                 robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `parabolic_teach_in_generator::first_step` określa czynności związane z generacją pierwszego kroku ruchu dla generatora odtwarzającego uprzednio nauczone pozycje. Ruch między kolejnymi pozycjami listy ma trójkątny profil prędkości. Kontakt z MP następuje tylko w momencie osiągnięcia kolejnej pozycji na liscie. Wpierw odbierany jest kolejne polecenie MP oraz przesyłane jest zwrotnie `ECP_ACKNOWLEDGE`. Następnie metoda sprawdza, czy wskaźnik bieżący listy pozycji nauczonych równa się `NULL` – lista pusta. Jezeli tak, to kończona jest generacja trajektorii, co sygnalizowane jest przez zwrócenie `false`. W przeciwnym przypadku analizowane jest polecenie przysłane przez MP. Jeżeli jest to `NEXT_POSE`, to formowany jest rozkaz dla EDP. Jest to `GET ARM`, czyli żądanie odczytu aktualnego położenia ramienia robota. W zależności od tego w jaki sposób zostało wyrażone położenie ramienia na liście pozycji nauczonych wybierany jest jeden z parametrów `MOTOR`, `JOINT`, `XYZ_EULER_ZYZ`, `XYZ_ANGLE_AXIS`. Na podstawie odczytanej aktualnej pozycji ramienia oraz bieżącej pozycji na liście wyznaczone będą przedziały interpolacji, czyli makrokroki do realizacji przez EDP. Instrukcją `first_interval = TRUE` zaznacza się, że będzie realizowany pierwszy przedział interpolacji, więc metoda `parabolic_teach_in_generator.next_step` będzie musiała wyznaczyć parametry ruchu.

Polecenie składowane jest w obrazie robota. Następnie metoda `the_robot.create_command` przepisze polecenie z obrazu do bufora przesyłkowego do procesu EDP. Polecenie operatora `STOP` oraz sytuacje awaryjne sygnalizowane są przez zgłoszenie odpowiednich wyjątków.

**Rezultat:** Metoda `parabolic_teach_in_generator::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

| | | |
|---|---|---|
| `true` | – | warunek końcowy nie jest spełniony (kontynuować ruch) |
| `false` | – | warunek końcowy jest spełniony (przerwać ruch) |

**Błędy:** Metoda `parabolic_teach_in_generator::first_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_MP_COMMAND`, `INVALID_POSE_SPECIFICATION`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `parabolic_teach_in_generator::first_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `parabolic_teach_in_generator::parabolic_teach_in_generator`, `parabolic_teach_in_generator.next_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:**

```
BOOLEAN parabolic_teach_in_generator::next_step (
        list<sensor>* sensor_list, robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:**

Metoda `parabolic_teach_in_generator::next_step` określa czynności związane z generacją kolejnych kroków ruchu dla generatora odtwarzającego uprzednio nauczone pozycje. Ruch między kolejnymi pozycjami listy ma trójkątny profil prędkości. Kontakt z MP następuje tylko w momencie osiągnięcia kolejnej pozycji na liscie. Wpierw analizowane jest pole `first_interval`. Jeżeli ma ono wartość `true`, to następuje przepisanie do obrazu robota danych otrzymanych z EDPoraz wyznaczenie liczby przedziałów interpolacji oraz pozostałych parametrów ruchu. Wyznaczane są przyspieszenia oraz sprawdza się, czy nie zostały przekroczone wartości maksymalne przyspieszeń i predkości. Obliczeń tych nie można wykonać w `parabolic_teach_in_generator.first_step`, gdyż wtedy odczyt aktualnego położenia ramienia jeszcze nie zostanie zrealizowany. Zrobi to dopiero `execute_motion` po wyjściu z `first_step`. W zależności od tego w jaki sposób zostało wyrażone położenie ramienia na liście pozycji nauczonych wybierany jest jeden z parametrów `MOTOR`, `JOINT`, `XYZ_EULER_ZYZ`, `XYZ_ANGLE_AXIS`. Następnie formowany jest rozkaz ruchu dla EDPoraz pole `first_interval` przestawiane jest na `false`. Jednocześnie licznik węzłów interpolacyjnych `node_counter` (pomiędzy kolejnymi pozycjami nauczonymi) nastawiany jest na zero.

Jeżeli przy wejściu do `next_step` `first_interval` ma wartość `false`, to powyższe czynności są opuszczane i obliczane jest nowe położenie ramienia na podstawie danych interpolacyjnych (zgodnie z upływem czasu). W zależności od tego w jaki sposób zostało wyrażone położenie ramienia na liście pozycji nauczonych wybierany jest jeden z parametrów `MOTOR`, `JOINT`, `XYZ_EULER_ZYZ`, `XYZ_ANGLE_AXIS`. Następnie formowany jest rozkaz ruchu dla EDP. Jeżeli licznik węzłów interpolacyjnych `node_counter` będzie równy liczbie przedziałów interpolacji, to przestawiony zostanie wskaźnik elementów listy pozycji nauczonych oraz zwrócone zostanie `false`. W przeciwnym przypadku `the_robot.create_command` przepisze polecenie z obrazu do bufora przesyłkowego do procesu EDPoraz zwrócone zostanie `true`. Polecenie operatora `STOP` oraz sytuacje awaryjne sygnalizowane są przez zgłoszenie odpowiednich wyjątków.

**Rezultat:**

Metoda `parabolic_teach_in_generator::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

- `true` – warunek końcowy nie jest spełniony (kontynuować ruch)
- `false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:**

Metoda `parabolic_teach_in_generator::next_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_POSE_SPECIFICATION`, `MAX_ACCELERATION_EXCEEDED`, `MAX_VELOCITY_EXCEEDED`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `next_step`.

**Zobacz:** `parabolic_teach_in_generator::parabolic_teach_in_generator`, `parabolic_teach_in_generator::first_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `parabolic_teach_in_generator::parabolic_teach_in_generator (void)`

`#include "ecp.h"`

**Opis:** Metoda `parabolic_teach_in_generator::parabolic_teach_in_generator` stanowi konstruktor generatora odtwarzajacego nauczone pozycje. Założono, że między kolejnymi nauczonymi pozycjami ramie będzie rozpędzane i hamowane według funkcji parabolicznej (pozycja zmienia się jak kwadratowa funkcja czasu, natomiast profil prędkości jest trójkątny). Jednocześnie zadaniem konstruktora jest wypełnienie tablic określających maksymalne dopuszczalne przyspieszenia i prędkości ruchu.

**Rezultat:** Metoda `parabolic_teach_in_generator::parabolic_teach_in_generator` nie zwraca żadnej wartości (jest konstruktorem obiektu).

**Błędy:** Metoda `parabolic_teach_in_generator::parabolic_teach_in_generator` nie zgłasza wyjątków.

**Zobacz:** `parabolic_teach_in_generator.first_step`, `parabolic_teach_in_generator.next_step`

**Definicja:** `void rnt_robot::create_command (void)`

`#include "ecp.h"`

**Opis:** Metoda `rnt_robot::create_command` wypełnia bufor wysyłkowy do procesu EDP na podstawie danych zawartych w obrazie robota. Aby poprawnie wypełnić bufor wysyłkowy, analizowane jest polecenie, które ma być przesłane.

**Rezultat:** Metoda `rnt_robot::create_command` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `rnt_robot::create_command` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne:
`INVALID_POSE_SPECIFICATION`,
`INVALID_RMODEL_TYPE`,
`INVALID_ECP_COMMAND`.

**Definicja:** `void rnt_robot::ecp_termination_notice(void)`

`#include "ecp.h"`

**Opis:** Metoda `rnt_robot::ecp_termination_notice` służy do powiadomienia procesu MP o zakończeniu zadania użytkownika (wykonywanego w ECP).

**Rezultat:** Metoda `rnt_robot::ecp_termination_notice` nie zwraca żadnej wartości.

**Błędy:** Funkcja `rnt_robot::ecp_termination_notice` nie wykrywa błędów, lecz funkcje i metody przez nią wołane mogą zgłaszać wyjątki.

**Zobacz:** `rnt_robot::get_mp_command`

**Przykład:** Patrz przykład dla funkcji `teach`.

**Definicja:** `BOOLEAN rnt_robot::ecp_wait_for_start(void)`

```
#include "ecp.h"
```

**Opis:** Metoda `rnt_robot::ecp_wait_for_start` realizuje oczekiwanie w procesie ECP na polecenie START od procesu MP. Proces ECP jest zawieszony do chwili przyjścia polecenia. Jeśli otrzyma polecenie START_TASK odpowiada wówczas ECP_ACKNOWLEDGE w przeciwnym przypadku odpowiada INCORRECT_MP_COMMAND.

**Rezultat:** Metoda `rnt_robot::ecp_wait_for_start` zwraca wartość typu `BOOLEAN`:
`false` – jeśli odebrano polecenie START_TASK

**Błędy:** Funkcja `rnt_robot::ecp_wait_for_start` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd niefatalny:
`INVALID_MP_COMMAND` – otrzymano niedozwolone w tym miejscu polecenie (inne niż START_TASK)

Ponadto jest wykrywany błąd systemowy:
`SYSTEM_ERROR` – błąd w komunikacji między procesem ECP i UI

Wyjątki mogą być również zgłaszane przez funkcje lub metody wywoływane przez funkcję `rnt_robot::ecp_wait_for_start`.

**Zobacz:** `rnt_robot::ecp_wait_for_stop`

**Przykład:** Patrz przykład dla funkcji `teach`.

**Definicja:** `void rnt_robot::ecp_wait_for_stop(void)`

`#include "ecp.h"`

**Opis:** Metoda `rnt_robot::ecp_wait_for_stop` realizuje oczekiwanie w procesie ECP na polecenie STOP od procesu MP. Proces ECP jest zawieszony do chwili przyjścia polecenia. Jeśli otrzyma polecenie STOP odpowiada wówczas ECP_ACKNOWLEDGE w przeciwnym przypadku odpowiada INCORRECT_MP_COMMAND.

**Rezultat:** Metoda `rnt_robot::ecp_wait_for_stop` nie zwraca żadnej wartości.

**Błędy:** Funkcja `rnt_robot::ecp_wait_for_stop` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd niefatalny:

`INVALID_MP_COMMAND` – otrzymano niedozwolone w tym miejscu polecenie (inne niż STOP)

Ponadto jest wykrywany błąd systemowy:

`SYSTEM_ERROR` – błąd w komunikacji między procesem ECP i UI

Wyjątki mogą być również zgłaszane przez funkcje lub metody wywoływane przez funkcję `rnt_robot::ecp_wait_for_stop`.

**Zobacz:** `rnt_robot::ecp_wait_for_start`

**Przykład:** Patrz przykład dla funkcji `teach`.

**Definicja:** `void rnt_robot::execute_motion (void)`

`#include "ecp.h"`

**Opis:** Metoda `rnt_robot::execute_motion` zleca wykonanie ruchu przez robota. Jest to polecenie dla EDP. Przesyłane jest metodą `send`. Ponieważ EDP stanowi serwer poleceń, natychmiast wysyłane jest polecenie `QUERY`. Robi to metoda `query`.

**Rezultat:** Metoda `rnt_robot::execute_motion` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `rnt_robot::execute_motion` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `EDP_ERROR`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `rnt_robot::execute_motion`.

**Zobacz:** `Move`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `void rnt_robot::get_mp_command (void)`

`#include "ecp.h"`

**Opis:** Metoda `rnt_robot::get_mp_command` oczekuje na polecenie od procesu MP. Gdy polecenie nadchodzi, to wczytuje go do bufora poleceń oraz przekazuje procesowi MP informację zwrotną znajdującą się w buforze odpowiedzi. Operacje te wykonywane są za pomocą rozkazów `Receive` i `Reply` systemu QNX.

**Rezultat:** Metoda `rnt_robot::get_mp_command` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `rnt_robot::get_mp_command` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są błędy systemowe `SYSTEM_ERROR`.

**Definicja:** `void rnt_robot::get_reply (void)`

`#include "ecp.h"`

**Opis:** Metoda `rnt_robot::get_reply` pobiera z pakietu przesłanego z EDP informacje i wstawia je do odpowiednich składowych obrazu robota. W tym celu analizuje przesłane dane.

**Rezultat:** Metoda `rnt_robot::get_reply` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Metoda `rnt_robot::get_reply` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne:
`INVALID_POSE_SPECIFICATION`,
`INVALID_RMODEL_TYPE`,
`INVALID_EDP_REPLY`.

**Definicja:** `void save_file(teach_in_generator& the_generator, POSE_SPECIFICATION ps)`

```
#include "ecp.h"
```

**Opis:** Funkcja realizuje zapis uprzednio nauczonej trajektorii do pliku tekstowego (ASCII). Obiekt `the_generator` zawiera listę nauczonych pozycji we współrzędnych określonych przez argument `ps`. Wywołanie funkcji powoduje nawiązanie komunikacji między procesem ECP i procesem UI. Wysłanie polecenia `SAVE_FILE` powoduje wyświetlenie okienka umożliwiającego użytkownikowi podanie nazwy nowego pliku (opcja `New`) lub wybranie już istniejącego pliku (opcja `Save`), do którego będzie zapisana trajektoria. Pliki zapisywane są do bieżącego katalogu (domyślnie jest to `../mrrocpp/bin`). Format zapisu danych do pliku jest następujący:

| | |
|---|---|
| Pierwszy wiersz: | układ współrzędnych MOTOR, JOINT, XYZ_EULER_ZYZ, itd. |
| Drugi wiersz: | liczba zapamiętanych pozycji |
| Trzeci wiersz i następne: | czas ruchu i kolejne współrzędne (np. XYZ_EULER_ZYZ są to: $x, y, z, \varphi, \theta, \psi$) |

Można zrezygnować z zapisywania pliku wybierając opcję `Cancel`

**Rezultat:** Funkcja `save_file` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Funkcja `save_file` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne:

| | | |
|---|---|---|
| `NON_EXISTENT_DIRECTORY` | – | nie istnieje katalog o podanej nazwie, do którego ma być zapisany plik |
| `NON_EXISTENT_FILE` | – | nie został utworzony, bądź nie ma pliku o podanej nazwie, |

Ponadto wykrywany jest błąd systemowy:

| | | |
|---|---|---|
| `SYSTEM_ERROR` | – | błąd w komunikacji między procesem ECP i UI |

Wyjątki mogą być również zgłaszane przez funkcje lub metody wywoływane przez funkcję `save_file`.

**Zobacz:** `load_file`

**Przykład:** Patrz przykład dla funkcji `teach`.

**Definicja:**

```
BOOLEAN sensor_generator::first_step (
                          list<sensor>* sensor_list,
                          robot& the_robot)

#include "ecp.h"
```

**Opis:** Metoda `sensor_generator::first_step` określa czynności związane z generacją pierwszego kroku ruchu. Generator ten ma za zadanie spowodowanie ruchu końcówki manipulatora w zależnosci od kombinacji wciśniętych przycisków bistabilnego przełącznika klawiszowego. Ruch wykonywany jest wkierunkach osi X, Y, Z globalnego układu odniesienia. Metoda `sensor_generator::first_step` przygotowuje rozkaz odczytu aktualnej pozycji końcówki. W ten sposób zostaną zaktualizowane dane o stanie robota zarówno w EDP jak i w obrazie robota w ECP. Czujnik klawiszowy reprezentowany jest przez obiekt, który znajduje się na jednoelementowej liście czujników.

**Rezultat:** Metoda `sensor_generator::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

| | | |
|---|---|---|
| `true` | – | warunek końcowy nie jest spełniony (kontynuować ruch) |
| `false` | – | warunek końcowy jest spełniony (przerwać ruch) |

**Błędy:** Metoda `sensor_generator::first_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_MP_COMMAND`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `sensor_generator::first_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `sensor_generator::next_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:**

```
BOOLEAN sensor_generator::next_step (
                    list<sensor>* sensor_list,
                    robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `sensor_generator::next_step` określa czynności związane z generacją kolejnych kroków ruchu. Generator ten ma za zadanie spowodowanie ruchu końcówki manipulatora w zależnosci od kombinacji wciśniętych przycisków bistabilnego przełącznika klawiszowego. Ruch wykonywany jest wkierunkach osi X, Y, Z globalnego układu odniesienia. Czujnik klawiszowy reprezentowany jest przez obiekt, który znajduje się na jednoelementowej liście czujników. Jednoczesne wciśnięcie przycisków ruchu w kierunku plus i minus pewnej osi układu odniesienia kończy ruch. Ruch odbywa się względem początkowego (aktualnego) polożenia, ale w kierunku osi globalnego układu odniesienia. Orientacja końcówki nie jest zmieniana.

**Rezultat:** Metoda `sensor_generator::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

`true` – warunek końcowy nie jest spełniony (kontynuować ruch)
`false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `sensor_generator::next_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_MP_COMMAND`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `sensor_generator::next_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `sensor_generator::first_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `BOOLEAN show_message(char* message)`

```
#include "ecp.h"
```

**Opis:** Funkcja `show_message` służy do wysłania z ECP komunikatu `message` do operatora. Wymagane jest potwierdzenie przyjęcia komunikatu. Wywołanie funkcji powoduje nawiązanie komunikacji między procesem ECP i UI, a następnie wyświetlenie okienka z komunikatem i oczekiwanie na potwierdzenie. Funkcja jest wykorzystywana, jeśli w trakcie wykonywania procesu ECP niezbędne jest powiadomienie operatora np. o zajściu pewnego zdarzenia.

**Rezultat:** Metoda `show_message` zwraca wartość typu `BOOLEAN`:
`true` – potwierdzono przyjęcie komunikatu (`ANSWER_YES`)
`false` – brak potwierdzenia przyjęcia komunikatu

**Błędy:** Funkcja `show_message` reaguje na wystąpienie błędu zgłoszeniem wyjątku. Wykrywany jest błąd systemowy:
`SYSTEM_ERROR` – błąd w komunikacji między procesem ECP i UI

**Zobacz:** `input_integer, input_double, operator_reaction`

**Definicja:**

```
void teach(teach_in_generator& the_generator,
           POSE_SPECIFICATION ps,
           const char message[MSG_LENGTH])
```

```
#include "ecp.h"
```

**Opis:** Funkcja `teach` umożliwia uczenie robota. Wywołanie funkcji powoduje nawiązanie komunikacji z procesem UI i wyświetlenie okienka obsługi ruchów ręcznych z możliwością zapamiętania nauczonych pozycji. Każda nauczona pozycja jest zapamiętywana na liście tworzonej przez `the_generator`. Generator klasy `teach_in_generator` służy wyłącznie do zapamiętywania nauczonych pozycji na liście. Rodzaj współrzędnych, w których uczy się robota jest określony przez `ps`. Uczenie może odbywać się we współrzędnych wałów silników, wewnętrznych i zewnętrznych. Tablica `message` zawiera tekst komunikatu wyświetlanego w okienku obsługi uczenia.

**Rezultat:** Funkcja `teach` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Funkcja `teach` zgłasza wyjątek, jeśli wystąpi błąd systemowy `SYSTEM_ERROR`. Błąd ten jest sygnalizowany, jeśli nie powiedzie się nawiązanie komunikacji z UI. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez funkcję `teach`.

**Zobacz:** `load_file`, `save_file`

**Przykład:** Przykład funkcji main procesu ECP realizującego sterownik robota dedykowany konkretnemu zadaniu.

```
// -------------------------------------------------------------------
//                              ecp_m.cc
//
//                       EFFECTOR CONTROL PROCESS (ECP) - main()
//
// Ostatnia modyfikacja: 10.04.99
// -------------------------------------------------------------------
//  Proces ECP do frezowania - wersja z uczeniem

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <i86.h>
#include <signal.h>
#include <process.h>
#include <sys/name.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/kernel.h>
#include <sys/sched.h>
#include <sys/qnx_glob.h>
#include <sys/proxy.h>
#include <string.h>
#include <fstream.h>
```

```
#include "typedefs.h"
#include "impconst.h"
#include "com_buf.h"
#include "lst.h"
#include "canal.h"
#include "srplib.h"
#include "ecp.h"

srp_ecp_rnt* msg;              // Wskaźnik na obiekt do komunikacji
                               // z SRP
pid_t UI_pid;                  // Identyfikator procesu UI

// Przechwycenie sygnału
void catch_signal(int sig) {
  switch(sig) {
case SIGTERM :
  _exit(EXIT_SUCCESS);
  break;
  } // end: switch
 }

void main(int argc, char *argv[]) {

  pid_t MP_pid;            // Identyfikator procesu MP
  pid_t EDP_MASTER_Pid;    // Identyfikator procesu EDP
  int ecp_id;              // Identyfikator nazwy procesu ECP
  nid_t UI_node;           // Numer węzła na którym działa proces UI
  pid_t caller;            // Identyfikator klienta
  unsigned long int e;     // Kod błędu systemowego
  char ch;                 // Znak decyzyjny
  int i, j;                // Licznik
  int pll;                 // licznik

  // Deklaracje obiektów do współpracy z driver'ami robotów.
  // Zarówno obiektów robotów jak i czujników konkretnych nie wolno
  // tworzyć inaczej gdyż destruktor listy niszczy jedynie listę,
  // ale nie jej elementy konkretne

  // Robot RNT:
  rnt_robot rnt(EDP_NAME); // Deklaracja obiektu klasy rnt_robot
  // Generatory odtwarzajace nauczone trajektorie frezowania:
  teach_in_generator_wt tig;          // generator służący do uczenia
                                      // robota
  parabolic_teach_in_generator adg1;  // generator dla trajektorii
                                      // dojscia
  parabolic_teach_in_generator adg2;  // generator dla trajektorii
                                      // dojscia
  parabolic_teach_in_generator rys;   // generator dla trajektorii
                                      // frezowania

  // Lista czujnikow
  list<sensor>* slp = NULL; //  Wskazuje początek listy

 try {
   setprio(getpid(), MAX_PRIORITY-3);
   signal(SIGTERM, &catch_signal);
   // Utworzenie obiektu do komunikacji z SRP
   if ((msg = new srp_ecp_rnt(ECP, ECP_NAME, SRP_NAME)) == NULL) {
```

```
      e = errno;
      perror ( "Unable to locate SRP\n");
    }
    // Rejestracja procesu ECP
    if ( (ecp_id = qnx_name_attach(0L, ECP_NAME)) == -1) {
      e = errno;
      perror("Failed to attach Effector Control Process\n");
      msg->message (SYSTEM_ERROR, e,
                   "Failed to attach Effector Control Process");
      throw ECP_main_error(SYSTEM_ERROR, (unsigned word32) 0);
    }

    // Podstawienie numeru węzła, na którym działa UI
    UI_node = (nid_t) atoi(argv[1]);

    // Utworzenie połączenia z UI
    if (( UI_pid = Connect(UI_NAME, UI_node)) < 0) {
      e = errno;
      perror("Connect to UI failed\n");
      msg->message (SYSTEM_ERROR, e, "Connect to UI failed");
      throw ECP_main_error(SYSTEM_ERROR, (unsigned word32) 0);
    }

    // Lokalizacja procesu EDP_MASTER - określenie jego identyfikatora
    if((EDP_MASTER_Pid = qnx_name_locate(0L, EDP_NAME, NULL, 0)) < 0 )
    {
       e = errno;
       perror("Unable to locate EDP_MASTER process\n");
       msg->message (SYSTEM_ERROR, e,
                    "Unable to locate EDP_MASTER process");
       throw ECP_main_error(SYSTEM_ERROR, (unsigned word32) 0);
    };

    // Lokalizacja procesu MP - określenie jego identyfikatora (pid)
    if((MP_pid = qnx_name_locate(0L, MP_NAME, NULL, 0)) < 0 ) {
      e = errno;
      perror("ECP: Unable to locate MP_MASTER process\n");
      msg->message (SYSTEM_ERROR, e,
                   "ECP: Unable to locate EDP_MASTER process");
      throw ECP_main_error(SYSTEM_ERROR, (unsigned word32) 0);
    };
    delay(2000);

    // Przepisanie identyfikatora procesu EDP_MASTER do obiektu klasy
    // rnt_robot
    rnt.set_edp_master_pid(EDP_MASTER_Pid);

    // Przepisanie identyfikatora procesu MP do obiektu klasy rnt_robot
    rnt.set_mp_pid(MP_pid);

    // Komunikat do SRP
    msg->message("ECP loaded");
  }
  catch (ECP_main_error e) {
    // Obsluga bledow ECP
   if (e.error_class == SYSTEM_ERROR)
     exit(EXIT_FAILURE);
  } //end: catch
 try {
```

```
for (;;) { //Zewnętrzna pętla nieskończona
 try {
   // Oczekiwanie na polecenie START od MP
   if ( rnt.ecp_wait_for_start() )
     continue;

   // Żądanie reakcji operatora -- czy rozpocząć uczenie
   // trajektorii dojścia?
   if( operator_reaction ("Teach in approach trajectory?") ) {
     adg1.flush_pose_list(); // Usunięcie listy pozycji,
                             // o ile istnieje

     // Uczenie trajektorii dojścia
     teach (adg1, MOTOR, "Teach-in approach trajectory\n");

     // Żądanie reakcji operatora -- czy zapisać trajektorię dojścia?
     if ( operator_reaction ("Save approach trajectory? ") )
       // Zapis trajektorii dojścia do pliku tekstowego
       save_file (adg1, MOTOR);
  }
  else
    // Żądanie reakcji operatora -- czy wczytać trajektorię dojścia?
    if ( operator_reaction ("Load approach trajectory? ") )
      // Wczytanie trajektorii dojścia
      load_file (adg1);

  // Żądanie reakcji operatora -- Czy rozpocząć uczenie trajektorii
  // frezowania?
  if ( operator_reaction ("Teach in milling trajectory?") ) {
    rys.flush_pose_list(); // Usunięcie listy pozycji, o ile istnieje

    // Uczenie trajektorii frezowania
    teach (rys, XYZ_EULER_ZYZ, "Teach-in milling trajectory\n");

    // Żądanie reakcji operatora -- czy zapisać trajektorię
    // frezowania?
    if ( operator_reaction ("Save milling trajectory? ") )
      save_file (rys, XYZ_EULER_ZYZ);
  }
  else
    // Żądanie reakcji operatora -- czy wczytać trajektorię
    // frezowania?
    if ( operator_reaction ("Load milling trajectory? ") )
      // Wczytanie trajektorii frezowania
      load_file (rys);

  // Żądanie reakcji operatora -- czy rozpocząć uczenie trajektorii
  // odejścia?
  if ( operator_reaction ("Teach in departure trajectory?") ) {
adg2.flush_pose_list(); // Usunięcie listy pozycji,
                        // o ile istnieje

    // Uczenie trajektorii odejścia
    teach (adg2, MOTOR, "Teach-in departure trajectory\n");
    if ( operator_reaction ("Save departure trajectory? ") )
      // Zapis trajektorii odejścia do pliku tekstowego
      save_file (adg2, MOTOR);
  }
  else
```

```
      // Żądanie reakcji operatora -- czy wczytać trajektorię odejścia?
      if ( operator_reaction ("Load departure trajectory? ") )
        // Wczytanie trajektorii odejścia
        load_file (adg2);

    // Żądanie reakcji operatora -- czy rozpocząć odtwarzanie
    // trajektorii?
    if ( operator_reaction ("Start motion? ") ) {
      // Odtwarzanie nauczonej lub wczytanej trajektorii zaczynamy
      // od początku

      // Ustawienie wskaźnika na początek listy z trajektorią dojścia
      adg1.initiate_pose_list();

      // Dlugosc listy = liczba pozycji do odtworzenia
      pll = adg1.pose_list_length();

      for (i=0; i< pll; i++) {  // Wewnętrzna pętla wykonuje się
                                // pll razy (tyle pozycji nauczono)
        Move (rnt, NULL, adg1); // Ruch do następnej nauczonej pozycji
      } // end: for

      // Żądanie reakcji operatora -- czy rozpocząć frezowanie?
      while ( !operator_reaction ("Start milling? ") );

      // Ustawienie wskaźnika na początek listy z trajektorią
      // frezowania
      rys.initiate_pose_list();

      // Dlugość listy = liczba pozycji do odtworzenia
      pll = rys.pose_list_length();

      // Frezowanie
      for (i=0; i< pll; i++) {  // Wewnętrzna pętla wykonuje się
                                // pll razy (tyle pozycji nauczono)
        Move (rnt, NULL, rys);  // Ruch do następnej nauczonej pozycji
      } // end: for

      // Żądanie reakcji operatora -- czy zakończyć frezowanie?
      while ( !operator_reaction ("End milling? ") );


      // Odtwarzanie nauczonej lub wczytanej trajektorii zaczynamy
      // od początku
      // Trajektoria odejscia
      adg2.initiate_pose_list();
      // Dlugosc listy = liczba pozycji do odtworzenia
      pll = adg2.pose_list_length();
      for (i=0; i< pll; i++) {    // Wewnętrzna pętla wykonuje się
                                  // pll razy (tyle pozycji nauczono)
Move (rnt, NULL, adg2);  // przejście do następnej nauczonej
                         // pozycji
      } // end: for
    }

    // Informacja dla procesu MP o zakończeniu zadania użytkownika
    rnt.ecp_termination_notice ();

    // Oczekiwanie na polecenie STOP od operatora
```

```
     rnt.ecp_wait_for_stop ();

    } //: end try zewnetrzne
    // Obsługa wyjątków
    catch (ECP_main_error e) {
    // Obsługa błędów ECP
  if (e.error_class == SYSTEM_ERROR)
    exit(EXIT_FAILURE);
    } //end: catch

    catch (robot::ECP_error er) {
     // Wyłapywanie błędów generowanych przez moduł transmisji danych
     // do EDP
     // Błąd systemowy już wysłano komunikat do SRP
     if ( er.error_class == SYSTEM_ERROR) {
        perror("ECP aborted due to SYSTEM_ERROR\n");
        exit(EXIT_FAILURE);
      }

      switch ( er.error_no ) {
        case INVALID_POSE_SPECIFICATION:
        case INVALID_ECP_COMMAND:
        case INVALID_COMMAND_TO_EDP:
        case EDP_ERROR:
        case INVALID_EDP_REPLY:
        case INVALID_RMODEL_TYPE:
//Komunikat o błędzie wysyłamy do SRP
msg->message (NON_FATAL_ERROR, er.error_no);
          rnt.set_ecp_reply (ERROR_IN_ECP);
          rnt.get_mp_command();
break;
        default:
         msg->message (NON_FATAL_ERROR, 0,
                "ECP: Unidentified exception");
         perror("Unidentified exception");
        exit(EXIT_FAILURE);
       } // end: switch
    } //end: catch

    catch (generator::ECP_error er) {
    // Wyłapywanie błędów generowanych przez generatory
      if ( er.error_class == SYSTEM_ERROR) {
        perror("ECP aborted due to SYSTEM_ERROR");
        exit(EXIT_FAILURE);
      }
      switch ( er.error_no ) {
        case INVALID_POSE_SPECIFICATION:
        case INVALID_MP_COMMAND:
        case NON_EXISTENT_DIRECTORY:
        case NON_TRAJECTORY_FILE:
        case NON_EXISTENT_FILE:
        case READ_FILE_ERROR:
        case NON_COMPATIBLE_LISTS:
        case MAX_ACCELERATION_EXCEEDED:
        case MAX_VELOCITY_EXCEEDED:
          //Komunikat o błędzie wysyłamy do SRP
        msg->message (NON_FATAL_ERROR, er.error_no);
          rnt.set_ecp_reply (ERROR_IN_ECP);
          rnt.get_mp_command();
```

```
        break;
      case ECP_STOP_ACCEPTED:
        break;
      default:
       msg->message (NON_FATAL_ERROR, 0,
              "ECP: Unidentified exception");
       perror("Unidentified exception");
       exit(EXIT_FAILURE);
    } // end: switch
  } //end: catch

  catch (condition::ECP_error er) {
  // Wyłapywanie błędów generowanych przez warunek początkowy
    if ( er.error_class == SYSTEM_ERROR) {
      perror("ECP aborted due to SYSTEM_ERROR");
      exit(EXIT_FAILURE);
    }
    switch ( er.error_no ) {
      case 0: // Zastąpić etykietą, gdy condition będzie zgłaszać
              // wyjątek
        //Komunikat o błędzie wysyłamy do SRP
        msg->message (NON_FATAL_ERROR, er.error_no);
        rnt.set_ecp_reply (ERROR_IN_ECP);
        rnt.get_mp_command();
        break;
      case ECP_STOP_ACCEPTED:
        break;
      default:
        msg->message (NON_FATAL_ERROR, 0,
              "ECP: Unidentified exception");
        perror("Unidentified exception");
        exit(EXIT_FAILURE);
    } // end: switch
  } //end: catch

  catch (...) {  // Dla zewnętrznej pętli try
    // Wyłapywanie niezdefiniowanych błędów
    // Komunikat o błędzie wysyłamy do SRP
    msg->message (NON_FATAL_ERROR,
                  (unsigned word32) ECP_UNIDENTIFIED_ERROR);
    exit(EXIT_FAILURE);
  } //end: catch

  msg->message("ECP user program is finished");
 } // end: for (;;) zewnetrznej
} // end: try zewn.
catch (...) {  // Dla zewnetrznej pętli try
  // Wyłapywanie błędów zgłaszanych w "catch'ach"
  //Komunikat o błędzie wysyłamy do SRP
  msg->message (SYSTEM_ERROR, (unsigned word32) 0,
               "ECP GLOBAL ERROR");
  exit(EXIT_FAILURE);
} //end: catch

}; // koniec: main()
// -------------------------------------------------------------------
```

**Definicja:**

```
BOOLEAN teach_in_generator::first_step (
                                list<sensor>* sensor_list,
                                robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `teach_in_generator::first_step` określa czynności związane z generacją pierwszego kroku ruchu. W tym przypadku jest to ustawienie wskaźnika na pierwszy element listy pozycji nauczonych. Następnie wywołuje `teach_in_generator::next_step`.

**Rezultat:** Metoda `teach_in_generator::first_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

| | | |
|---|---|---|
| `true` | – | warunek końcowy nie jest spełniony (kontynuować ruch) |
| `false` | – | warunek końcowy jest spełniony (przerwać ruch) |

**Błędy:** Metoda `teach_in_generator::first_step` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `teach_in_generator::first_step`.

**Zobacz:** `teach_in_generator.next_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:**

```
BOOLEAN teach_in_generator::next_step (
                                list<sensor>* sensor_list,
                                robot& the_robot)
```

```
#include "ecp.h"
```

**Opis:** Metoda `teach_in_generator::next_step` określa czynności związane z generacją kolejnych kroków ruchu. Wpierw metoda kontaktuje się z MPprzekazując `ECP_ACKNOWLEDGE`. Następnie sprawdza, czy wskaźnik bieżący listy pozycji nauczonych równa się `NULL`. Jezeli tak, to kończona jest generacja trajektorii. W przeciwnym przypadku, jeżeli MP przyłało polecenie to `NEXT_POSE`, to formowany jest rozkaz dla EDP. Jest to `SET ARM ABSOLUTE`. W zależności od tego w jaki sposób zostało wyrażone położenie ramienia na liście pozycji nauczonych wybierany jest jeden z parametrów `MOTOR`, `JOINT`, `XYZ_EULER_ZYZ`, `XYZ_ANGLE_AXIS`. Ponadto określane są parametry ruchu (liczba kroków oraz kiedy ma być przysłana informacja o realizacji ruchu). Polecenie składowane jest w obrazie robota. Następnie metoda `the_robot.create_command` przepisze polecenie z obrazu do bufora przesyłkowego do procesu EDP. Ostatnią czynnością jest przesunięcie wskaźnika elementów listy pozycji nauczonych na kolejną pozycje, jeżeli istnieje lub wstawienie `NULL`. Polecenie operatora `STOP` oraz sytuacje awaryjne sygnalizowane są przez zgłoszenie odpowiednich wyjątków.

**Rezultat:** Metoda `teach_in_generator::next_step` zwraca wartość `BOOLEAN`. Jest to negacja warunku końcowego, tzn.:

`true` – warunek końcowy nie jest spełniony (kontynuować ruch)
`false` – warunek końcowy jest spełniony (przerwać ruch)

**Błędy:** Metoda `teach_in_generator::next_step` reaguje na wykrycie błędu zgłoszeniem wyjątku. Wykrywane są następujące błędy niefatalne: `INVALID_MP_COMMAND`, `INVALID_POSE_SPECIFICATION`. Ponadto wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez metodę `teach_in_generator::next_step`. Przerwanie ruchu poleceniem operatora `STOP` także sygnalizowane jest wyjątkiem – `ECP_STOP_ACCEPTED`.

**Zobacz:** `teach_in_generator.first_step`

**Przykład:** Patrz tekst `Move` dla procesu ECP.

**Definicja:** `teach_in_generator::teach_in_generator (void)`

`#include "ecp.h"`

**Opis:** Metoda `teach_in_generator::teach_in_generator` tworzy i inicjuje obiekt klasy `teach_in_generator`.

**Rezultat:** Metoda `teach_in_generator::teach_in_generator` nie zwraca żadnej wartości (jest konstruktorem obiektu).

**Błędy:** Metoda `teach_in_generator::teach_in_generator` nie zgłasza wyjątków.

**Zobacz:** `teach_in_generator.first_step`, `teach_in_generator.next_step`

**Definicja:**

```
void Wait (robot& the_robot,
           list<sensor>* sensor_list,
           condition& the_condition)
```

```
#include "ecp.h"
```

**Opis:** Funkcja `Wait` powoduje oczekiwanie na spełnienie warunku początkowego `the_condition`. Korzysta z informacji pochodzacych w czujników wirtualnych umieszczonych na liście `sensor_list` oraz aktualnego stanu efektora `the_robot`. Warunek początkowy określony jest specyfikacją zawartą w metodzie `the_condition.condition_value`. Na wstępie żądane są dane od wszystkich czujników umieszczonych na liście `sensor_list`. W kolejnej fazie wykonania funkcji `Wait` odczytywane są dane uzyskane przez wszystke czujniki umieszczone na liście `sensor_list`. Na koniec metoda `the_generator.condition_value` sprawdza warunek końcowy oraz jeżeli nie jest on spełniony ponawia powyższe operacje. W przeciwnym przypadku `Wait` kończy swe działanie. Sieć działań funkcji `Wait` przedstawiono na rysunku 4.2.

**Rezultat:** Funkcja `Wait` nie zwraca żadnej wartości (jest procedurą).

**Błędy:** Funkcja `Wait` nie zgłasza wyjątków. Wyjątki mogą być zgłaszane przez funkcje lub metody wywoływane przez funkcję `Wait`.

**Tekst:**

```
void Wait(robot& the_robot, list<sensor>* sensor_list,
          condition& the_condition) {

// Instrukcja oczekiwania (dla ECP)

 list<sensor>* sensor_lptr = NULL;  // wskazuje aktualnie przetwarzany
                                    // element listy czujników

 do { // kontakt z czujnikami oraz sprawdzenie warunku poczatkowego

   // żądanie danych od wszystkich czujników
   for (sensor_lptr = sensor_list; sensor_lptr;
        sensor_lptr = sensor_lptr->next)
     sensor_lptr->E_ptr->initiate_reading();

   // odczytanie danych z wszystkich czujników
   for (sensor_lptr = sensor_list; sensor_lptr;
        sensor_lptr = sensor_lptr->next)
     sensor_lptr->E_ptr->get_reading();

   // sprawdzenie warunku poczatkowego
   // - jego spelnienie konczy oczekiwanie
 } while ( !the_condition.condition_value(sensor_list, the_robot) );

}; // end: Wait()
```

# Rozdział 6

# Funkcje i metody procesu VSP

W tym rozdziale zgromadzono wszystkie funkcje i metody, które mogą być przydatne dla użytkownika (programisty systemu MRROC++) przy tworzeniu procesów VSP.

Funkcje i metody zostały wymienione w porządku alfabetycznym. W przypadku metod wpierw uwzględniono nazwę klasy, z której pochodzą.

**Definicja:** `BOOLEAN vsp_bin_sensor::get_ecp_demand ( void )`

`#include "vsp.cc"`

**Opis:** Funkcja `get_ecp_demand` przetwarza aktywnie bufor `to_vsp` otrzymany z ECP. Jeżeli otrzymany w polu `to_vsp.instruction_code` kod instrukcji odpowiada `VSP_INIT` albo `VSP_GET_READING` oraz definicja programisty (`MY_DEFINITION`) oczekiwanego typu danych jest prawidlowa (`FLOAT_READING` albo `INTEGER_READING`) dokonywany jest odczyt poprzez wywołanie funkcji `GetSensorReading`. Jeżeli z ECP debrano żądanie zakończenia (`VSP_TERMINATE`) to funkcja kończy działania odpowiadając ECP (`Reply..`) oraz zwracając wartość `FALSE`. Jeżeli odebrana od ECP instrukcja jest inna (nie przewidziana w liście instrukcji), to w polu `from_vsp.vsp_report` przesyłana jest informacja `INVALID_ECP2VSP_COMMAND`.

Zarówno przy odczycie, jak i w tej błędnej sytuacji funkcja zwraca wartość `TRUE`.

W programie głównym następuje powołanie do życia obiektu klasy `vsp_bin_sensor`, na przykład `vsp_bi`. Następnie wywoływana jest funkcja `get_ecp_demand` (instrukcja: `while(vsp_bi.get_ecp_demand())`) dopóty dopóki zwraca ona wartość `TRUE`, natomiast przekazanie przez nią wartości `FALSE` powoduje zakończenie programu głównego.

**Rezultat:** Funkcja `vsp_bin_sensor::get_ecp_demand` zwraca wartość `TRUE` po każdym odczycie wykonamym przy poprawnym żadaniu otrzymanym z ECP oraz zwraca wartość `FALSE`, gdy z odpowiedniego ECP otrzymała ona kod instrukcji: `VSP_TERMINATE`

**Błędy:** Funkcja `vsp_bin_sensor::get_ecp_demand` nie sygnalizuje żadnych błędów.

**Zobacz:** `vsp_bin_sensor::GetSensorReading`

**Przykład:**

```
int main( int argc, char *argv[] )
{
   int e;

   if ((msg = new srp_vsp_rnt(VSP, VSP_NAME, SRP_NAME)) == NULL) {
     e = errno;
     perror ( "Unable to locate SRP\n");
   }

  // czujnik zwiazany z tym VSP
  vsp_bin_sensor vsp_bi;

  while(vsp_bi.get_ecp_demand());
  return 0;
}
```

**Definicja:** `void vsp_bin_sensor::GetSensorReading ( void )`

```
#include "vsp.cc"
```

**Opis:** Funkcja `GetSensorReading` klasy `vsp_bin_sensor` wykorzystywana jest w `vsp_bin_sensor::get_ecp_demand`. Dokonuje ona odczytu stanu czujników dwustanowych (klawiszowych) i podstawia te odczyty do bufora komunikacyjnego `from_vsp` w postaci jednej danej ośmiobitowej (stan ośmiu czujników) albo danych przetworzonych gdzie jednemu czujnikowi odpowiada jedna dana typu `integer`, która może mieć wartość `1` albo `0`. Jeżeli programista zdefiniował stałą `MY_DEFINITION` (`vsp.h`) jako `FLOAT_READING` to realizowana jest pierwsza opcja zapisu danych z czujnikow, jezeli ta stała jest zdefiniowana jako `INTEGER_READING` to realizowana jest druga opcja.

**Rezultat:** Funkcja `vsp_bin_sensor::GetSensorReading` nie zwraca żadnej wartości.

**Błędy:** Funkcja `vsp_bin_sensor::GetSensorReading` nie sygnalizuje żadnych błędów.

**Zobacz:** `vsp_bin_sensor::get_ecp_demand`

**Definicja:** `vsp_bin_sensor::vsp_bin_sensor ( void )`

```
#include "vsp.cc"
```

**Opis:** Konstruktor `vsp_bin_sensor::vsp_bin_sensor` klasy `vsp_bin_sensor` rejestruje w węźle wykonującym proces nazwę (podaną jako stała `VSP_NAME`).

**Rezultat:** Konstruktor `vsp_bin_sensor::vsp_bin_sensor` nie zwraca żadnej wartości.

**Błędy:** Funkcja `vsp_bin_sensor::vsp_bin_sensor` reaguje na wystąpienie błędu rejestracji procesu zgłoszeniem wyjątku.

Generowany jest błąd systemowy:
`SYSTEM_ERROR` – błąd w rejestracji VSP.

**Zobacz:** `vsp_bin_sensor::GetSensorReading`, `vsp_bin_sensor::get_ecp_demand`

# Dodatek A

# Słownik terminów

**Objaśnienie:** terminy zapisane wytłuszczoną czcionką w treści definicji danego hasła są zdefiniowane w innym miejscu niniejszym słownika. Aby uzupełnić objaśnienie należy się odwołać do tego terminu.

## A

**Agregacja danych** — proces obliczeniowy mający na celu wyodrębnienie, istotnych z punktu widzenia sterowania ruchem, informacji uzyskanych z **czujników rzeczywistych**. W wyniku agregacji powstaje odczyt **czujnika wirtualnego**.

## B

**Bufor** — miejsce składowania w procesie **pakietu komunikacyjnego**.

## C

**Czujnik rzeczywisty** — urządzenie techniczne przeznaczone do zbierania informacji ze środowiska. Jego odczyt nie nadaje się do bezpośredniego wykorzystania do generacji trajektorii ruchu robotów. Musi podlegać **agregacji**.

**Czujnik wirtualny** — zagregowany odczyt kilku prostych lub jednego złożonego **czujnika rzeczywistego**. Odczyt czujnika wirtualnego nadaje się do bezpośredniego wykorzystania do generacji trajektorii ruchu robotów.

## E

**ECP** — *ang.* Effector Control Process — proces realizujący **program użytkowy** dla pojedynczego robota. Proces ten nie zależy od typu robota, ale zależy od zadania, które ma być zrealizowane.

**EDP** — *ang.* Effector Driver Process — proces realizujący bezpośrednie sterowanie pojedynczym robotem. Proces ten zależy od typu robota, ale nie zależy od zadania, które ma być zrealizowane.

**Efektor** — urządzenie techniczne przeznaczone do przemieszczania narzędzia lub innych obiektów. Może zmieniać stan środowiska. Zazwyczaj jest to **robot**, ale może to być dowolne urządzenie współpracujące, np. taśmociąg lub podajnik.

## G

**Generator trajektorii** — obiekt (w sensie `C++`) odpowiedzialny za wyliczenie następnej wartości zadanej dla efektorów na podstawie aktualnej wartości zmiennych programowych, odczytów **czujników wirtualnych** oraz stanu efektorów. Jest on równocześnie odpowiedzialny za określenie wartośći **warunku końcowego**. Generatory trajektorii są obiektami powoływanymi do życia w procesach MP i ECP.

## I

**Instrukcja Move** — instrukcja zmieniająca aktywnie stan robotów lub ich środowiska.

**Instrukcja ruchowa** — instrukcja zmieniająca aktywnie lub biernie stan robota lub środowiska. Zmiana aktywna polega na przemieszczeniu efektora. **Efektor** z kolei może zmienić stan środowiska. Zmiana bierna polega na antycypacji tejże zmiany. Sama zmiana jest spowodowana działaniem jakiegoś urządzenia lub innego agenta (np. człowieka), który nie wchodzi w skład systemu.

**Instrukcja Wait** — instrukcja zmieniająca biernie stan środowiska. Działanie jej sprowadza sie do oczekiwania na spełnienie **warunku wstępnego**. Spełnienie tego warunky jest równoważne zajściu określonej zmiany w stanie środowiska.

## J

**Jądro procesu** — wymienna część procesu dostarczana w postaci napisanego w `C++` kodu programu użytkowego, który ma być zrealizowany przez system. Jądro umieszczane jest w **powłoce procesu**.

## K

**Klasa abstrakcyjna** — klasa (w sensie `C++`), z której wywodzi się klasy pochodne reprezentujące konkretne pojęcia lub urządzenia. Obiektów tej klasy nie można powołać do życia. Dopiero obiekty klas pochodnych będących jednocześnie obiektami klas konkretnych można użyć w **programie użytkowym**.

**Klasa czujnik** — bazowa klasa abstrakcyjna (w sensie `C++`), z której wywodzi się klasy pochodne reprezentujące czujniki określonych typów. Występuje w MP i ECP. Klasy pochodne reprezentują we wzmiankowanych procesach **czujniki wirtualne**.

**Klasa generator** — klasa (w sensie `C++`), z której wywodzi się klasy pochodne reprezentujące konkretne **generatory trajektorii**. Obiekty klas pochodnych powołuje się do życia w procesach MP i ECP.

**Klasa konkretna** — klasa (w sensie `C++`) wywiedziona z **klasy abstrakcyjnej**. Reprezentuje konkretne pojęcia lub urządzenia. Dopiero obiekty tej klasy można użyć w **programie użytkowym**.

**Klasa robot** — bazowa klasa abstrakcyjna (w sensie `C++`), z której wywodzi się klasy pochodne reprezentujące roboty określonych typów. Występuje w MP i ECP. Klasy pochodne reprezentują w wyżej wzmiankowanych procesach **roboty**.

**Klasa sensor** — **klasa czujnik**

**Klasa warunek** — bazowa klasa abstrakcyjna (w sensie `C++`), z której wywodzi się klasy pochodne reprezentujące konkretne **warunki wstępne**.

**Krok ruchu** — przyrost położenia ramienia robota realizowany w pojedynczym okresie próbkowania (obecnie 2 ms – okres pracy algorytmu regulacji).

## M

**Makrokrok** — Pojedyncze zlecenie ruchu dla procesu EDP lub SERVO_GROUP. Składa się z **kroków ruchu**.

**Move** — funkcja (w sensie `C++`) implementująca instrukcję zmieniającą aktywnie stan robotów lub ich środowiska.

**MP** — *ang.* Master Process — proces zależny od zadania, które ma wykonać system. Koordynuje pracę wszystkich robotów i urządzeń współpracujących (**efektorów**).

**MRROC** — *ang.* Multi-Robot Research Oriented Controller. Wersja proceduralna biblioteki służącej do konstruowania sterowników dedykowanych konkretnym zadaniom, które mają być zrealizowane przez system wielorobotowy wyposażony w różnorodne czujniki i urządzenia współpracujące.

**MRROC++** — wersja obiektowa **MRROC**.

## O

**Obiekt czujnik** — obiekt (w sensie `C++`) klasy wywiedzionej z **klasy czujnik** (`sensor`).

**Obiekt robot** — obiekt (w sensie `C++`) klasy wywiedzionej z **klasy robot**.

**Obiekt sensor** — obiekt (w sensie `C++`) klasy wywiedzionej z **klasy sensor**.

**Obraz czujnika** — struktura danych reprezentująca model czujnika, takim jakim **czujnik wirtualny** jest postrzegany przez programistę piszącego **program użytkowy**, a dokładniej jego część składowa jaką jest **generator trajektorii** używany przez MP lub ECP. Stanowi część składową klasy `sensor`.

**Obraz robota** — struktura danych reprezentująca model robota, takim jakim **robot** jest postrzegany przez programistę piszącego **program użytkowy**, a dokładniej jego część składowa jaką jest **generator trajektorii** używany przez MP lub ECP. Stanowi część składową klasy `robot`.

**Operator** — osoba wydająca polecenia systemowi w trakcie jego pracy. Może zlecić, zakończyć, wstrzymać lub wznowić wykonanie **programu użytkowego**. Ponadto odpowiedzialna jest za synchronizację robotów, konfigurację systemu, wykonywanie ruchów ręcznych *etc.*

## P

**Pakiet komunikacyjny** — struktura danych przesyłana między procesami.

**Pętla bierna** — fragment procesu SERVO_GROUP, który jest realizowany, gdy procesy nadrzędne nie przyślą nowych wartości zadanych dla regulatorów położeń wałów silników. Realizowany jest wtedy bierny **krok ruchu**.

**Pętla czynna** — fragment procesu SERVO_GROUP, który jest realizowany, gdy procesy nadrzędne przyślą nowe wartości zadane dla regulatorów położenia wałów silników. Realizowany jest wtedy czynny **krok ruchu**.

**Powłoka procesu** — niezmienna część procesu odpowiedzialna za komunikację z innymi procesami oraz powoływanie i likwidację łączy komunikacyjnych oraz innych procesów. Powłoka otacza **jądro procesu**.

**Pośrednik** — *ang.* `proxy`. Służy przekazaniu stałej wiadomości między procesami. Wykorzystywane również do synchronizacji procesów.

**Program użytkowy** — program napisany w `C++` z użyciem funkcji bibliotecznych systemu `MRROC++`, który realizuje zadanie zlecone systemowi do wykonania przez użytkownika. Program użytkowy stanowi **jądra procesów**: MP i ECP.

## R

**Robot** — urządzenie techniczne przeznaczone do przemieszczania narzędzia.

**Rozkaz ruchu** — to samo co **instrukcja ruchowa**.

## S

**Spotkanie** — *fr.* `rendez vous`. Służy wzajemnemu przekazaniu między procesami wiadomości o zmiennej treści. Wykorzystywane również do synchronizacji procesów.

**SRP** — *ang.* System Response Process — proces odpowiedzialny za formatowanie i wyświetlanie komunikatów o stanie poszczególnych procesów systemu oraz o ewentualnych błędach wykrytych w tych procesach lub przez nie.

**Synchronizacja robota** — sekwencyjne przemieszczanie poszczególnych stopni swobody ramienia robota aż do wykrycia wyłączników zwanych synchronizacyjnymi. Procedura ta musi być wykonana jednokrotnie, zaraz po uruchomieniu systemu, w celu wyzerowania liczników mierzących położenie wałów silników w sposób przyrostowy względem pozycji synchronizacji.

## U

**UI** — *ang.* User Interface — proces odpowiedzialny za nasłuch poleceń operatora. Zawiaduje klawiaturą.

## W

**Wait** — funkcja (w sensie `C++`) implementująca instrukcję zmieniającą biernie stan środowiska. Działanie jej sprowadza sie do oczekiwania na spełnienie **warunku wstępnego**. Spełnienie tego warunku jest równoważne zajściu określonej zmiany w stanie środowiska.

**Warunek końcowy** — warunek badany przez **instrukcję Move**. Spełnienie tego warunku kończy realizację ruchu (kończy realizację funkcji **Move**).

**Warunek początkowy** — to samo co **warunek wstępny**.

**Warunek wstępny** — warunek badany przez **instrukcję Wait**. Jego spełnienie kończy oczekiwanie. Warunek wstępny jest wyrażeniem logicznym zbudowanym z relacji pomiędzy wartościami zmiennych, odczytami **czujników wirtualnych** oraz aktualnym stanem **efektorów**.