System MRROC++ dla robota IRp-6

Cezary Zieliński, Wojciech Szynkiewicz

Instytut Automatyki i Informatyki Stosowanej
Politechnika Warszawska

Raport IAIS nr 99–20
Warszawa, Maj 1999

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

Spis treści

Rozdział 1

Wstęp

1.1. Programowanie robotów

Metody programowania robotów mogą być podzielone na trzy grupy:

- on-line,
- off-line,
- hybrydowe.

Programowanie on-line korzysta w bardzo dużym stopniu z robota do stworzenia programu, natomiast **programowanie off-line** nie wymaga użycia manipulatora do napisania programu. Pomiędzy tymi dwoma skrajnymi metodami można wyróżnić całą gamę sposobów, które łączą w sobie cechy obu powyższych metod. Metody te zalicza się do **programowania hybrydowego**, to znaczy że w niewielkim stopniu, ale jednak, wymagają one użycia robota do programowania.

1.1.1. Programowanie on-line

Programowanie on-line bazuje na uczeniu robota sposobu realizacji trajektorii. **Uczenie** polega na spowodowaniu wykonania sekwencji ruchów oraz na zapamiętaniu tych ruchów. Po nauczeniu robota wykonywania ruchów program odtwarzany jest automatycznie. Podczas uczenia ramię może być przemieszczane ręcznie, to znaczy że operator używa siły swych mięśni do przesuwania manipulatora, lub przy użyciu napędów robota. W drugim przypadku operator używa do uczenia klawiatury panelu programowania, joystick'a lub repliki manipulatora w zmniejszonej skali, a czasem nawet manipulatora o innej strukturze. Sygnały wytworzone przez dowolne z tych urządzeń wykorzystywane są do sterowania napędami robota podlegającego uczeniu.

Niezależnie od sposobu przemieszczania ramienia podczas uczenia istnieją dwie metody zapamiętywania trajektorii. W metodzie **PTP** (Point To Point) ramię przemieszczane jest do kolejnych charakterystycznych położeń na trajektorii, zatrzymywane tam i, poprzez wciśnięcie odpowiedniego przycisku na panelu programowania, kolejne miejsce jest zapamiętywane. Podczas odtwarzania trajektorii robot wykorzystuje jakąś formę interpolacji pomiędzy zapamiętanymi położeniami. W metodzie **CP** (Continuous Path) w trakcie przemieszczania ramienia, co ściśle określony kwant czasu, zapamiętywane są kolejne położenia. Ponieważ w tym przypadku kolejne położenia znajdują się bardzo blisko siebie, to zbędne stają się procedury interpolacji. W tym przypadku kolejne położenia mogą być odtwarzane ze zmienionym kwantem czasu, przez co uzyskuje się zmianę prędkości ruchu.

Główną zaletą uczenia jest prostota tej metody programowania. Nawet operator o niewielkich kwalifikacjach może zaprogramować robota tym sposobem. Niestety metoda ta, w jej czystej postaci, posiada wiele wad. Są nimi: trudność wykorzystania danych pochodzących z czujników, brak dokumentacji programu, kłopoty z modyfikacją już istniejących programów oraz, przede wszystkim, w trakcie uczenia robot nie jest wykorzystywany w procesie wytwórczym, a więc to drogie urządzenie jest bezproduktywne.

1.1.2. Programowanie off-line

U podstaw programowania off-line leżą metody tekstowe, bądź graficzne, wyrażania zadania, które ma zrealizować robot. Ponieważ metody graficzne są dopiero w koncepcyjnym etapie badań oraz ponieważ stanowią jedynie preprocesor dla metod tekstowych, należy się przede wszystkim skoncentrować na tych drugich. W przypadku metod tekstowych zadanie dla robota wyrażane jest w jakimś języku programowania robotów (JPR). Może to być język specjalnie zdefiniowany do programowania robotów lub też uniwersalny język stosowany do programowania komputerów. Zaletą JPR jest to, że podczas programowania nie wykorzystuje się robota, a więc robot w tym czasie może być wykorzystywany do produkcji, istnieje możliwość wykorzystania czujników oraz to, że w trakcie programowania powstaje dokumentacja.

Ponieważ program przygotowywany jest off-line, robot niezbędny jest jedynie do wczytania kodu wynikowego, a więc na krótką chwilę. Zmiana programu jego działania mogłaby trwać bardzo krótko. Niestety w produkcji oraz przy montażu robotów powstają pewne niedokładności. Dwie jednostki tego samego typu różnią się między sobą. Z tego powodu polecenie dwóm robotom osiągnięcie tego samego położenia (oczywiście rozdzielnie w czasie) zakończy się tym, że każdy z nich ustawi swe narzędzie w nieco innym miejscu. Oznacza to, że robot i jego program muszą być skalibrowane. Kalibracja stanowi obecnie intensywnie rozwijaną dziedzinę badań [26]. Do chwili zadowalającego rozwiązania powyższego problemu metody programowania off-line, które nie wykorzystują czujników do korekcji działań robota, nie będą mogły być stosowane w swej czystej formie. Prawdopodobnie bardziej będzie opłaciło się wykorzystanie czujników korekcyjnych niż precyzyjna kalibracja każdej jednostki użytej w linii produkcyjnej. Niezależnie od tego, które z podejść będzie zastosowane w przyszłości JPR będą jego istotną składową. Dlatego też JPR przyciągają uwagę badaczy.

1.1.3. Programowanie hybrydowe

Wady programowania on-line oraz off-line, w ich czystych postaciach, są częściowo usuwane przez programowanie hybrydowe. Metoda ta najczęściej bazuje na tekstowej formie zapisu programu oraz na wprowadzaniu niektórych argumentów instrukcji ruchu przez uczenie (n.p. `VAL II` [103]). W ten sposób programowanie w zasadzie nie absorbuje robota i możliwe jest skorzystanie z informacji uzyskanych przez czujniki, a ponadto otrzymuje się pełną dokumentację programu. Robota uczy się dochodzenia do nielicznych położeń wymagających dużej dokładności. Pozostałe położenia albo nie muszą być osiągnięte precyzyjnie albo są wyznaczane względem położeń nauczonych.

1.2. Sposoby implementacji języków programowania robotów

Jak widać z powyższych rozważań, niezależnie od tego, czy uczenie będzie wykorzystywane do lokalnej kalibracji, czy też robot zostanie skalibrowany innymi metodami, tekstowy sposób zapisu programu będzie konieczny. Dlatego też dużo wysiłku badawczego wkłada się zarówno w definiowanie języków programowania robotów jak i ich implementację. Języki programowania robotów mogą być implementowane na trzy sposoby [9]:

- jako języki specjalizowane,
- jako rozszerzenia istniejących języków uniwersalnych,
- bądź jako biblioteki istniejących języków uniwersalnych.

JPR zdefiniowany jako język specjalizowany albo będzie dostosowany do niewielkiej liczby klas zadań albo nabierze wszelkich cech języka uniwersalnego. W przypadku zbliżenia się do języka uniwersalnego wysiłek implementacyjny oraz trudność w szkoleniu programistów czynią tę drogę nieopłacalną. Z drugiej strony ograniczenie się do niewielu klas podzadań może być opłacalne jedynie wtedy, gdy mamy pewność, że robot nie będzie stosowany w innych przypadkach. Taka sytuacja bardzo często zachodzi w konkretnych gniazdach produkcyjnych. Niemniej jednak w przypadku prowadzenia różnorodnych badań nie można *a priori* stwierdzić do czego zostanie użyty robot. Dlatego też do celów badawczych obecnie raczej stosuje się uniwersalne języki programowania.

W tej sytuacji logicznym wydaje się rozszerzenie istniejącego języka uniwersalnego o instrukcje specyficzne dla robotów. Niestety rzadko kiedy kompilatory tych języków dostępne są w postaci źródłowej, a nawet jeżeli tak by było, to ze względu na przewidywaną różnorodność zadań nie jest w pełni jasne jakie instrukcje powinny być dodane. Na przykład rodzajów czujników może być bardzo wiele i każdy z nich musiałby mieć swoją listę instrukcji. Dodawanie nowych instrukcji przy rozszerzaniu zestawu czujników byłoby równie kłopotliwe jak w przypadku języka specjalizowanego. Dlatego też ta metoda implementacji nie jest stosowana.

Ostatnią możliwością jest wykorzystanie języka uniwersalnego zarówno do programowania robota jak i do stworzenia biblioteki procedur i procesów specyficznych dla robota oraz użytych czujników. Wtedy rozszerzenie zbioru urządzeń stanowiących system sprowadza się co najwyżej do dopisania do biblioteki nowych procedur obsługi. Ta metoda jest najbardziej pożądana, ponieważ w wyniku jej zastosowania otrzymujemy system otwarty, który może być przystosowany do prowadzenia dowolnych badań.

Oczywiście w przypadku stworzenia biblioteki program użytkownika będzie składał się z wywołań adekwatnych procedur, a więc jego składnia będzie mniej czytelna niż w przypadku odpowiadającego zapisu w języku specjalizowanym. Biorąc jednak pod uwagę kwalifikacje użytkowników – w tym przypadku inżynierów – nie stanowi to istotnego problemu.

W ostatnich latach definiowano zarówno nowe języki specjalizowane [5, 65], jak i implementowano biblioteki specyficznych dla robotów procedur napisanych w uniwersalnych językach programowania [4, 7, 23, 80]. Jak już wspomniano, wadą języków specjalizowanych jest ich hermetyczność. Konieczność dołączenia do systemu robotowego dodatkowego sprzętu (np. nowych czujników), zazwyczaj wymusza zmiany w definicji języka, a co za tym idzie i translator języka musi ulec modyfikacji. Nowe czujniki nie tylko wymagają nowych procedur obsługi (*driver'ów*), ale również informacja uzyskana dzięki nim musi zostać wykorzystana w inny sposób w procesie sterowania ruchem, wiec zmiany mogą być znaczne. Dlatego też, środowisko naukowe zajmujące się programowaniem robotów

obecnie preferuje, w zastosowaniach badawczych, biblioteki stanowiące rozszerzenia uniwersalnych języków programowania. Biblioteki te, również zwane skrótowo językami programowania robotów, umożliwiają konstruowanie systemów robotowych o otwartej strukturze. W przypadku rozbudowy sprzętowej takiego systemu, nowe procedury obsługi oraz sposoby wykorzystania informacji zapisywane są w języku uniwersalnym i dołączane do biblioteki (języka), przez co translator nie ulega modyfikacji. Jest to metoda o wiele szybsza i tańsza w implementacji.

Najczęściej używanymi językami uniwersalnymi stanowiącymi bazę do implementacji języków (bibliotek) programowania robotów są `Pascal`, `C`, a ostatnio także `C++`. Zaimplementowano w nich biblioteki do sterowania zarówno systemami jedno- jak i wielorobotowymi, np.: `RCCL` [23], `ARCL` [7], RCI [31], `KALI` [2, 24, 25, 35, 52], `RORC` [69, 71], `PASRO` [4, 5], `ROPAS` [74], `ROOPL` [70], `ZERO++` [53]. Przy tworzeniu większości z tych języków posługiwano się doświadczeniami praktycznymi i intuicją naukową. Niniejsza praca zawiera rozważania teoretyczne potwierdzające zasadność przyjętej struktury układu sterowania robotem IRp-6, a przez to i struktury biblioteki, czyli składnika programowego takiego systemu. Dla ogólności rozważań oraz uniwersalności projektowanego systemu, przyjęto, że robot IRp-6 będzie traktowany jako jeden z wielu robotów, którymi ten układ będzie sterował.

1.3. Środki implementacji

Niezależnie od tego, czy zdecydujemy się na zdefiniowanie języka specjalizowanego, czy też zadowolimy się biblioteką dla uniwersalnego języka programowania, musimy wybrać zarówno język implementacji jak i metodę programowania. Szczególnie w przypadku implementacji biblioteki wybór języka jest istotny, ponieważ język ten będzie jednocześnie językiem implementacji i językiem programowania robota, a więc językiem, którym będzie posługiwał się użytkownik. Podobnie jest z wyborem metody programowania.

Przyjęta metoda programowania ma zasadnicze znaczenie dla szybkości tworzenia implementacji oraz jej niezawodności. Przyjmuje się, że podział programu na małe moduły, o nielicznych interakcjach, sprzyja zarówno unikaniu błędów programistycznych jak i uzyskaniu przejrzystej struktury programu, a więc ułatwia jego przyszłe modyfikacje oraz wprowadzanie poprawek. Aby uzyskać program w postaci modularnej, należy unikać stosowania instrukcji skoku, a więc należy zastosować metodę programowania strukturalnego. **Programowanie strukturalne** jest metodą zapisu zadania w postaci hierachii modułów. Każdy kolejny poziom modułów przedstawia zadanie w większych szczegółach, aż w końcu osiągnie się program wykonalny (programowanie metodą zstępującą). Istnieją dwie odmiany programowania strukturalnego:

- programowanie proceduralne oraz
- programowanie obiektowe.

W przypadku **programowania proceduralnego** dane i kod programu działający na tych danych stanowią oddzielne części. Kod programu zapisywany jest w postaci procedur. Procedury działając na tych danych przekształcają je aż do osiągnięcia wyniku lub wykrycia błędu. Wpierw definuje się struktury danych (zmienne), na których będzie działał program, a następnie określa się procedury, w najbardziej ogólnej postaci, działające na tych danych. Etap definiowania procedur powtarza się dzieląc każdą ogólną procedurę na szereg procedur bardziej szczegółowych, aż osiągnie się stopień szczegółowości umożliwiający zapisanie działania procedur najniższego poziomu jedynie za pomocą instrukcji i wyrażeń zastosowanego języka.

Rozdzielność danych i procedur na nich działających prowadzi do niebezpieczeństwa użycia procedury do niewłaściwych danych. Dlatego też w przypadku **programowania obiektowego** dane i procedury na nich działające zostały zintegrowane w jedną całość, tzn. obiekty. **Obiektem** jest zespół danych (zmiennych, które należy traktować jak pola rekordu) oraz zbiór funkcji (zwanych **metodami**) działających na tych danych. Programowanie obiektowe ma następujące własności [102]:

- **kapsułkowanie** — traktowanie grup danych oraz kodu programu działającego na nich jako oddzielnych tworów – obiektów,
- **dziedziczenie** — zdefiniowanie hierarchii obiektów, w której każdy obiekt potomny dziedziczy wszystkie własności obiektu przodka (np. dostęp do jego danych i metod) oraz uzyskuje dodatkowe własności specyficzne dla nowo tworzonego obiektu,
- **polimorfizm** — użycie tej samej nazwy do określenia działań przeprowadzanych na różnych obiektach spokrewnionych przez dziedziczenie (działania te są semantycznie podobne, ale są realizowane w sposób specyficzny dla każdego obiektu hierarchii),
- **wyjątki** — rozdzielenie kodu dla sytuacji bezawaryjnego przebiegu obliczeń oraz kodu obsługi błędów (obsługa wyjątków),
- **przeciążanie** — użycie tego samego operatora ze zmienioną semantyką dla nowych typów danych.

Programowanie obiektowe zakłada, że programista zdefiniuje pewne abstrakcyjne obiekty, które będą odzwierciedlały obiekty występujące w zadaniu. Te abstrakcyjne obiekty mają swe własności (wartości danych) oraz przejawiają pewne zachowania pod wpływem zastosowania ich metod do ich danych. Program jest pisany w kategoriach obiektów zachowujących się w taki sposób, aby zmienić swe własności, czyli stosuje się metody, aby zmienić wartości danych (zmiennych).

1.4. Podejście zastosowane do programowania robota IRp-6

Z powyższych rozważań wynika, że sterownik robota, na którym mają być przeprowadzane różnorodne badania, powinien mieć strukturę otwartą, umożliwiającą modyfikacje i zmiany poszczególnych jego części. Powinna istnieć możliwość dołączania czujników różnych typów oraz sporządzania dokładnej dokumentacji programu. Z tych wymagań wynika, że powinno się zastosować metodę programowania off-line. Zmiany w niektórych częściach sterownika nie powinny powodować konieczności wprowadzania zmian w innych jego częściach. Stąd konieczność zastosowania programowania strukturalnego.

Ponieważ projektowany sterownik miał mieć strukturę otwartą, a więc taką, w której wiadomo, że użytkownik będzie dokonywał modyfikacji, w zależności od problemu badawczego, którym będzie się zajmował, istotnym było, by ta struktura była przejrzysta i mało podatna na wprowadzenie błędów przy dokonywaniu zmian. Z tego powodu zdecydowano się użyć techniki programowania obiektowego. Zastosowanie tej techniki jest możliwe dzięki wykorzystaniu języka `C++` [8, 40, 102]. Takie własności programowania obiektowego jak: kapsułkowanie kodu i danych, na których on działa, obsługa błędów przez wyjątki, przeciążanie operatorów i nazw funkcji, wartości domyślne parametrów wywołań funkcji oraz mechanizm dziedziczenia niewątpliwie upraszczają strukturę oraz ułatwiają korzystanie ze stworzonego oprogramowania. Własności te okazały się szczególne przydatne przy realizacji niższych warstw sterowania robotem IRp-6. Dlatego też zdecydowano sie zastosować programowanie obiektowe również do warstw wyższych i przekształcić `MRROC+` w `MRROC++`. Tak więc robot IRp-6 [3] jest traktowany jako element sys-

temu wielorobotowego programowanego off-line przy wykorzystaniu obiektowej biblioteki modułów zapisanych w C++.

Należy tu przypomnieć, że język C++ został zasadniczo stworzony do pisania programów wykonywanych sekwencyjnie na jednym komputerze. Wykorzystanie techniki programowania obiektowego oraz języka C++ w rozproszonym środowisku wielokomputerowym, w którym część procesów działa w podziale czasu na tym samym procesorze, a część wykorzystuje inne węzły sieci jest pewnym wyzwaniem zarówno projektowym jak i badawczym.

Rozdział 2

Ogólna struktura sterownika MRROC++

2.1. Dekompozyja systemu

System robotyczny można zdekomponować na trzy główne części: **efektory** e, **receptory** r (czujniki rzeczywiste) oraz **podsystem sterowania** c. Dla potrzeb tej pracy oznaczenie części systemu oraz ich stanu nie będą odróżniane, ponieważ kontekst czyni rozróżnienie ewidentnym.

$$s =< e, r, c >, \qquad s \in S, \quad e \in E, r \in R, \quad c \in C, \tag{2.1}$$

gdzie:

e	jest stanem efektorów,	E	jest przestrzenią stanów efektorów,
r	jest stanem receptorów,	R	jest przestrzenią stanów receptorów,
c	jest stanem podsystemu sterowania,	C	jest przestrzenią stanów podsystemu sterowania.

Dane dostarczane przez rzeczywiste czujniki muszą zostać przekształcone (zagregowane) do postaci użytecznej w sterowaniu, a więc w odczyty **czujników wirtualnych**:

$$v = f_v(r, c, e), \quad v \in V \tag{2.2}$$

gdzie V jest przestrzenią odczytów czujników wirtualnych. Funkcja f_v nazywana jest **funkcją agregującą** dane z czujników rzeczywistych. Funkcja ta zazwyczaj zależy od r, rzadziej od c, a niezmiernie rzadko od e. Niemniej jednak, dla kompletności, wymieniono jej wszystkie trzy potencjalne argumenty.

Głównymi instrukcjami języków programowania robotów są **instrukcje ruchowe**. Powodują one przemieszczanie efektorów, a w konsekwencji zmianę stanu środowiska, w którym operuje system robotyczny. W tych rozważaniach ich forma syntaktyczna ma znaczenie drugorzędne. Zależy ona od sposobu implementacji tych instrukcji. W przypadku języków specjalizowanych ich składnia będzie dostosowana do gramatyki języka specjalizowanego, natomiast w przypadku zanurzenia języka robota w języku uniwersalnym, składnia tych instrukcji będzie odzwierciedlała syntaktykę wywołania procedur lub funkcji danego języka uniwersalnego. Pierwszorzędne znaczenie ma notomiast semantyka instrukcji ruchowych.

Jednym z głównych zadań projektanta systemu robotycznego jest dostarczenie właściwych narzędzi do jego programowania. Zadanie to nie jest proste, gdyż, jak to już wielokrotnie

podkreślano, zarówno konfiguracja sprzętowa systemu jak i zbiór czynności wykonywanych przez system mogą ulegać zmianie, natomiast narzędzia programistyczne powinny pozostać bez zmian. Aby projektowane narzędzie programistyczne – język programowania robotów – było łatwe w użyciu, liczba jego instrukcji ruchowych musi być ograniczona. Z drugiej strony zdefiniowane instrukcje ruchowe muszą być dostatecznie ogólne, aby pokrywać całe spektrum możliwych do zaistnienia sytuacji. W przypadku, gdy efektorami są ramiona robotów należy się skoncentrować na dwóch aspektach problemu:

- sposobie wyrażania stanu efektorów oraz
- sposobie wykorzystania czujników.

2.2. Sposób wyrażania stanu efektorów

Problem wyrażania stanu efektorów jest nierozerwalnie związany z językami programowania robotów. Każdy język programowania operuje pewnymi abstrakcyjnymi pojęciami. Instrukcja języka składa się ze słowa kluczowgo lub specyficznego symbolu pełniącego taką rolę oraz listy argumentów – potencjalnie pustej. Argumenty te reprezentują pojęcia abstrakcyjne. Języki programowania komputerów operują zmiennymi różnych typów. Wartości tych zmiennych reprezentują stan pewnych pojęć abstrakcyjnych. Najistotniejszymi instrukcjami języków programowania robotów są polecenia wykonania ruchu przez efektory – czyli **instrukcje ruchowe**. Pojęcia abstrakcyjne, do których odnoszą się te instrukcje to: wzajemne położenia poszczególnych członów manipulatorów, położenia końcówek robotów lub obiekty znajdujące sie w orzestrzeni roboczej. Te instrukcje, a więc i języki programowania robotów, mogą być sklasyfikowane w zależności od pojęć abstrakcyjnych, do których się odwołują.

Języki najniższego poziomu zwane są **językami zorientowanymi na przemieszczanie fragmentów łańcucha kinematycznego**. Trudność związana z określeniem położenia narzędzia w przestrzeni roboczej na podstawie wartości argumentów instrukcji języków tego poziomu oraz względna łatwość rozwiązania prostego i odwrotnego zagadnienia kinematyki dla stosowanych konstrukcji manipulatorów spowodowała zanik języków tego poziomu.

Języki następnego poziomu odnoszą się do końcówki manipulatora, więc nazwano je **językami zorientowanymi na przemieszczanie końcówki manipulatora** (n.p. `WAVE` [36], `AML` [51, 5], `RCCL` [23, 31], `KALI` [24, 25], `RORC` [80, 69, 71]). Instrukcje **języków zorientowanych na przemieszczanie obiektów** operują modelami obiektów istniejących w otoczeniu robota (n.p. `AL` [34, 67], `RAPT` [1, 37, 67], `TORBOL` [80, 67, 65], `SRL` [5], `PASRO` [4, 5]). W tym przypadku programista jedynie określa, które obiekty powinny zostać przemieszczone, aby zostało zrealizowane zadanie. Układ sterujący robota, używając swej wiedzy o obiektach oraz relacjach zachodzących między nimi, przemieszcza manipulator w taki sposób, aby zadanie zostało wykonane.

Językami ostatniego poziomu są **języki zorientowane na wykonanie zadania**. W tym przypadku program wyraża jedynie ogólny cel, który ma być osiągnięty, pomijając dokładną specyfikację operacji, które mają być wykonane (e.g. `AUTOPASS` [30] i w niewielkim stopniu `RAPT` [1, 37, 67]). W związku z tym to układ sterowania sam musi wygenerować plan realizacji zadania, a następnie musi go zrealizować. Niestety różnorodność konfiguracji sprzętowych systemów oraz środowisk, w których operują roboty, jest tak duża, że opracowanie uniwersalnych generatorów planów dla takich systemów przekracza aktualne możliwości techniczne. Należy podkreślić, że nawet próby skonstruowania takich systemów dla pojedynczego robota i ograniczonej klasy zadań powiodły się tylko częściowo.

Główny problem stanowią niedokładności w wykonaniu obiektów oraz ich pozycjonowaniu. Oczywiście próbuje się ograniczyć problemy związane z niedokładnościami przez zastosowanie czujników, ale wtedy plany akcji muszą obejmować dużo więcej potencjalnych dróg rozwiązań, przez co stają się nadmiernie rozbudowane. Te trudności oraz fakt, że programista zazwyczaj nie ma większych problemów z dostarczeniem planu akcji, spowodowały zmniejszenie zainteresowania językami zorientowanymi na wykonanie zadania. Istnieją również próby ominięcia tego problemu poprzez zastosowanie sterowania behawioralnego [6, 32] lub reakcyjnego [75, 76, 82, 83], ale do oprogramowania systemu w taki sposób, aby przejawiał zachowania reakcyjne, także trzeba użyć jakiegoś języka programowania robotów – wtedy najczęściej stosuje się języki zorientowane na przemieszczanie końcówki manipulatora.

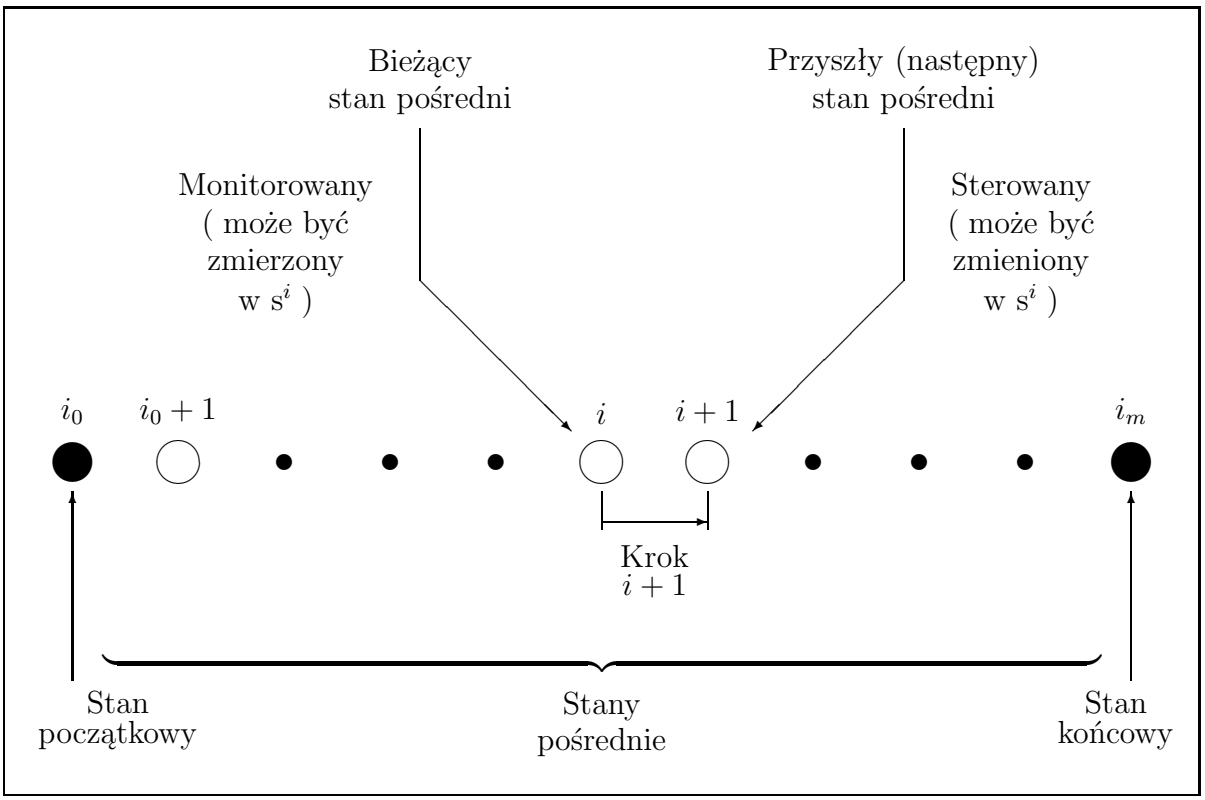
O ile powstał szereg języków zorientowanych na przemieszczanie obiektów dla pojedynczych robotów (n.p. `AL` [34, 67], `RAPT` [67, 1], `TORBOL` [80, 67, 65], `SRL` [5], `PASRO` [5]), to bardzo niewiele jest danych literaturowych na temat tego typu języków dla systemów wielorobotowych bądź wieloefektorowych. Główną przyczyną tego stanu rzeczy jest problem z automatycznym określeniem, który z efektorów powinien być w danej chwili odpowiedzialny za przemieszczanie, którego z obiektów. Na tym poziomie programista nie odwołuje się do robotów, ale jedynie do obiektów, więc efektory, jako abstrakcyjne pojęcia funkcjonujące w programie, znikają z jego widoku. Ponadto w językach tego poziomu występują te same trudności z niedokładnościami pozycjonowania i wykonania obiektów co na poziomie języków zorientowanych na wykonanie zadania. Inkorporacja czujników do języków zorientowanych na przemieszczenia obiektów jest utrudniona faktem, że czujniki są raczej związane z robotem niż z obiektami. Dodatkowo, roboty wykonują wiele zadań, w których obiekty z otoczenia odgrywają rolę drugorzędną (n.p. spawanie czy malowanie natryskowe).

Warto również zwrócić uwagę, że proces przetwarzania (kompilacji lub interpretacji) języka wyższego poziomu daje w wyniku język niższego poziomu. W związku z tym, tak długo jak co najmniej większość problemów związanych z implementacją języków niższego poziomu nie będzie rozwiązana w zadowalający sposób, nie wydaje się być celowym koncentrowanie się na językach poziomów wyższych. Ponieważ postanowiono, że układ sterowania robotem IRp-6 powinien być zdolny sterować systemem wieloefektorowym [77] oraz ponieważ problemy związane z inkorporacją różnorodnych czujników i wykorzystaniem danych z nich otrzymywanych do sterowania ruchem nie zostały jeszcze w pełni rozwiązane, zdecydowano się dla konstruowanego systemu sterowania przyjąć, że będzie on programowany językiem zorientowanym na przemieszczanie efektorów (równoważnym językom zorientowanym na przemieszczanie końcówki manipulatora). Ponadto, ponieważ język ten będzie musiał być przetwarzany na język niższego poziomu, nie ma powodu, aby stan efektorów nie mógł być równocześnie wyrażany w postaci specyficznej dla języków niższego poziomu. W konsekwencji przyjęto, że stan efektorów e będzie mógł być wyrażany jako:

- zestaw położeń wałów poszczególnych silników (siłowników),
- zestaw kątów pomiędzy poszczególnymi fragmentami łańcucha kinematycznego,
- trzy współrzędne kartezjańskie początku układu związanego z narzędziem efektora oraz zestaw trzech kątów określających orientację tego układu (mogą to być n.p. kąty Eulera) — rozwiązanie dla efektorów (w tym przypadku dla robotów) mających sześć stopni swobody.

2.3. Wykorzystanie czujników do sterowania robotem

Zadaniem systemu robotycznego jest wykonanie zadania opisanego programem dostarczanym przez użytkownika. Instrukcje ruchowe tego programu powodują zmiany stanu efektrorów e. Wykonanie instrukcji ruchowej rozpoczyna się w **stanie początkowym**, kończy w **stanie końcowym** oraz powoduje przejście systemu poprzez szereg **stanów pośrednich**. Ponieważ obecne systemy robotyczne sterowane są komputerowo, wykonanie każdej instrukcji może być podzielone na kroki. Każdy **krok** powoduje zmianę stanu z jednego stanu pośredniego do następnego.



Rys. 2.1: Stany systemu w trakcie realizacji instrukcji ruchu

W każdym stanie pośrednim (lub w trakcie jego osiągania) stan systemu może zostać zmierzony – może być **monitorowany** przez receptory. Aktualny stan systemu może być jedynie zmierzony, czyli monitorowany, natomiast na przyszłe systemu można wpływać, a więc mogą być **sterowane** (rys. 2.1). Stan początkowy może być traktowany jako aktualny stan pośredni na początku wykonania instrukcji, natomist stan końcowy jako ostatni stan pośredni realizacji polecenia. Ponieważ w ten sposób stan początkowy i końcowy stają się szczególnymi przypadkami stanów pośrednich, można skoncentrować uwagę jedynie na stanach pośrednich, rozróżniając tylko stan aktualny i te, które mają jeszcze być osiągnięte. Stany, które zostały zrealizowane uprzednio, nie są istotne z punktu widzenia sterowania, bo istotna informacja o nich jest skumulowana w aktualnym stanie systemu.

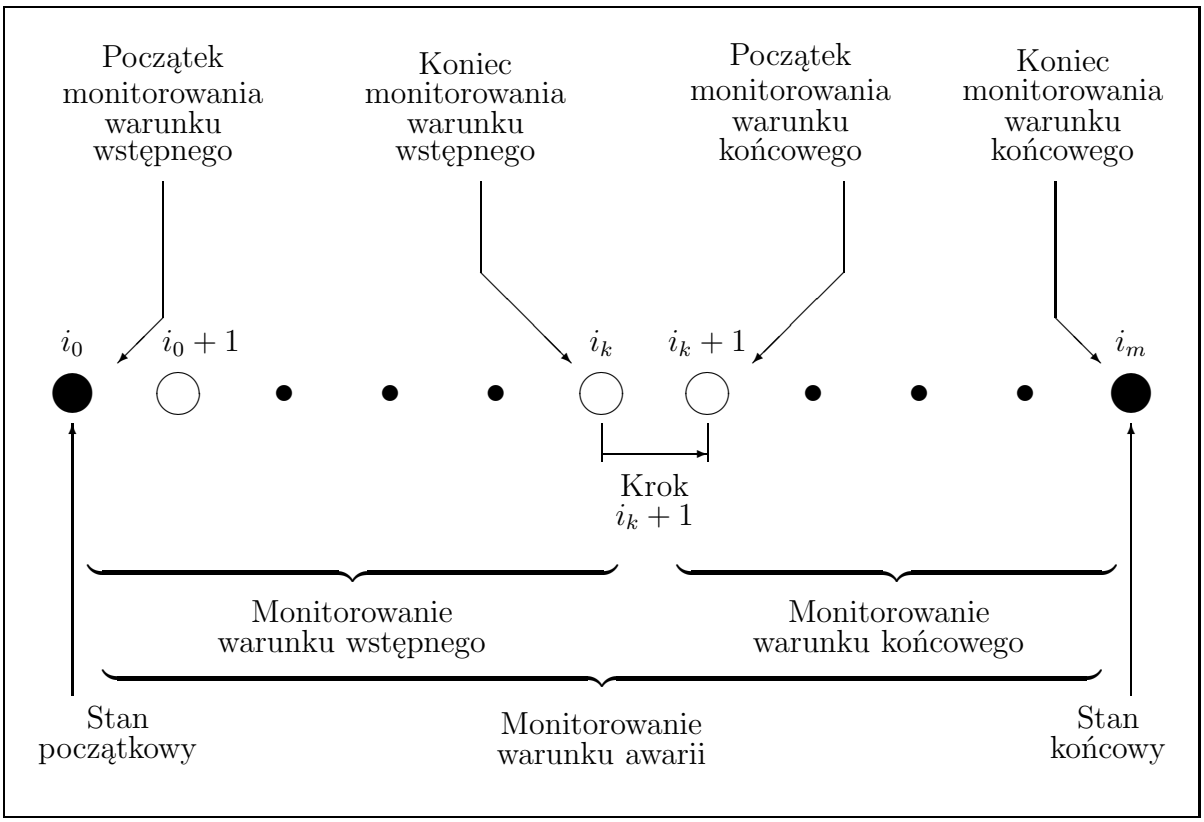
Monitorowanie i sterowanie są nierozerwalnie związane z wykonaniem instrukcji ruchowych. Trzy powody monitorowania stanu mogą być wymienione:

- monitorowanie warunku wstępnego,
- monitorowanie warunku końcowego,

• monitorowanie warunku awarii.

W aktualnym stanie pośrednim sprawdza się spełnienie tych warunków. **Monitorowanie warunku wstępnego** rozpoczyna się w stanie początkowym i powoduje wykonanie kolejnych kroków w oczekiwaniu na spełnienie tego waruneku. Jeżeli warunek wstępny jest spełniony przed rozpoczęciem wykonania instrukcji, liczba kroków tej fazy realizacji polecenia wynosi zero. Jeżeli warunek wstępny zostanie spełniony, to przez pozostałe kroki realizacji instrukcji sprawdzane jest spełnienie warunku końcowego. **Monitorowanie warunku końcowego** jest przerywane w momencie jego spełnienia, a wykonanie instrukcji w tym momencie dobiega końca (rys. 2.2). Tutaj także liczba wykonanych kroków zależy od momentu spełnienia warunku, ale tym razem – końcowego. Ponadto należy wziąć pod uwagę fakt, że w dowolnym momencie realizacji instrukcji mogą wystąpić błędy (awaria). Dlatego też, musi być **monitorowany warunek awarii**.

Jak już wspomniano, przyszłe stany pośrednie mogą być **sterowane**, tzn. albo modyfikowane w stosunku do stanów planowanych albo generowane. W obu przypadkach odczyty czujników wirtualnych mogą być wykorzystywane.



Rys. 2.2: Monitorowanie wykonania instrukcji ruchu przez czujniki

Niech i_0 będzie numerem stanu początkowego wykonania instrukcji, natomiast kolejne stany pośrednie mają numery $i = i_0 + 1, \ldots, i_m$, gdzie i_m jest numerem stanu końcowego. Jeżeli system wykonał i kroków, to znajduje się w stanie s^i i następny stan efektorów e^{i+1} jest obliczany za pomocą **funkcji przejścia** f_e.

Monitorowanie warunku wstępnego polega wykonywaniu przez system kolejnych kroków w oczekiwaniu na spełnienie warunku wstępnego, tak aby ruch mógł się rozpocząć. Semantyka monitorowania warunku wstępnego może być określona w następujący spo-

sób:

$$e^{i+1} = \begin{cases} e^i = e^{i_0} & \text{gdy} \quad f_I(c^i, e^i, v^i) = false \quad \bigwedge \quad f_E(c^i, e^i, r^i) = false \\ e^{i_k} = e^{i_0} & \text{gdy} \quad f_I(c^i, e^i, v^i) = true \quad \bigwedge \quad f_E(c^i, e^i, r^i) = false \\ e^{i_{k_*}} = e^{i_0} & \text{gdy} \quad f_E(c^i, e^i, r^i) = true \end{cases}$$
$$\text{dla } i = i_0, \ldots, i_k, \quad i_{k_*} \leq i_k$$

gdzie:

$f_I(c^i, e^i, v^i)$ jest warunkiem wstępnym,

$f_E(c^i, e^i, r^i)$ jest warunkiem awarii,

i_0 jest numerem pierwszego kroku wykonania instrukcji,

i_k jest numerem kroku, w którym $f_I(c^i, e^i, v^i)$ staje się $true$, więc monitorowanie warunku wstępnego jest kończone,

i_{k_*} jest numerem kroku, w którym $f_E(c^i, e^i, r^i)$ staje się $true$, więc monitorowanie warunku wstępnego musi zostać przerwane.

Monitorowanie warunku końcowego polega na zmienianiu stanu systemu aż warunek końcowy zostanie spełniony. Semantykę monitorowania warunku końcowego można zapisać w następujący sposób:

$$\begin{cases} e^{i+1} = f_e(c^i, e^i) & \text{gdy:} \quad f_T(c^i, e^i, v^i) = false \quad \bigwedge \quad f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_m} & \text{gdy:} \quad f_T(c^i, e^i, v^i) = true \quad \bigwedge \quad f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_{m_*}} & \text{gdy:} \quad f_E(c^i, e^i, r^i) = true \end{cases}$$
$$\text{dla } i = i_0, \ldots, i_m, \quad i_{m_*} \leq i_m$$

gdzie:

$f_e(c^i, e^i)$ jest funkcją przejścia efektorów (nie zależy od v^i, ponieważ w tym przypadku stan jest jedynie monitorowany – nie jest zmieniany przy użyciu odczytów czujników),

$f_T(c^i, e^i, v^i)$ jest warunkiem końcowym,

i_m jest numerem kroku, w którym $f_T(c^i, e^i, v^i)$ staje się $true$, więc monitorowanie warunku końcowego jest kończone,

i_{m_*} jest numerem kroku, w którym $f_E(c^i, e^i, r^i)$ staje się $true$, więc monitorowanie warunku końcowego musi zostać przerwane.

Sterowanie przyszłymi stanami pośrednimi zazwyczaj łączone jest z monitorowaniem warunku końcowego, więc semantyka instrukcji wykonującej obie te czynności razem może być zdefiniowana następująco:

$$\begin{cases} e^{i+1} = f_e^*(c^i, e^i, v^i) & \text{gdy:} \quad f_T(c^i, e^i, v^i) = false \quad \bigwedge \quad f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_m} & \text{gdy:} \quad f_T(c^i, e^i, v^i) = true \quad \bigwedge \quad f_E(c^i, e^i, r^i) = false \\ e^i = e^{i_{m_*}} & \text{gdy:} \quad f_E(c^i, e^i, r^i) = true \end{cases}$$
$$\text{dla } i = i_0, \ldots, i_m, \quad i_{m_*} \leq i_m$$

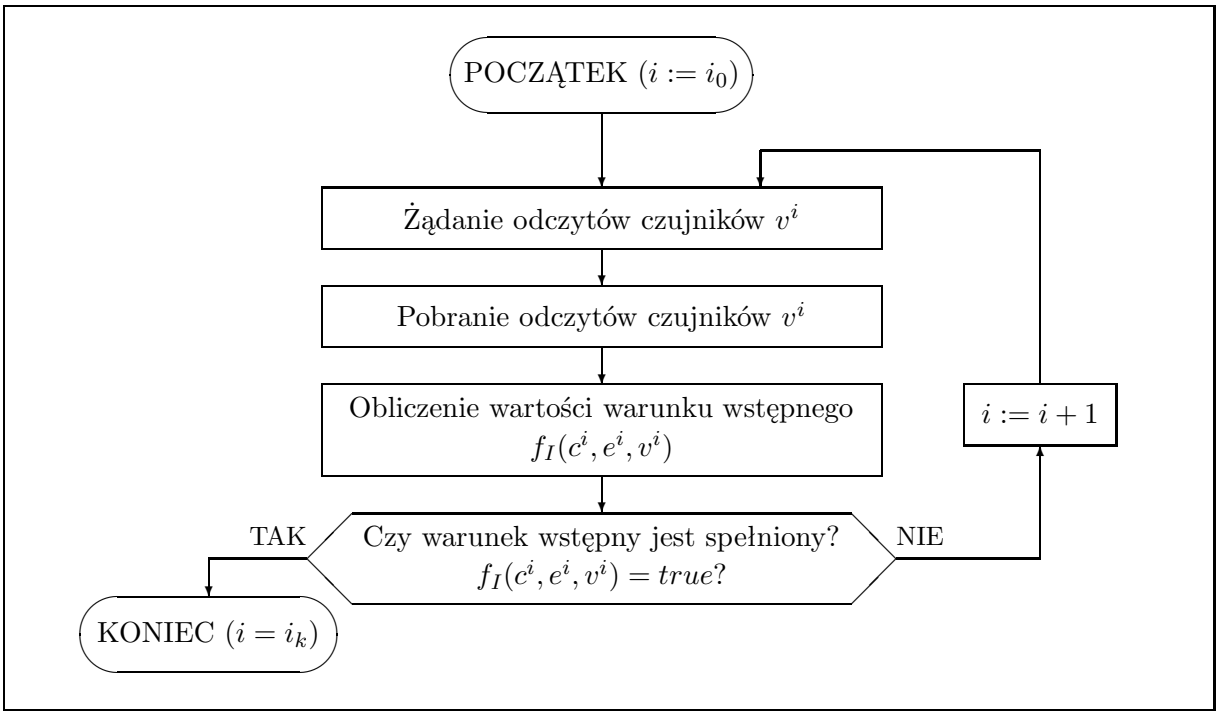
gdzie:

$f_e^*(c^i, e^i, v^i)$ jest funkcją przejścia efektorów (zależy od v^i, ponieważ stan jest nie tylko monitorowany, ale również zmieniany wykorzystując odczyty czujników).

Jeżeli (**??**), (**??**), i (**??**) zostaną połączone, to semantyka najbardziej ogólnej postaci instrukcji ruchu przybierze następującą formę:

$$\begin{cases} e^{i+1} = e^i = e^{i_0} & \text{gdy: } f_I(c^i, e^i, v^i) = false \ \bigwedge \ f_E(c^i, e^i, r^i) = false \\ & \text{dla } i = i_0, \ldots, i_k \\ e^i = e^{i_k} & \text{gdy: } f_I(c^i, e^i, v^i) = true, \\ & \text{dla } i = i_k \\ e^{i+1} = f_e^*(c^i, e^i, v^i) & \text{gdy: } f_T(c^i, e^i, v^i) = false \ \bigwedge \ f_E(c^i, e^i, r^i) = false \\ & \text{dla } i = i_k, \ldots, i_m \\ e^i = e^{i_m} & \text{gdy: } f_T(c^i, e^i, v^i) = true, \\ & \text{dla } i = i_m \\ e^i = e^{i_{m*}} & \text{gdy: } f_E(c^i, e^i, r^i) = true, \\ & \text{dla } i = i_{m*} \end{cases} \tag{2.3}$$

dla: $i = i_0, \ldots, i_k, \ldots, i_m; \quad i_0 \leq i_{m*} \leq i_m$



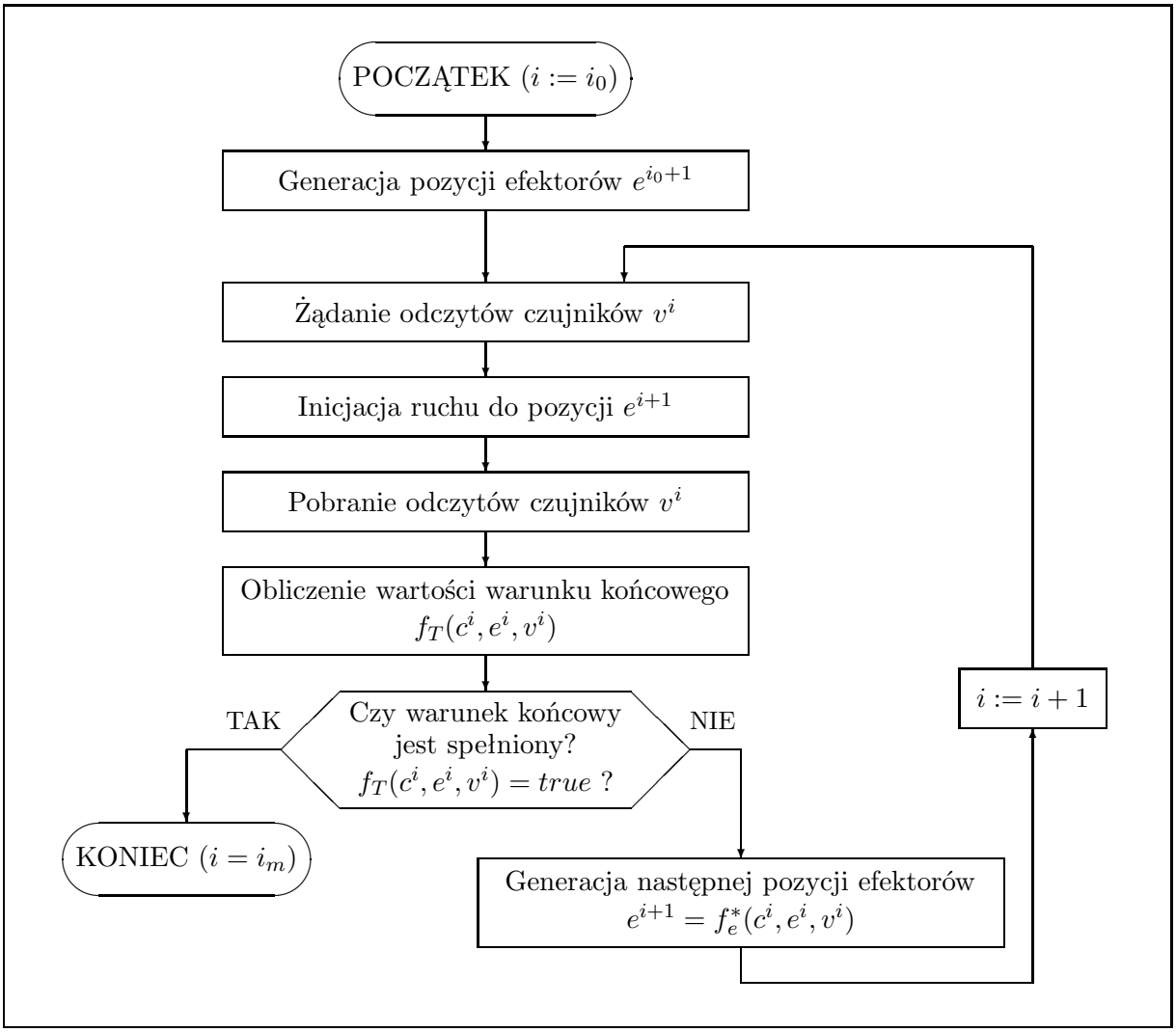
Rys. 2.3: Sieć działań instrukcji `Wait`

Zazwyczaj najbardziej ogólnej semantyki się nie implementuje, ponieważ rzadko się zdarza, aby monitorowanie warunku wstępnego i końcowego musiało być zrealizowane w jednej instrukcji ruchowej. Dlatego najczęściej implementuje się dwie instrukcje:

`Wait` – monitorującą warunek wstępny oraz

`Move` – monitorującą warunek końcowy z jednoczesnym sterowaniem przyszłych stanów pośrednich.

Instrukcję `Wait` traktuje się jako instrukcję ruchową, ponieważ najczęściej powoduje ona oczekiwanie na zajście pewnego zdarzenia w otoczeniu robota. Zdarzenie to może być związane z wykryciem przez czujniki przemieszczenia obiektów przez czynniki zewnętrzne względem systemu, a więc system oczekuje na relizację ruchu. Jeśli pominie się przyczynę ruchu, to skutek jest taki sam – wystąpił ruch obiektów, więc zarówno `Move` jak i `Wait` traktuje się jak instrukcje ruchowe.



Rys. 2.4: Sieć działań instrukcji `Move`

Semantyka instrukcji `Wait` określona jest przez zależności (**??**), natomiast sieć działań jej wykonania przedstawia rys. 2.3. Semantyka instrukcji `Move` zdefiniowana jest zależnościami (**??**), natomiast sieć działań jej wykonania przedstawia rys. 2.4. Należy zwrócić uwagę, że zależności (**??**) i (**??**) biorą pod uwagę możliwość wystąpienia awarii, natomiast fakt ten nie został odnotowany na rysunkach 2.3 i 2.4. Przewiduje się, że monitorowanie warunku awarii będzie realizowane jako obsługa wyjątków i w związku z tym umieszczenie tego warunku w sieci działań jedynie zaciemniłoby obraz bezbłędnego wykonania instrukcji.

Rozdział 3

Struktura oprogramowania sterownika

3.1. Uzasadnienie teoretyczne przyjętej struktury

Podsystem sterujący jest odpowiedzialny za obliczanie trajektorii ruchu efektorów (w tym przypadku ramion robotów) używając do tego celu wewnętrznie zapamiętanych danych, aktualnego stanu efektorów oraz odczytów czujników wirtualnych. Stan całego systemu s można zdekomponować na stany poszczególnych efektorów oraz wziąć pod uwagę, że w sterowaniu korzysta się z odczytów czujników wirtualnych v, a nie bezpośrednio z danych dostarczanych przez czujniki rzeczywiste r. W konsekwencji

$$s = < e_1, \ldots, e_{n_e}, v_1, \ldots, v_{n_v}, c > \tag{3.1}$$

gdzie:
n_e jest liczbą efektorów systemu ($e =< e_1, \ldots, e_{n_e} >$) oraz
n_v jest liczbą czujników wirtualnych ($v =< v_1, \ldots, v_{n_v} >$).
Aby obliczyć następny stan efektorów, podsystem sterowania musi przetworzyć dane wspólne dla wszystkich współpracujących efektorów oraz wykonać obliczenia specyficzne dla każdego z robotów. Oczywiście całość obliczeń może być wykonana przez jeden scentralizowany system obliczeniowy, ale dużo lepszym rozwiązaniem jest podzielenie podsystemu sterowania na $n_e + 1$ części:

$$c = < c_0, c_1, \ldots, c_{n_e} > \tag{3.2}$$

Każdy z podsystemów c_j, $j = 1, \ldots, n_e$, odpowiedzialny jest za sterowanie pojedynczym efektorem, natomiast podsystem c_0 koordynuje pracę wszystkich efektorów. Bazując na powyższych rozważaniach formalnych otrzymujemy następującą strukturę systemu. Każdemu efektorowi $e_j, j = 1, \ldots, n_e$ przyporządkowujemy *Effector Control Process* ECP_l. Jego stan wyrażany jest przez c_j, $j = 1, \ldots, n_e$. Proces koordynujący nazywany jest *Master Process* MP, a jego stan wyrażany jest przez c_0. Każda z części c_j podsystemu sterowania, wspólnie z częścią c_0, odpowiedzialna jest za obliczenie następnego stanu (stan obliczony: e_{c_j}) efektora e_j oraz doprowadzenie do zrównania e_j i e_{c_j}, czyli realizację kroku ruchu. Traktując system jako system z czasem dyskretnym, można wyznaczyć następny stan każdego z efektorów jako funkcję przejścia:

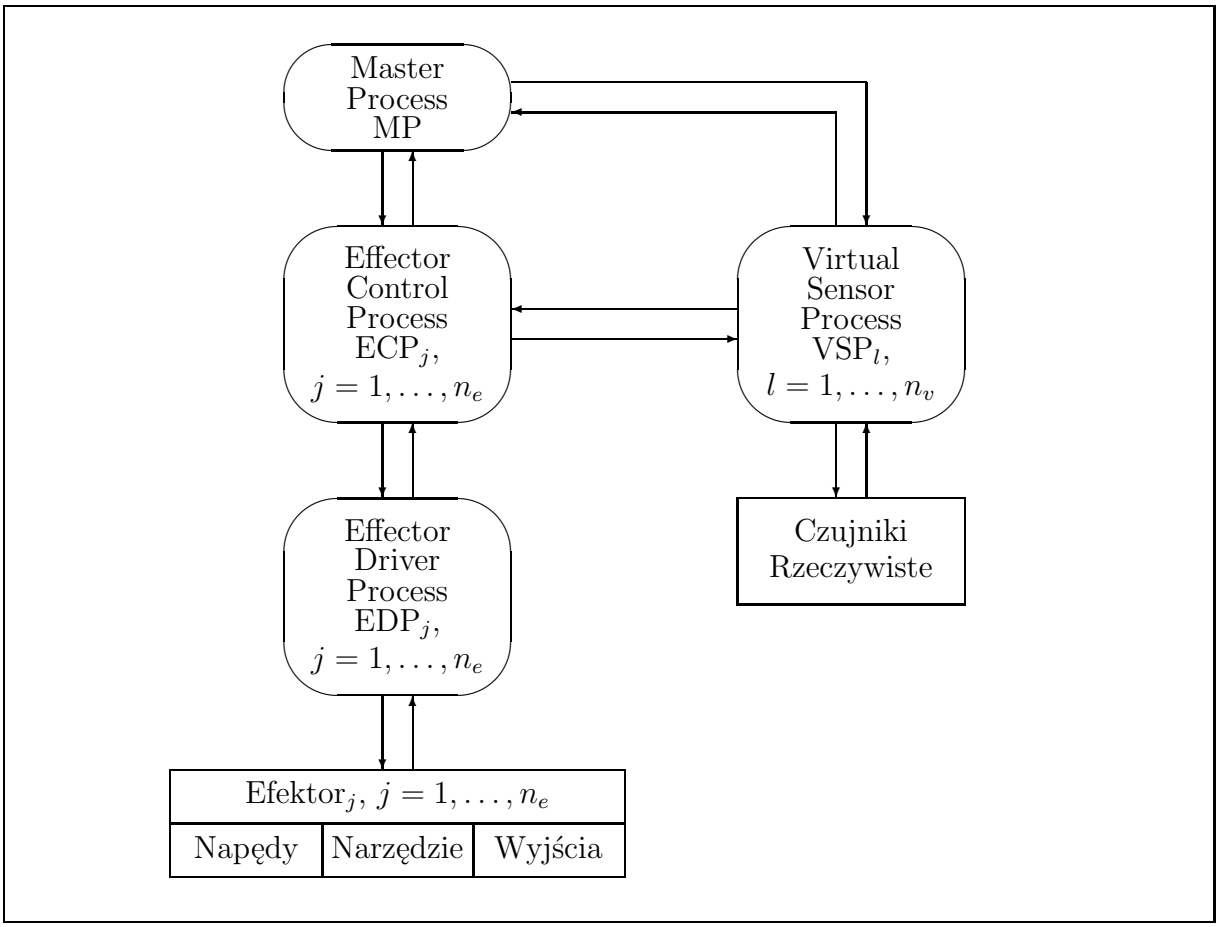
$$e_{c_j}^{i+1} = f_{e_j}(e_l^i, v_1^i, \ldots, v_{n_v}^i, c_0^i, c_j^i), \quad j = 1, \ldots, n_e \tag{3.3}$$

Te funkcje przejścia będą zastępować funkcję f_e^* (z zależności (**??**)) w instrukcjach `Move` umieszczonych w ciałach procesów ECP_j. Każdy proces ECP_j oblicza wartość funkcji

przejścia (3.3) na podstawie danych uzyskanych z czujników wirtualnych v^i, a więc danych wytwarzanych przez oddzielne procesy VSP (*Virtual Sensor Process*), oraz danych umożliwiających koordynację pracy efektorów, uzyskanych z procesu MP. Jednocześnie ulega zmianie stan wewnętrzny części składowych podsystemu sterowania:

$$\begin{aligned} c_0^{i+1} &= f_{c_0}(e_1^i, \ldots, e_{n_e}^i, v_1^i, \ldots, v_{n_v}^i, c_0^i, c_1^i, \ldots c_{n_e}^i) \\ c_j^{i+1} &= f_{c_j}(e_j^i, v_1^i, \ldots, v_{n_v}^i, c_0^i, c_j^i), \\ & \quad j = 1, \ldots, n_e \end{aligned} \tag{3.4}$$

W konsekwencji powyższej analizy system został zbudowany z pojedynczego procesu MP koordynującego działanie efektorów, tylu procesów ECP ile jest efektorów oraz tylu procesów VSP ile jest czujników wirtualnych. Wszystkie procesy muszą pracować współbieżnie. Zakłada się, że wszystkie procesy komunikują się wykorzystując mechanizm *spotkań* z jednoczesnym przesłaniem wiadomości. Jest to typowy mechanizm synchronizacji i komunikacji zadań (procesów) w rozproszonych systemach wieloprocesorowych. Takim systemem rozproszonym może być sieć lokalna komputerów klasy PC pracujących pod kontrolą wielozadniowego systemu operacyjnego czasu rzeczywistego QNX [22, 38, 39, 101]. Konstruowany sterownik jest aplikacją systemu QNX.



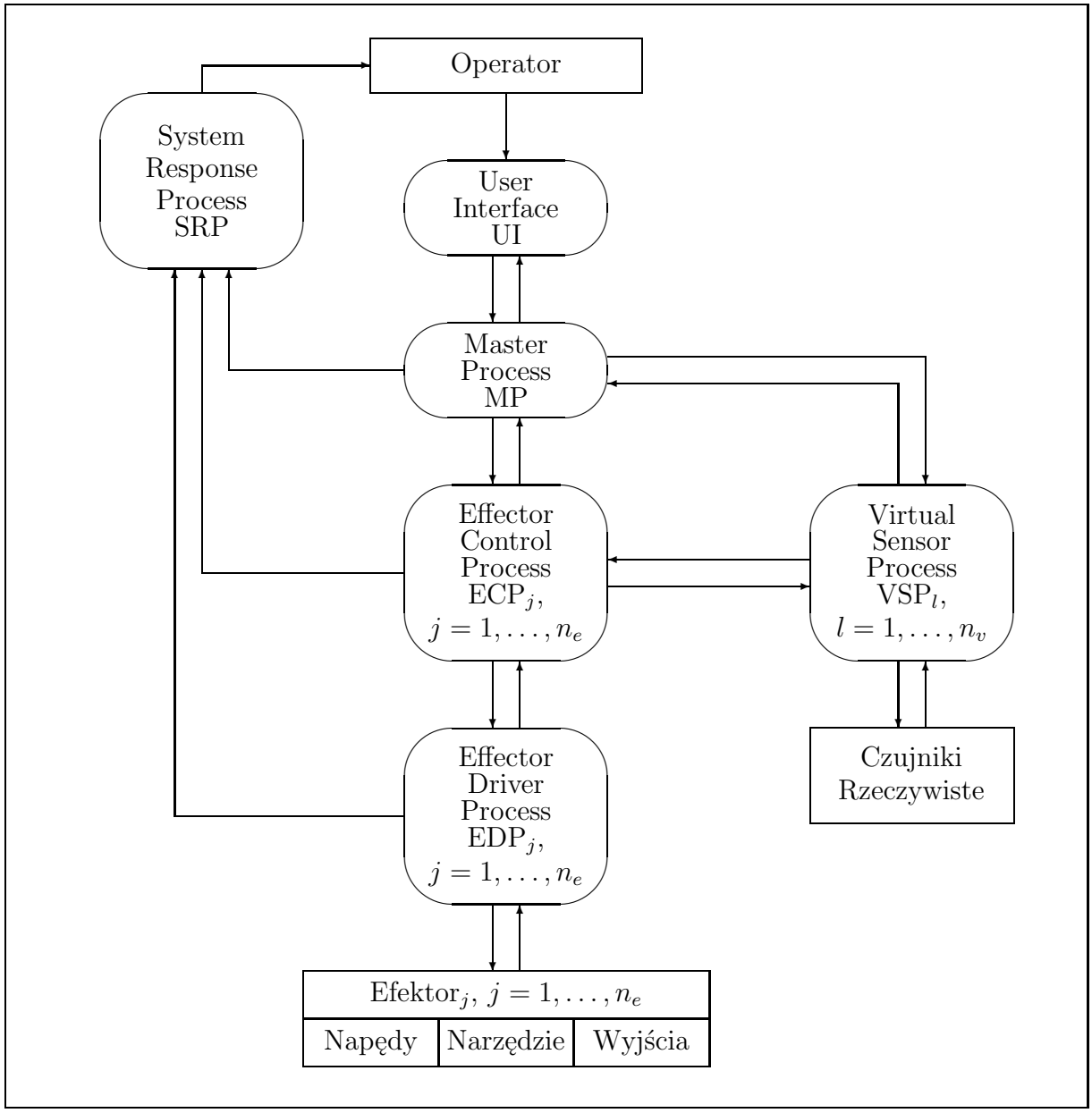
Rys. 3.1: Struktura procesów sterujących efektorami

Bardziej elegancką strukturę systemu uzyskuje się dzieląc każdy z procesów ECP na dwa podprocesy. Pierwszy, to właściwy proces ECP, odpowiedzialny za realizację zadania przez dany efektor, a drugi to proces EDP (*Effector Driver Process*) odpowiedzialny za rozwiązanie prostego i odwrotnego zagadnienia kinematyki oraz bezpośredni dostęp do

sprzętu, a więc za funkcje charakterystyczne dla danego typu robota, a nie realizowanego zadania. W konsekwencji uzyskano strukturę jak na rysunku 3.1.

3.2. Implementacja przyjętej struktury

Zgodnie z proponowaną koncepcją tworzenia układów sterujących, sterownik ma hierarchiczną strukturę funkcjonalną odziedziczoną po systemie `MRROC` [47, 48, 80, 81]. Poszczególne funkcje 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 sterownika przedstawiona została na rys. 3.2. Uzupełniono ją o kontakt z operatorem systemu.



Rys. 3.2: Struktura funkcjonalna sterownika

Przyjęto, że konstruowany sterownik będzie mógł sterować systemem wielorobotowym z nieznaną *a priori* liczbą efektorów (robotów i innych urządzeń). Niemniej jednak skoncentrowano się na realizacji oprogramowania dla robota IRp-6. Ponadto nie poczyniono żadnych założeń co do liczby oraz rodzaju receptorów (czujników sprzętowych) dołączonych do systemu, ponieważ wstępnie nie można założyć do wykonywania jakich zadań ten system będzie wykorzystany. Przy powyższych założeniach, struktura sterownika systemu musi być otwarta i łatwo modyfikowalna.

Sterowniki tworzone są z modułów w postaci funkcji, obiektów i procesów napisanych w języku `C++`. Aby uprościć sposób tworzenia tego typu sterowników, została napisana biblioteka gotowych do użycia modułów: procesów i obiektów. Wybierając dowolne elementy z biblioteki, a w przypadku braku odpowiednich elementów, modyfikując istniejące lub tworząc całkowicie nowe funkcje, obiekty i procesy można skonstruować sterownik dedykowany dokładnie danemu zadaniu. A zatem, ze strony sterownika praktycznie nie ma ograniczeń na rodzaj zadania, które może realizować system robotyczny.

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 [39, 101]. 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.

Z punktu widzenia implementacji układów sterowania robotami istotne są następujące cechy systemu QNX:

- przewidywalna reakcja (w określonym z góry czasie) na pojawiające się zdarzenia – *system czasu rzeczywistego*
- organizacja pracy sieci komputerowej i dostarczenie jednolitych mechanizmów do synchronizacji i komunikacji zadań w obrębie całej sieci – *system rozproszony*
- współbieżnie wykonywanie się wielu zadań: na kilku procesorach lub w podziale czasu na jednym procesorze – *system wielozadaniowy.*

Sterowniki tworzone są z modułów w postaci funkcji i procesów napisanych w języku `C++`. Aby uprościć sposób tworzenia tego typu sterowników, została napisana biblioteka gotowych do użycia modułów: procesów i obiektów. Wybierając dowolne elementy z biblioteki, a w przypadku braku odpowiednich elementów, modyfikując istniejące lub tworząc całkowicie nowe funkcje i procesy można skonstruować sterownik dedykowany dokładnie danemu zadaniu. A zatem, ze strony sterownika praktycznie nie ma ograniczeń na rodzaj zadania, które może realizować system robotyczny.

W przypadku systemów złożonych, a takim jest system wielorobotowy, układ sterowania ma często wielowarstwową strukturę hierarchiczną typu *master/slave*. Warstwa nadrzędna – koordynator (*master*) – realizuje cel globalny systemu jako całości, warstwa niższa (*slave*) realizuje sterowanie poszczególnymi efektorami. Mamy więc wyraźny podział funkcji, przez co struktura sterownika jest przejrzysta. To zaś z kolei ułatwia diagnostykę potencjalnych błędów i znacznie upraszcza modyfikację oraz rozbudowę sterownika. Cecha ta jest szczególnie cenna przy proponowanej koncepcji tworzenia sterowników, kiedy

każda zmiana zadania powoduje rekonstrukcję sterownika. Poszczególne funkcje 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.

Można wyróżnić kilka warstw w układzie sterowania (rys. 3.3). 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. Warstwa druga i trzecia wspólnie tworzą część zależną od zadania, które ma być zrealizowane. Dodatkowo istnieje jeszcze warstwa nie związana z bezpośrednio z realizacją zadania – procesy UI (ang. *User Interface*) i SRP (ang. *System Response Process*) zapewniające komunikację sterownika z operatorem systemu.

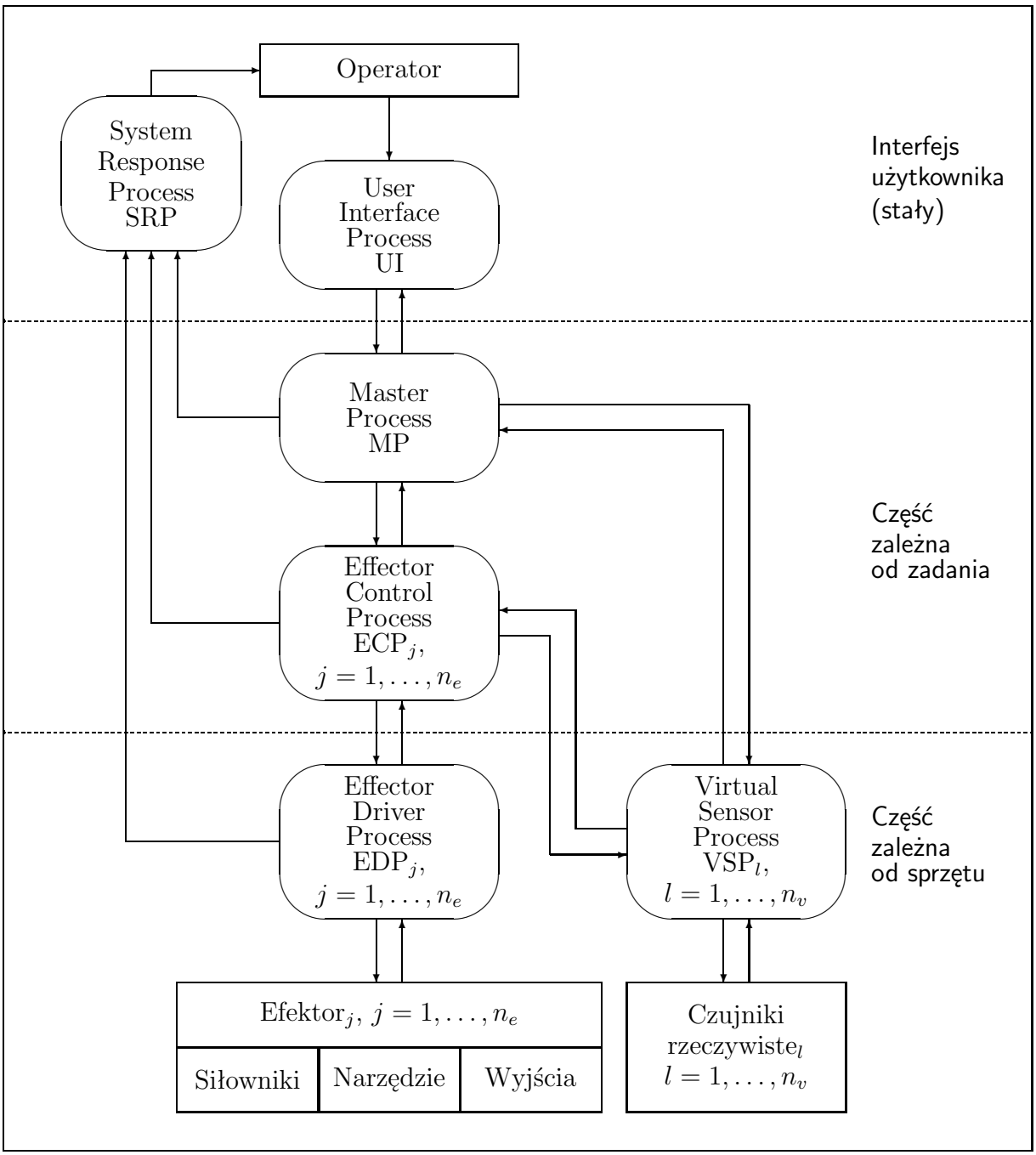
Warstwa komunikacji użytkownika z rozproszonym sterownikiem wielorobotowym składa się z dwóch modułów: procesu *User Interface* obsługującego zlecenia operatora oraz procesu *System Response Process* wyświetlającego komunikaty o stanie systemu. Oba procesy realizują komunikację z użytkownikiem za pośrednictwem 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 (np. nie można ręcznie sterować robotem w trakcie działania procesu MP).

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.

Zlecenia wstrzymania, wznawiania i zakończenia działania procesu MP są realizowane poprzez wzajemną, programową wymianę informacji między procesami UI i MP. Rozwiązanie to pozwala na ścisłą kontrolę miejsca wstrzymania wykonywania procesu MP. Realizacja tych zleceń następuje nie natychmiastowo, lecz po dokończeniu wykonania przez proces MP każdej instrukcji ruchowej. Wstrzymywanie wykonania procesu MP możliwe jest też na poziomie systemu operacyjnego. Operacja ta przewidziana jest do użycia w sytuacjach awaryjnych, gdy trzeba natychmiast wstrzymać wykonanie procesu sterującego. Wykorzystuje się tu *sygnały* dostępne w systemie operacyjnym QNX. Zasadnicza różnica w stosunku do programowej wymiany komunikatów polega na natychmiastowym wstrzymaniu wykonania procesu MP bez „jego wiedzy" (nie jest kończone wykonanie instrukcji ruchu).

Proces SRP odbiera komunikaty od wszystkich procesów, które informują użytkownika o zaistnieniu szczególnej sytuacji (zmiana stanu systemu, informacja o błędzie), formatuje



Rys. 3.3: Warstwy sterownika `MRROC++`

i wyświetla komunikaty na ekranie monitora, przechowuje w buforze nadchodzące komunikaty, tak aby można było śledzić zachowanie systemu od momentu jego uruchomienia.

Proces MP składa się z dwu podstawowych części: *stałej powłoki* (niemodyfikowalnej części procesu) oraz *części modyfikowanej* przez użytkownika. W części stałej procesu dokonuje się: rejestracja procesu MP w systemie operacyjnym, powoływanie procesów potomnych oraz inicjacja kanałów komunikacyjnych między odpowiednimi procesami. Struktura oraz funkcje części modyfikowanej procesu MP, 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 części zmiennej procesu MP jak i procesów ECP$_j$.

Biorąc pod uwagę sposób współpracy robotów, wyróżnia się trzy grupy zadań [49, 50, 84]:

- 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 inicjalizowane łącza komunikacyjne między procesami, proces MP zawieszony jest w oczekiwaniu na zlecenie operatora. Procesy ECP_j ($j = 1, \ldots, n_e$, gdzie n_e jest liczbą efektorów), autonomicznie sterują ruchem poszczególnych robotów. W tym przypadku instrukcje `Move` i `Wait` będą właśnie umieszczane w wymiennych częściach procesów ECP_j. Instrukcje te dotyczą pojedynczego efektora związanego z danym procesem ECP_j.

W przypadku **luźnej współpracy robotów** proces MP pełni funkcje takie jak uprzednio, a ponadto synchronizuje procesy ECP_j. W algorytmie realizacji zadania, w części wykonywanej przez proces MP, zapisuje się, które procesy ECP_j mają być synchronizowane i w których chwilach. Zgłoszenie żądania synchronizacji jest zapamiętywane i zgłaszające się procesy ECP_j są zawieszane aż do chwili zgłoszenia się wszystkich synchronizowanych procesów 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 ECP_j. W tym przypadku również, instrukcje `Move` i `Wait` będą umieszczane w wymiennych częściach procesów ECP_j. Tutaj również instrukcje te dotyczą pojedynczego efektora związanego z danym procesem ECP_j.

W zadaniach wymagających **ścisłej współpracy robotów** rola procesu MP jest dominująca. Proces MP wykonuje całość programu sterującego wszystkimi efektorami. Użytkownik nie pisze własnych procesów ECP_j, które są tylko pośrednikami między procesem MP, a opisanymi dalej procesami EDP. W tym przypadku instrukcje `Move` i `Wait` umieszczane są w

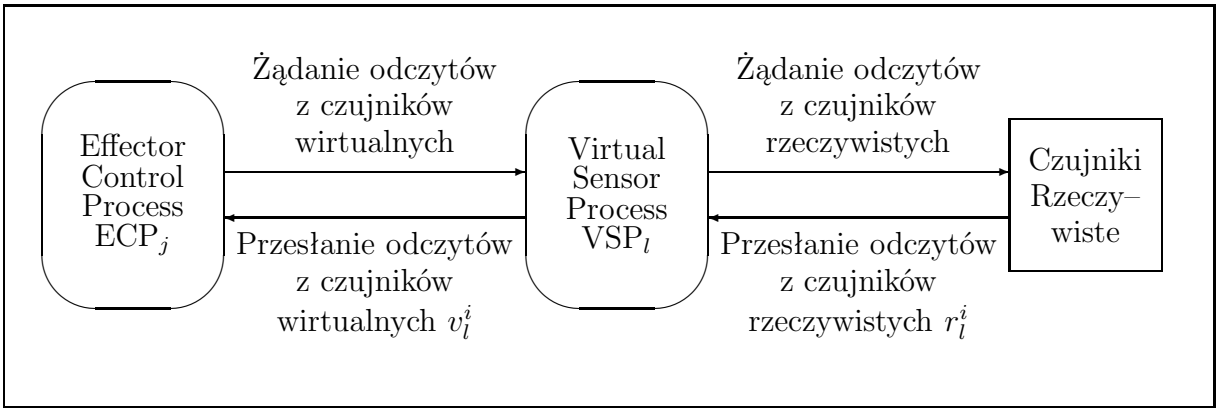
Procesy sterowania efektorami ECP_j są tworzone przez użytkownika stosownie do potrzeb zadania. Podobnie jak proces MP, procesy ECP składają się z części niemodyfikowalnej oraz części pisanej przez użytkownika. Algorytm realizacji danego zadania implikuje strukturę, funkcje oraz liczbę procesów ECP_j. 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 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 ECP_j w ściśle określonych miejscach programu sterującego. Procesy 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 sterującym, w których konieczna jest synchronizacja poszczególnych ECP_j, możemy realizować fizyczną koordynację działania efektorów.

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

Proces EDP jest odpowiedzialny za zarządzanie pracą pojedynczego efektora (robota). Po uruchomieniu, proces ten oczekuje na zlecenia od innych procesów (klientów) systemu. Proces EDP spełnia rolę interpretera poleceń przysyłanych do niego przez proces ECP. Sam proces EDP ma strukturę złożoną i składa się z wielu podprocesów. Prace nad sterownikiem robota IRp-6 głównie koncentrowały się nad zaprojektowaniem i uruchomieniem tego procesu.



Rys. 3.4: Interaktywna metoda komunikacji

Komunikacja procesu ECP_j z procesami VSP_l, których on używa, może być zrealizowana jako przekaz:

- interaktywny (rys. 3.4) lub
- nieinteraktywny (rys. 3.5).

W przypadku komunikacji interaktywnej ECP_j wysyła żądania danych do swoich VSP_l. Procesy VSP_l odczytują adekwatne czujniki rzeczywiste, dokonują agregacji danych i przesyłają uzyskany w ten sposób odczyt czujnika wirtualnego z powrotem do ECP_j (rys. 3.4). W międzyczasie ECP_j może sterować efektorem. Jeżeli odczyty nie nadejdą do chwili gdy ECP_j wykona operację odczytu danych, jest on zawieszany do chwili ich nadejścia.

W przypadku komunikacji nieinteraktywnej VSP_l wyzwalany jest przerwaniami (rys. 3.5). Gdy czasomierz wygeneruje przerwanie, VSP_l odczytuje adekwatne czujniki rzeczywiste, dokonuje agregacji danych i otrzymany wynik wstawia do bufora. Następnie proces VSP_l jest zawieszany. W ten sposób każdy z procesów ECP_j może odczytać ostatnie dane w każdej chwili. Oczywiście dostęp procesów do bufora musi być synchronizowany (wzajemne wykluczanie).

3.3. Sposób konstruowania sterownika dedykowanego zadaniu użytkownika

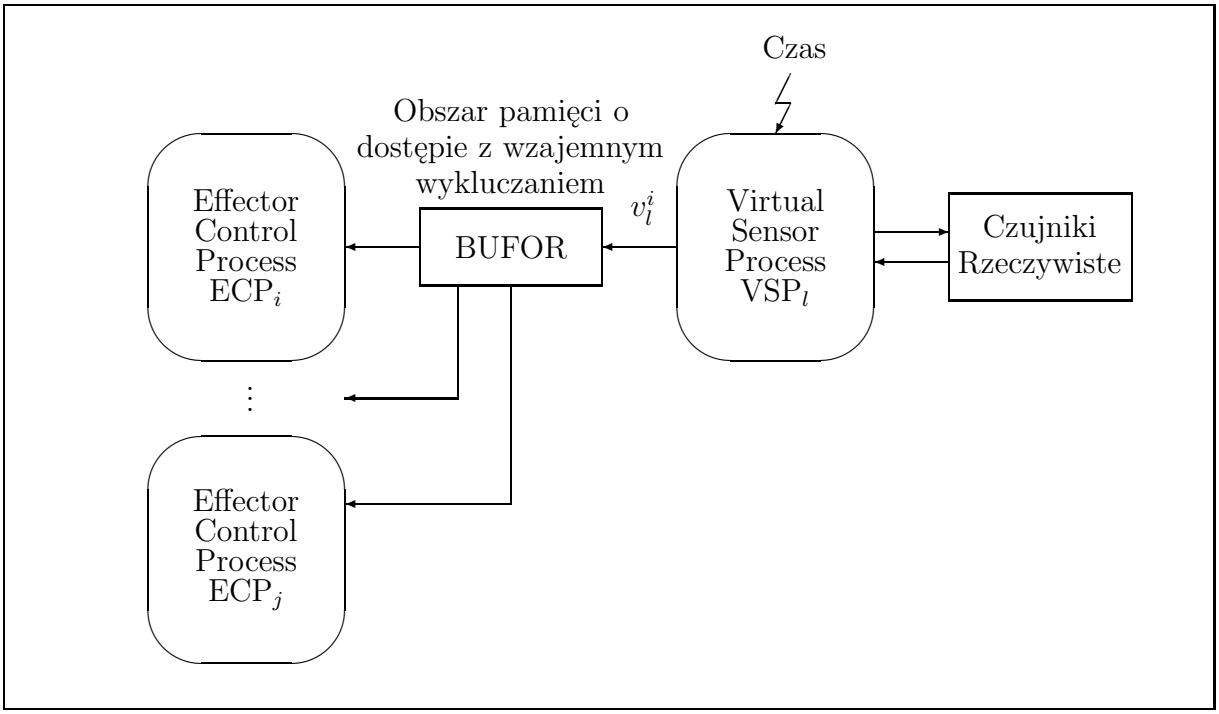
Wysiłek programisty, przy konstruowaniu programu dla systemu robotycznego, koncentruje się na kilku szczegółach. Po pierwsze, programista musi podać listy czujników kon-

kretnych oraz robotów, które wezmą udział w kolejnym ruchu. Po drugie, jeżeli kolejną instrukcją jest polecenie `Wait`, to musi dostarczyć warunek początkowy, a więc `condition`. Jeżeli natomiast kolejną instrukcją jest `Move`, to musi dostarczyć generator trajektorii zadanej, który prócz funkcji przejścia musi zawierać warunek końcowy. Po trzecie, programista musi określić sekwencję instrukcji `Wait`, `Move` oraz instrukcji pomocniczych języka `C++` (n.p. konstruujących adekwatne listy), które będą stanowiły wymienną część procesu MP, a jednocześnie program sterujący całością systemu. Jak widać, każdy krok budowy programu jest precyzyjnie określony i dotyczy niewielkiego i dobrze zdefiniowanego fragmentu kodu programu sterującego, co powinno wydatnie ułatwić programowanie i drastycznie zmniejszyć możliwość wprowadzenia błędów do sterownika, a więc skrócić czas jego implementacji i testowania.

Programowanie systemu wielorobotowego zawierającego robot IRp-6 sprowadza się do zapisania fragmentów procesów ECP oraz MP zależnych od treści zadania, które roboty mają wykonać. Każdy z procesów ECP i MP, prócz części zależnej od wykonywanego zadania, ma część niezmienną, umożliwiającą kontakt z resztą systemu, tzn. procesami UI oraz SRP.

Część zmienna składa się z wywołań procedur `Move` oraz `Wait` z adekwatnymi argumentami. Oczywiście wpierw te argumenty, tzn. obiekty konkretne, muszą być stworzone przez programistę. Użytkownik odpowiedzialny jest za stworzenie odpowiednich list czujników, robotów oraz kolejnych generatorów. Musi również zagwarantować zgodność pomiędzy listami robotów oraz czujników, a sposobem tworzenia trajektorii przez odpowiednie generatory konkretne.

Istotą programowania, a właściwie tworzenia sterownika dedykowanego zadaniu, jest napisanie pewnych rozdzielnych modułów programowych, które poprzez procedury `Move` oraz `Wait` zostaną złożone w całość, tak aby zostało zrealizowane zadanie. Programista każdorazowo koncentruje się na małym wycinku kodu, np.: pojedynczym czujniku, pojedynczym robocie, a następnie z tych elementów wytwarza odpowiednie listy według



Rys. 3.5: Nieinteraktywna metoda komunikacji

załączonego wzorca – klas abstrakcyjnych. Najistotniejszą, a być może również i najtrudniejszą koncepcyjnie, sprawą jest wytworzenie szeregu generatorów ruchu dla kolejnych wywołań procedury `Move`, ale i tu istnieją wzorce, do których należy się dostosować.

Rozdział 4

Proces MP

4.1. Rola procesu MP

Podstawowymi zadaniami Master Process są:

- koordynacja pracy całego systemu (tzn. synchronizacja działania procesów ECP w czasie),
- kontakt z operatorem systemu (nasłuch jego poleceń oraz ich realizacja),
- generacja trajektorii dla ściśle współpracujących robotów,
- ewentualny kontakt z procesami VSP,
- przesylanie do procesu SRP informacji o swoim stanie oraz o ewentualnych błędach i awariach powstałych w procesie MP lub procesach, z ktorymi ma on nawiązaną komunikację.

4.2. Struktura procesu MP

Master Process składa się z dwóch zasadniczych części (rys. 4.1):

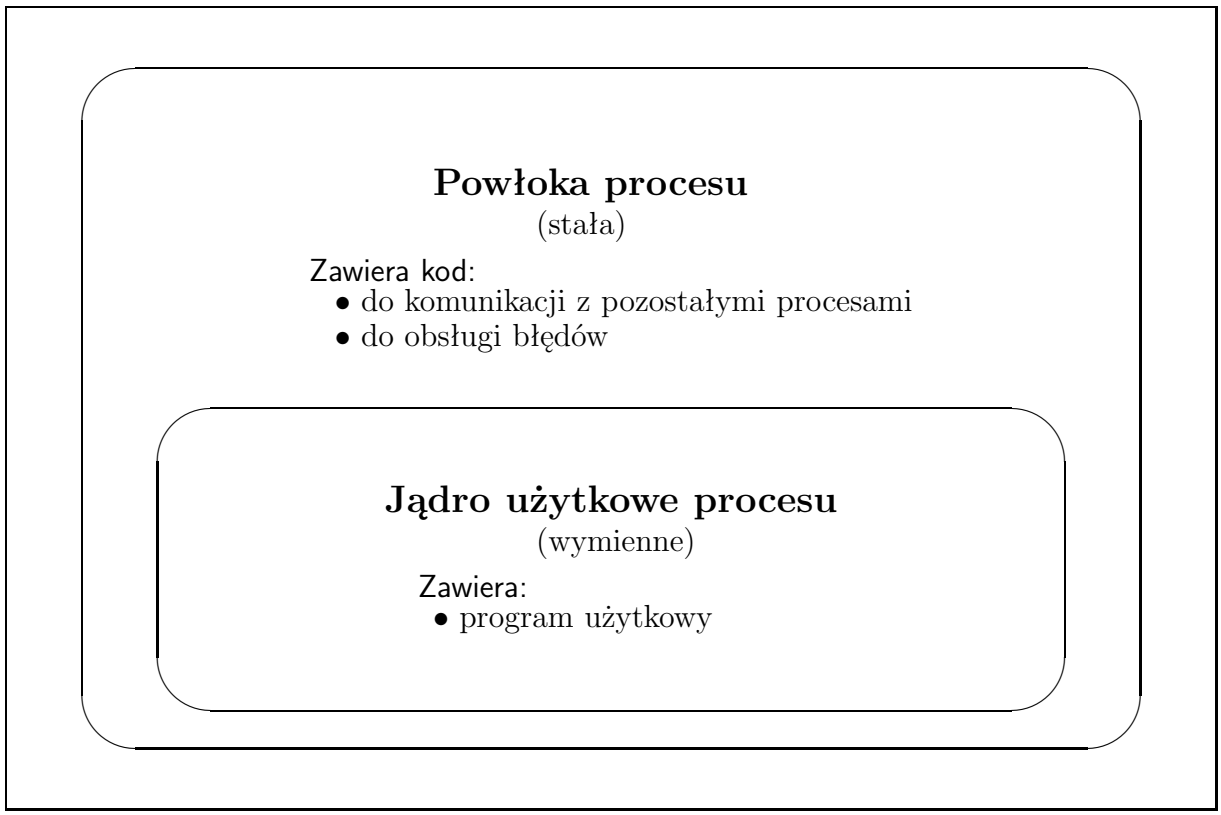
- powłoki (niezmiennej części procesu – niezależnej od wykonywanego zadania),
- jądra (wymiennej część procesu – zależnej od programu użytkowego realizującego zadanie zlecone systemowi przez użytkownika).

4.2.1. Część stała procesu – powłoka

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)



Rys. 4.1: Struktura procesów ECP i MP.

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.

4.2.2. Część wymienna procesu – jądro

Struktura jądra procesu MP przedstawiona jest na rys. 4.2. Jądro procesu MP zawiera:

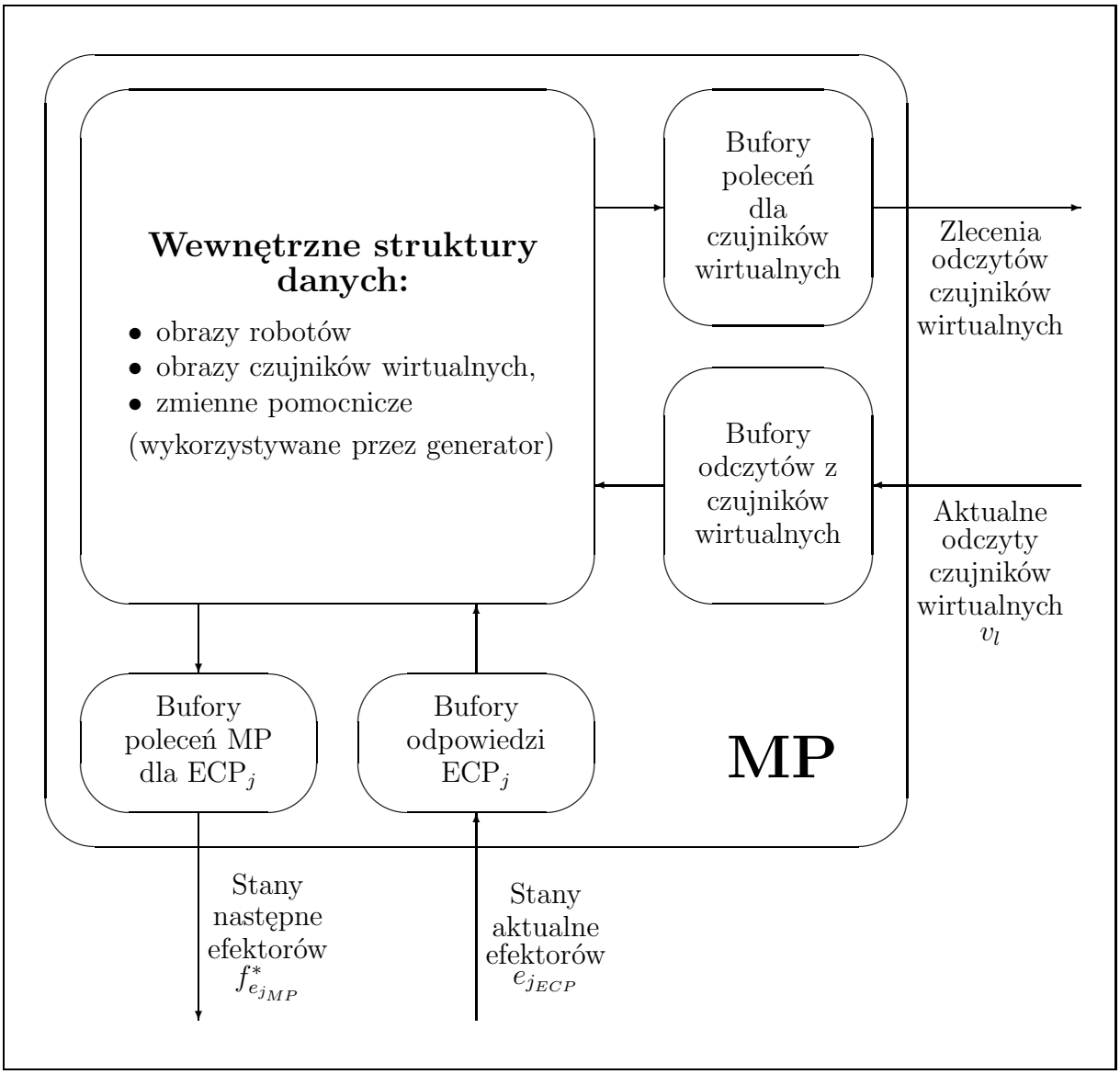
- deklaracje obiektów, które będą używane w programie użytkowym,
- program użytkowy.

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. 4.2 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`.



Rys. 4.2: Wewnętrzna struktura jądra procesu MP.

4.2.3. 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.

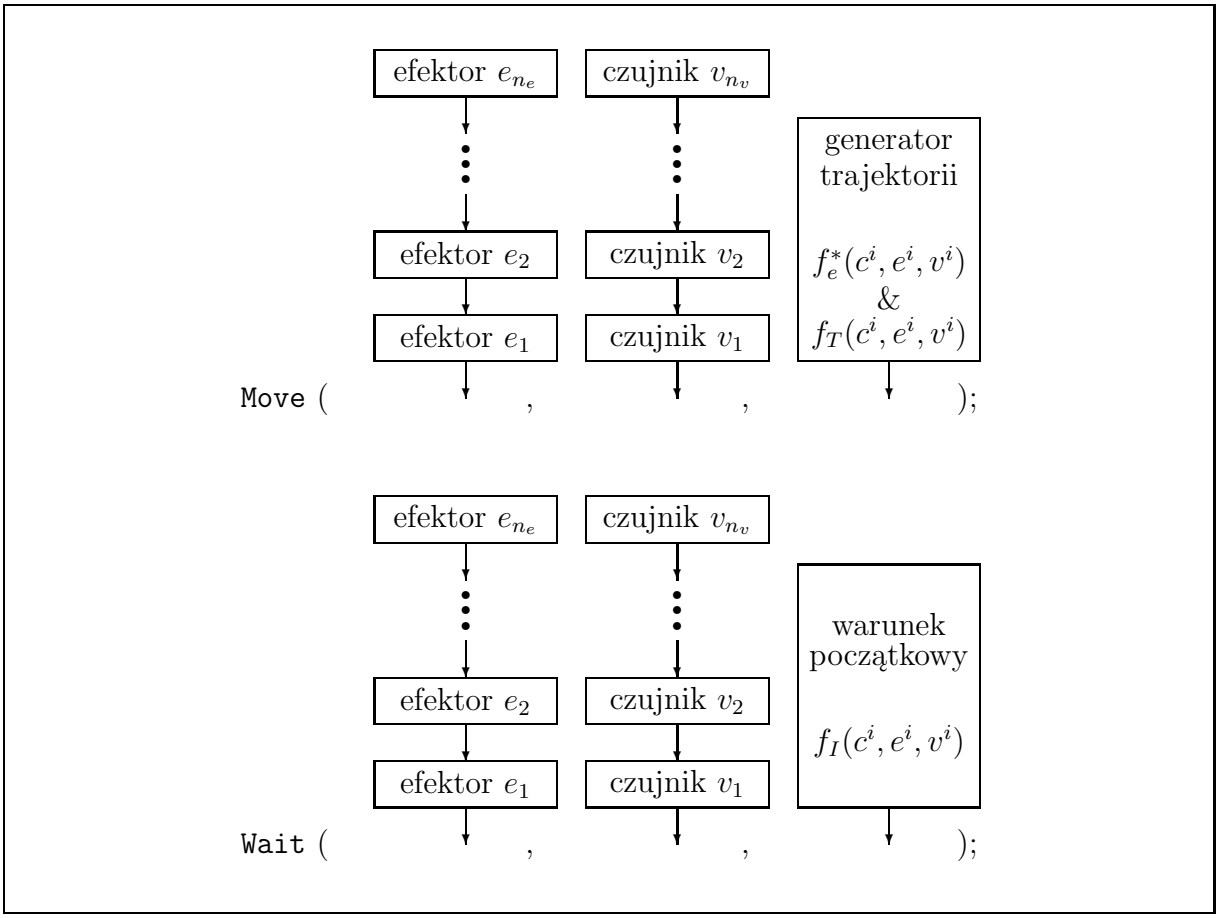
Najczęściej program użytkowy skonstruowany jest jako pętla (`for`), w której umieszczone są instrukcje (funkcje) `Move` i `Wait` oraz dodatkowe instrukcje języka `C++`. Rozpoczyna on swe działanie od wysłania zlecenia wszczęcia wykonania do wszystkich podległych mu procesów ECP (`start_all`).

4.3. Uzasadnienie przyjętej postaci instrukcji `Move` i `Wait`

W przypadku pisania programu działań robotów programista musi określić jakie ruchy i kiedy ma wykonać każdy z robotów. Aby to zrobić, musi zdefiniować trajektorie ruchu narzędzi przytwierdzonych do manipulatorów oraz stwierdzić czy i jakie czujniki będą niezbędne do jej modyfikacji w trakcie realizacji ruchu. Z ogólnego zapisu semantyki instrukcji ruchu `Move` (zależność (**??**) i rys. 2.4) oraz `Wait` (zależność (**??**) i rys. 2.3) wynika, że dla każdego z ruchów niezbędne jest określenie jakie czujniki mają być użyte do monitorowania oraz jaką konkretną postać powinny przyjąć warunki: wstępny $f_I(c^i, e^i, v^i)$ i końcowy $f_T(c^i, e^i, v^i)$, a ponadto w jaki sposób ma być generowana trajektoria ruchu narzędzia – $f_e^*(c^i, e^i, v^i)$.

Ponieważ w funkcjach: $f_I(c^i, e^i, v^i)$ (**??**) oraz $f_T(c^i, e^i, v^i)$ i $f_e^*(c^i, e^i, v^i)$ (**??**) argumentami są wektory e^i i v^i o zmiennej i *a priori* nieznanej liczbie elementów, to implementując instrukcje `Move` i `Wait` jako procedury języka `C++`, albo należy użyć funkcji o zmiennej liczbie argumentów (tak jak na przykład dla instrukcji `printf`) – co nie jest wygodne, ponieważ musiałby istnieć argument określający jak interpretować pozostałe argumenty, albo zdecydować się na stałą liczbę argumentów, ale wtedy argumentami tymi muszą być listy lub tablice dynamiczne wyżej wzmiankowanych elementów. Implementując proces MP zdecydowano się na użycie list. Użytkownik musi stworzyć zarówno listę efektorów (chwilowo przyjęto, że będą to roboty) jak i listę czujników, które to listy będą wykorzystane przez daną instrukcję `Move` bądź `Wait`. Zadaniem użytkownika jest stworzenie programu dla systemu jako sekwencji instrukcji `Move` i `Wait`, których argumentami będą listy: robotów, które należy w danym ruchu przemieścić, oraz czujników wirtualnych, które w trakcie ruchu będą dostarczały danych o środowisku.

Każdy z czujników wirtualnych dostarcza danych w innej postaci. Ponadto dane te mogą być różnie wykorzystane do modyfikacji trajektorii w zależności od potrzeb zadania, a więc w sposób zależny od programisty (różna postać funkcji $f_e^*(c^i, e^i, v^i)$). Powyższy dylemat postanowiono rozwiązać dwuetapowo. Wpierw, jak już wspomniano, określa się: czujniki, które mają być wykorzystane w realizacji ruchu, oraz listę robotów, które uczestniczą w ruchu, a następnie: generator trajektorii dla danego ruchu oraz warunki: początkowy i końcowy. Tak więc, listy robotów i czujników stanowią argumenty instrukcji `Move`, a ponadto argumentami muszą być takie pojęcia abstrakcyjne jak generator trajektorii, który generuje kolejne położenia zadane dla robotów na podstawie funkcji $f_e^*(c^i, e^i, v^i)$ oraz sprawdza warunek końcowy $f_T(c^i, e^i, v^i)$. Dla instrukcji `Wait` argumentami są: lista czujników oraz warunek początkowy $f_I(c^i, e^i, v^i)$. W ten sposób otrzymano **instrukcje ruchu**, czyli procedury `Move` i `Wait`, o stałej strukturze, niezależnej zarówno od liczby i typu czujników jak i od sposobu generacji kolejnych położeń zadanych dla robotów przez generator trajektorii oraz postaci warunków początkowych i końcowych. Aby to było możliwe do zrealizowania technicznego, przy implementacji takich procedur trzeba skorzystać z mechanizmów dziedziczenia i polimorfizmu będących elementami programowania obiektowego. Należy zwrócić uwagę, że rozwiązanie to umożliwia wyodrębnienie, w postaci adekwatnych parametrów, tych niewielu elementów, które zmieniają się w każdym ruchu (tzn.: $f_I(c^i, e^i, v^i)$ (**??**) oraz $f_T(c^i, e^i, v^i)$ i $f_e^*(c^i, e^i, v^i)$ (**??**)), a występują w zapisie semantyki instrukcji `Move` i `Wait`. Dzięki temu niezmienne części mogą być zakodowane jednokrotnie jako ciała procedur realizujących te instrukcje. W szczególności struktury algorytmów realizujących instrukcje ruchowe są niezmienne, co widać z rysunków: 2.3 i 2.4. Postacie instrukcji `Wait` oraz `Move` przedstawiono na rys. 4.3.



Rys. 4.3: Instrukcje ruchu systemu MRROC++. Efektorem najczęściej jest robot.

4.4. 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 prefiksem `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` posiada 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 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 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
```

4.5. Czujniki

Abstrakcyjna klasa bazowa dla czujników ma następującą postać:

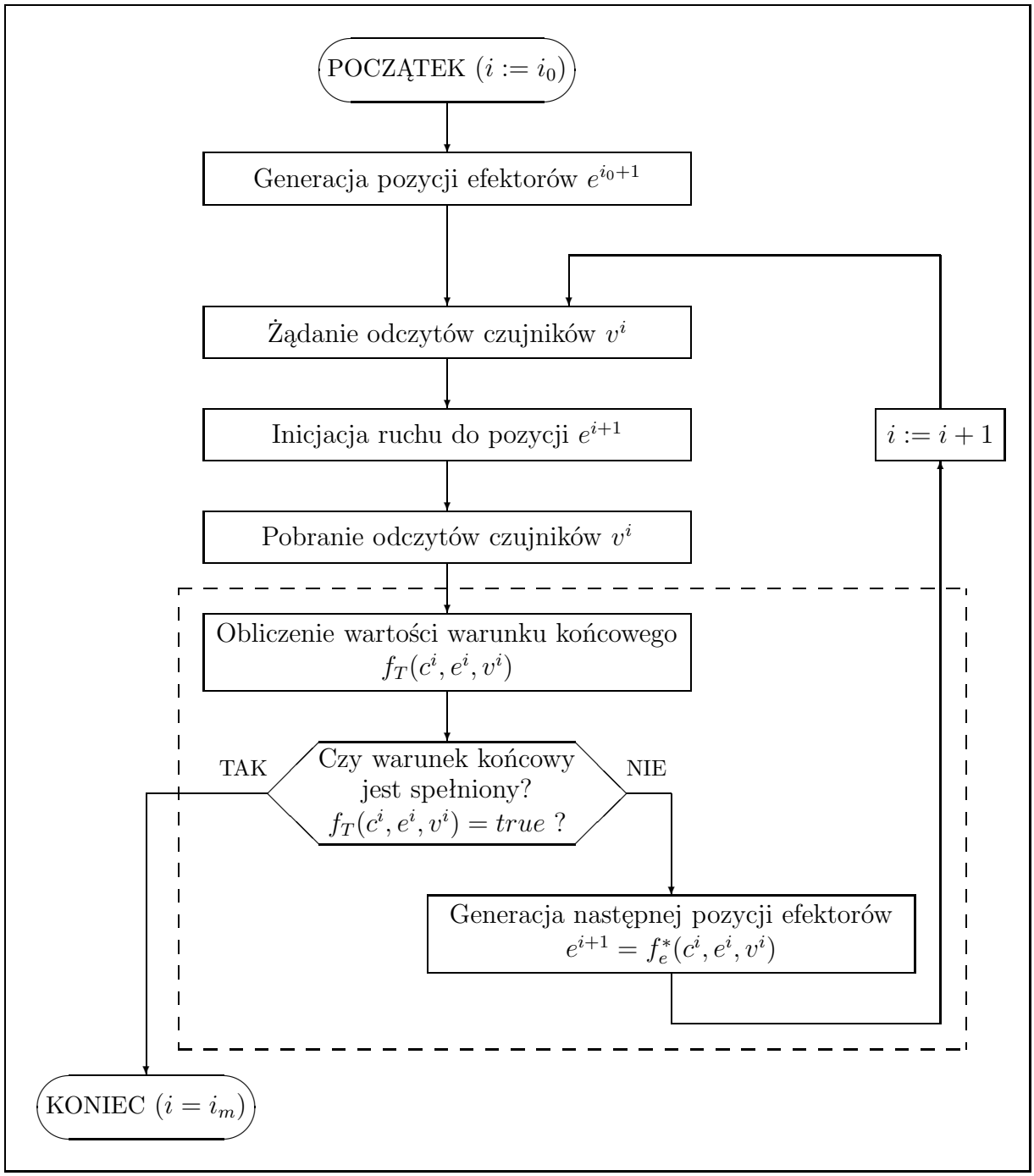
```
class sensor {
   // Klasa bazowa dla czujników (klasa abstrakcyjna)
   // Czujniki konkretne wyprowadzane są z klasy bazowej
  public:

    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
```

4.6. 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:
  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
```



Rys. 4.4: Sieć działań instrukcji `Move` z zaznaczonym zakresem działania metody `generator.next_step`

Ponieważ w sieci działań pokazanej na rys. 4.4 można wyróżnić zwartą część, zaznaczoną prostokątem narysowanym linią przerywaną, odpowiedzialną za obliczenie warunku końcowego $f_T(c^i, e^i, v^i)$, sprawdzenie jego wartości oraz obliczenie następnej wartości zadanej na podstawie funkcji przejścia $f_e^*(c^i, e^i, v^i)$, to odpowiadający jej kod programu można umieścić w jednej procedurze (tzn. metodzie obiektu `generator`). Tak też zrobiono tworząc metodę `next_step`. Jednocześnie stworzono metodę `first_step` odpowiadającą za operacje wykonywane przez pierwszy blok sieci działań z rys. 4.4.

Dopiero obiekty klas pochodnych (generatory konkretne) mogą być użyte 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.

Generatory na poziomie MP można sklasyfikować w zależności od typu argumentów, które są używane przez funkcję generującą trajektorię, czyli $f^*_{e_{MP}}$. Wyróżniamy pięć typów generatorów: $f^{**}_{e_{MP}}$ oraz $f^{*k}_{e_{MP}}, k = 0, \ldots, 3$. Potencjalnymi argumentami tych funkcji są: c_0, e, v. Aby wykonać dowolne obliczenia potrzeba zaangażować zasoby pamięciowe MP, a więc c_0. Wszakże, gdy efektory (roboty) działają niezależnie żadne obliczenia na poziomie MP nie są potrzebne. Wtedy nie ma również potrzeby, aby na ten poziom dostarczać odczyty czujników v oraz aktualny stan efektorów e. Pobudzanie procesów ECP traktujemy jako przesłanie stałej. W konsekwencji funkcja generująca trajektorię przyjmuje postać: $f^{**}_{e_{MP}}() = const$. Jeżeli jednak efektory współpracują ze sobą w jakiejś formie, to niezbędne jest przeprowadzenie obliczeń na poziomie MP, a więc c_0 musi wystąpić jako argument funkcji $f^{*k}_{e_{MP}}$. W tym przypadku możemy wyróżnić cztery sytuacje w za-

leżności od tego czy występują pozostałe dwa z potencjalnych argumentów, czyli: e lub v. Jeżeli do generacji trajektorii wykorzystywane są odczyty czujników, to wystąpi argument: v. Natomiast, jeżeli do generacji trajektorii wykorzystuje się sprzężenie zwrotne wysokiego poziomu (tzn. poziomu MP) od aktualnego stanu efektorów, to wystąpi argument: e. Ponieważ dopuszczalne są wszystkie kombinacje argumentów e i v, funkcja generująca trajektorię przyjmie jedną z czterech postaci: $f^{*k}_{e_{MP}}(c_0, \bullet, \bullet)$, gdzie $k = 0, \ldots, 3$ oraz symbol $\bullet$ należy rozumieć jako wskazanie, że argument stojący na tym miejscu w liście argumentów funkcji $f^{*k}_{e_{MP}}(c_0, e, v)$ może istnieć lub nie. Numery k przyporządkowano funkcjom $f^{*k}_{e_{MP}}$ wykorzystując odpowiednik dwójkowy dziesiętnej reprezentacji liczby k. W ten sposób dla $k = 0, \ldots, 3$ uzyskuje się kody: 00, 01, 10, 11. W parach (e, v) 0 oznacza brak danego argumentu, natomiast 1 jego występowanie. W ten sposób powstają funkcje: $f^{*0}_{e_{MP}}(c_0)$, $f^{*1}_{e_{MP}}(c_0, v)$, $f^{*2}_{e_{MP}}(c_0, e)$, $f^{*3}_{e_{MP}}(c_0, e, v)$. Powyższe rozważania zostały podsumowane w tabeli 4.1. Znakiem $-$ podkreślono brak danego argumentu.

Kryterium podziału:	**Typ funkcji**	
Działanie robotów	niezależne $f^{**}_{e_{MP}}() = const$	współpraca $f^{*k}_{e_{MP}}(c_0, \bullet, \bullet)$, $k = 0, \ldots, 3$
Wykorzystanie czujników	bez czujników (brak kontaktu z VSP) $f^{*k}_{e_{MP}}(c_0, \bullet, -)$, $k = 0, 2$	z czujnikami (istnieje kontakt z VSP) $f^{*k}_{e_{MP}}(c_0, \bullet, v)$, $k = 1, 3$
Wykorzystanie informacji o aktualnym stanie robota	bez sprzężenia zwrotnego na poziomie MP $f^{*k}_{e_{MP}}(c_0, -, \bullet)$, $k = 0, 1$	ze sprzężeniem zwrotnym na poziomie MP $f^{*k}_{e_{MP}}(c_0, e, \bullet)$, $k = 2, 3$

Tabela 4.1: Klasyfikacja generatorów procesu MP

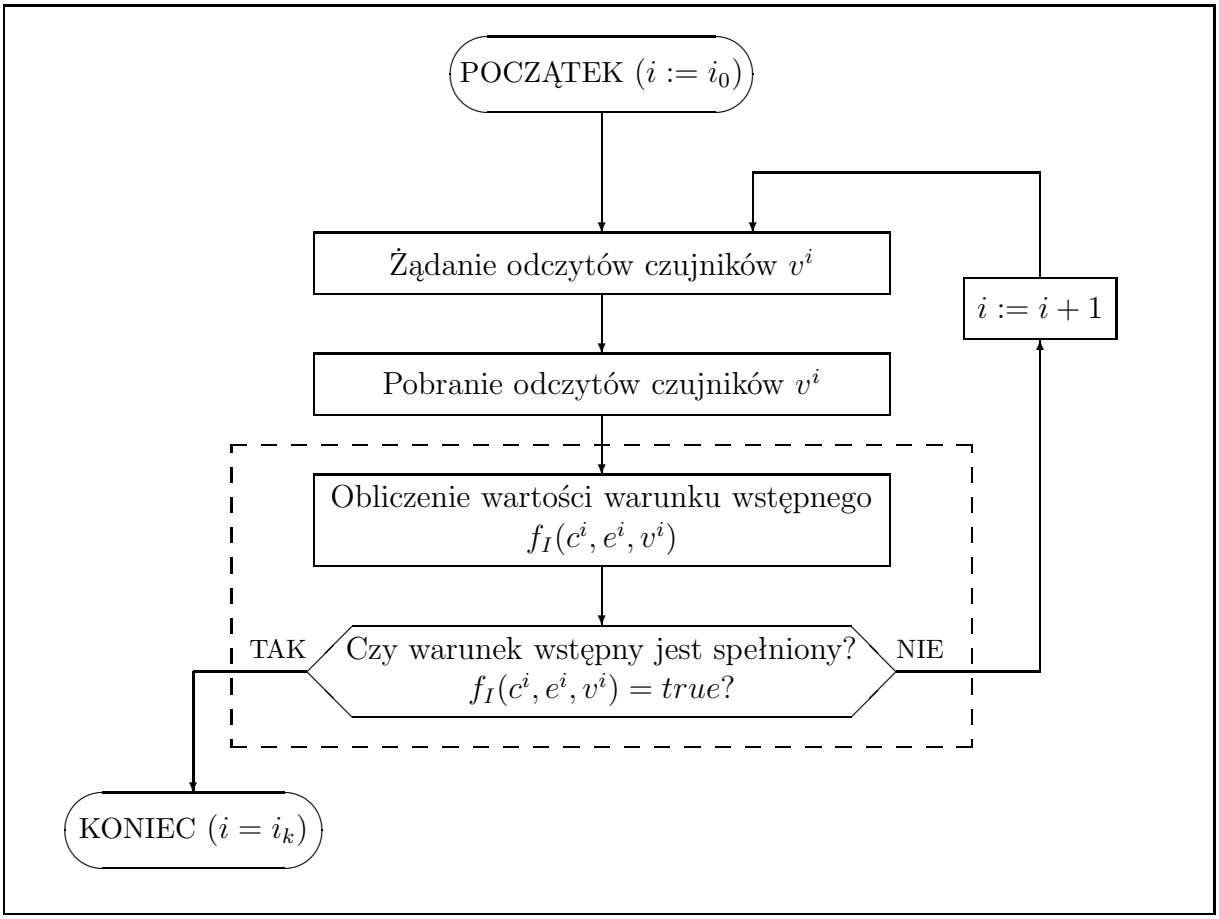
Stworzenie nowego generatora sprowadza się do napisania dwóch jego metod `first_step` oraz `next_step`. Zazwyczaj obliczenia dla pierwszego kroku realizacji trajektorii ruchu różnią się nieco od tych dla każdego następnego, dlatego wprowadzono dwie różne metody. Obie metody mogą zlecać dowolne polecenia 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.

4.7. Warunki wstępne

Monitorowanie warunku wstępnego $f_I(c^i, e^i, v^i)$ (**??**) wykonania instrukcji ruchu (rys. 4.5) przyjęło się, jak już wspominano, rozpatrywać oddzielnie od samej instrukcji przemieszczającej efektory – czyni to instrukcja `Wait`. Klasą obiektów odpowiedzialnych za przechowywanie i obliczanie waryunków wstępnych jest `condition`. Jej metoda odpowiedzialna jest za wykonanie operacji zawartych w prostokącie zaznaczonym linią przerywaną na rys. 4.5, tzn. za obliczenie i sprawdzenie wartości funkcji $f_I(c^i, e^i, v^i)$. 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
```



Rys. 4.5: Sieć działań instrukcji `Wait` z zaznaczonym zakresem działania obiektu `condition`

4.8. Implementacja instrukcji `Move` i `Wait`

Ponieważ procedura (instrukcja) `Move` może być zdefiniowana dla parametrów stanowiących klasy abstrakcyjne, które jedynie w trakcie wykonania muszą być zastąpione przez obiekty klas konkretnych, można podać jej postać już teraz.

```
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 wywolany 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 nie przyszlo polecenie operatora
    if((pid = Creceive(MPProxyPids.stop, NULL, 0 ) ) != -1 ) {
      // Przeslanie 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) {
      // Blad komunikacji miedzyprocesowej - wyjatek
      e = errno;
      perror("Creceive STOP proxy from UI failed ?\n");
      msg->message(MP_SYSTEM_ERROR, e, "MP: Creceive STOP proxy from UI failed");
      throw MP_main_error (MP_SYSTEM_ERROR, (unsigned word32) 0);
    }

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

// Porządki końcowe
return FALSE;
}; // end: Move
```

Ogólna struktura procedury realizującej instrukcję `Wait` przedstawiona jest na rys. 2.3. Powoduje ona oczekiwanie na spełnienie warunku wstępnego. Warunek wstępny określa, czy został osiągnięty stan, w którym może rozpocząć się wykonanie ruchu – innymi słowy, czy spełnione są warunki rozpoczęcia wykonania instrukcji ruchu. Argumentami procedury `Wait` realizującej tę instrukcję są: lista czujników oraz obiekt pochodny `condition` zawierający wyrażenie $f_I(c^i, e^i, v^i)$ określające, czy spełniony został warunek wstępny.

```
BOOLEAN Wait (list<robot>* robot_list, list<sensor>* sensor_list,
              condition&  the_condition ) {
 // Funkcja oczekiwania na spełnienie warunku wstępnego
 // Funkcja zwraca FALSE, gdy samoistny koniec oczekiwania
 // Funkcja zwraca TRUE, gdy koniec 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)

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

    if ( ( pid = Creceive(MPProxyPids.pause, NULL, 0) ) != -1 ) {
      msg->message("To resume MP click RESUME icon");
      if (Receive (MPProxyPids.resume, NULL, 0) == -1) { //Oczekiwanie na RESUME
```

```
        // Wystąpił błąd systemowy przy Receive
        e = errno;
        perror("Creceive RESUME proxy from UI failed ?\n");
        msg->message(MP_SYSTEM_ERROR, e, "MP: Creceive RESUME proxy from UI failed");
        throw MP_main_error (MP_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(MP_SYSTEM_ERROR, e, "MP: Creceive PAUSE proxy from UI failed");
      throw MP_main_error (MP_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();

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

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

4.9. 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`

4.10. 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.

Rozdział 5

Procesy ECP

5.1. Rola procesu ECP

Podstawowymi zadaniami Effector Control Process są:

- realizacja programu użytkowego dla pojedynczego robota,
- kontakt z MP i EDP oraz ewentualnie z procesami VSP i UI w celu realizacji programu użytkowego,
- generacja trajektorii, jeżeli robot znajdujący się pod kontrolą tego procesu ECP działa niezależnie lub luźno współpracuje z innymi,
- przesyłanie do procesu SRP informacji o swoim stanie oraz o ewentualnych błędach i awariach powstałych w procesie ECP lub procesach niższych warstw, z którymi ma on nawiązaną komunikację.

5.2. Struktura procesu ECP

Effector Control Process składa się z dwóch części (rys. 4.1):

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

5.2.1. Część stała procesu – powłoka

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,
- 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, 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.

5.2.2. Część wymienna procesu – jądro użytkowe

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. 5.1. 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. 5.1 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`.

Program użytkowy procesu ECP rozpoczyna swe działanie od powołania do życia 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`. Po zakończeniu fazy inicjacji przechodzi do oczekiwania na polecenie wszczęcia działania (`START_TASK`) od procesu MP.

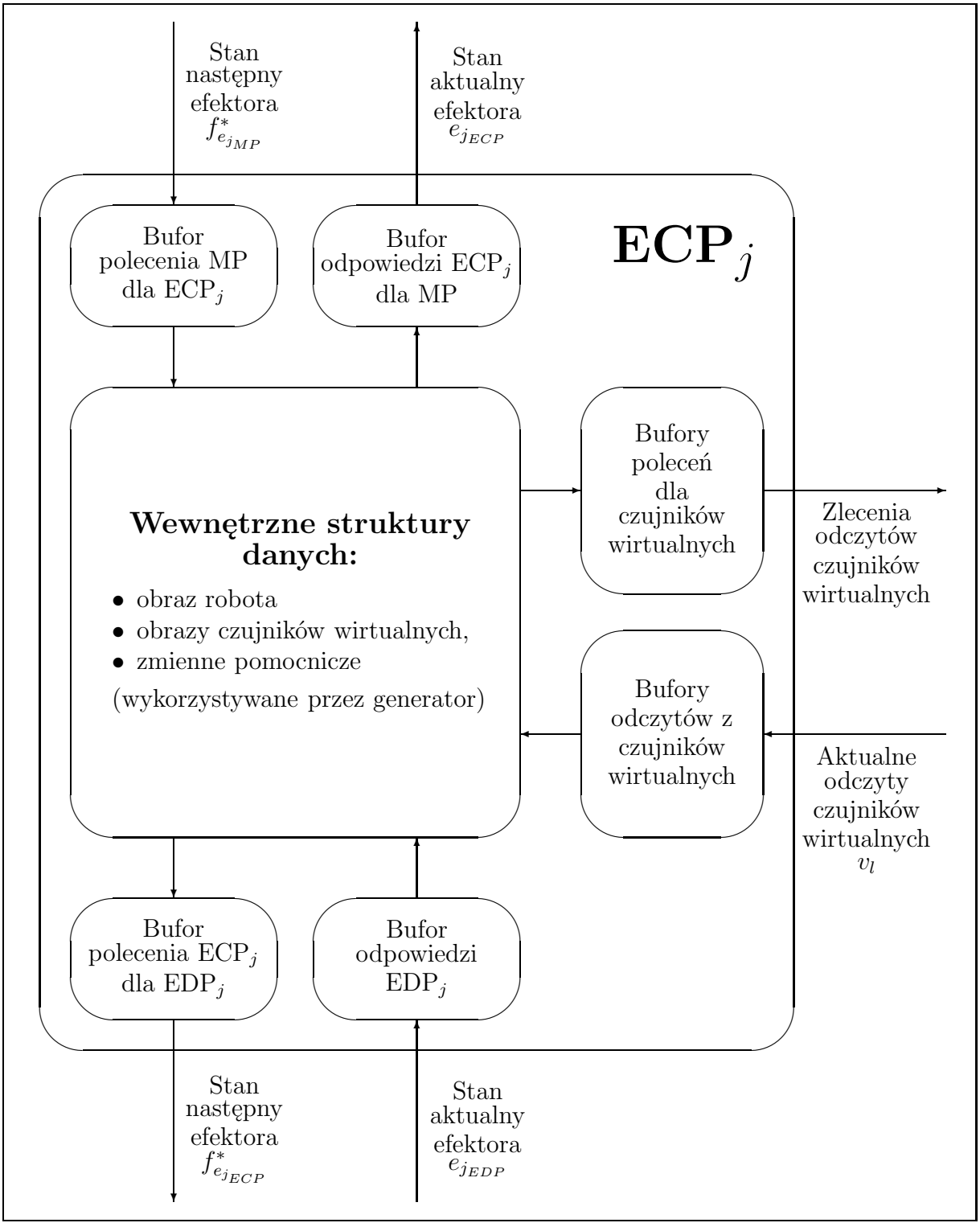
5.2.3. Program użytkowy

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++`.

5.3. 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



Rys. 5.1: Wewnętrzna struktura jądra procesu ECP.

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 odpowie-

dzi 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
```

5.4. 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).

5.5. Generatory

Ostatnim argumentem procedury `Move` jest obiekt stanowiący generator trajektorii. Użytkownik musi wskazać taki generator dla każdego ruchu. Ponieważ generator trajektorii w każdym kroku sterowania może jednocześnie sprawdzić spełnienie warunku końcowego $f_T(c^i, e^i, v^i)$ oraz obliczyć następną wartość zadaną na podstawie funkcji przejścia $f_e^*(c^i, e^i, v^i)$, więc obie te funkcje mogą być zawarte w jednym obiekcie. 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.

Generatory na poziomie ECP można sklasyfikować w zależności od typu argumentów, które są używane przez funkcję generującą trajektorię, czyli $f^*_{e_{j_{ECP}}}$, gdzie $j = 1, \ldots, n_e$. Wyróżniamy osiem typów generatorów: $f^{*k}_{e_{j_{ECP}}}, k = 0, \ldots, 7$. Potencjalnymi argumentami tych funkcji są: c_j, c_0, e, v, gdzie c_j jest stanem j-tego procesu ECP, natomiast c_0 jest stanem procesu MP. Aby wykonać dowolne obliczenia potrzeba zaangażować zasoby pamięciowe ECP$_j$, a więc c_j musi zawsze być argumentem funkcji $f^{*k}_{e_{j_{ECP}}}$. Pozostałe trzy argumenty (tzn. c_0, e, v) mogą występować na liście lub nie. W związku z tym powstaje osiem typów funkcji $f^{*k}_{e_{j_{ECP}}}$. Jeżeli efektor (robot) działa niezależnie od pozostałych, to z poziomu MP przesyłane są jedynie stałe pobudzenia. Konieczne są one do ewentualnego przerwania ruchu przez operatora. Operator informuje, poprzez proces UI, jedynie proces MP o konieczności raptownego przerwania ruchu, a ten z kolei rozsyła adekwatną informację do procesów ECP. Oczywiście informacja ta wykorzystywana jest przez warunek końcowy, a więc przez $f_{T_{j_{ECP}}}$. Tak więc sama funkcja $f^{*k}_{e_{j_{ECP}}}$ nie korzysta w tym przy-

padku z c_0. Gdy efektory ściśle współpracują, to ich wspólna trajektoria obliczana jest na poziomie MP i w związku z tym funkcja $f^{*k}_{e_{j_{ECP}}}$ musi korzystać z c_0. Jeżeli do generacji trajektorii wykorzystywane są odczyty czujników, to wystąpi argument: v. Natomiast, jeżeli do generacji trajektorii wykorzystuje się sprzężenie zwrotne wysokiego poziomu (tzn. poziomu ECP) od aktualnego stanu efektorów, to wystąpi argument: e. Ponieważ dopuszczalne są wszystkie kombinacje argumentów e i v, funkcja generująca trajektorię przyjmie jedną z ośmiu postaci: $f^{*k}_{e_{MP}}(c_j, \bullet, \bullet, \bullet)$, gdzie $k = 0, \ldots, 8$ oraz symbol $\bullet$ należy rozumieć jako wskazanie, że argument stojący na tym miejscu w liście argumentów funkcji $f^{*k}_{e_{MP}}(c_j, c_0, e, v)$ może istnieć lub nie. Numery k przyporządkowano funkcjom $f^{*k}_{e_{MP}}$ wykorzystując odpowiednik dwójkowy dziesiętnej reprezentacji liczby k. W ten sposób dla $k = 0, \ldots, 7$ uzyskuje się kody: 000, 001, 010, 011, 100, 101, 110, 111. W trójkach (c_0, e, v) 0 oznacza brak danego argumentu, natomiast 1 jego występowanie. W ten sposób powstają funkcje: $f^{*0}_{e_{jECP}}(c_j)$, $f^{*1}_{e_{jECP}}(c_j, v)$, $f^{*2}_{e_{jECP}}(c_j, e)$, $f^{*3}_{e_{jECP}}(c_j, e, v)$, $f^{*4}_{e_{jECP}}(c_j, c_0)$, $f^{*5}_{e_{jECP}}(c_j, c_0, v)$, $f^{*6}_{e_{jECP}}(c_j, c_0, e)$, $f^{*7}_{e_{jECP}}(c_j, c_0, e, v)$, Powyższe rozważania zostały podsumowane w tabeli 5.1. Znakiem $-$ podkreślono brak danego argumentu. Rozróżnienie rodzajów współpracy następuje poprzez określenie typu informacji przesyłanej z procesu MP (tabela 5.2). Funkcje generatorów na poziomie MP i ECP w zależności od realizowanego zadania przedstawiono w tabeli 5.3.

Kryterium podziału:	**Typ funkcji**	
Działanie robotów	niezależne (brak kontaktu z MP) $f^{*k}_{e_{j_{ECP}}}(c_j, -, \bullet, \bullet)$ $k = 0, \ldots, 3$	współpraca (istnieje kontakt z MP) $f^{*k}_{e_{j_{ECP}}}(c_j, c_0, \bullet, \bullet)$ $k = 4, \ldots, 7$
Wykorzystanie czujników	bez czujników (brak kontaktu z VSP) $f^{*k}_{e_{j_{ECP}}}(c_j, \bullet, \bullet, -)$ $k = 0, 2, 4, 6$	z czujnikami (istnieje kontakt z VSP) $f^{*k}_{e_{j_{ECP}}}(c_j, \bullet, \bullet, v)$ $k = 1, 3, 5, 7$
Wykorzystanie informacji o aktualnym stanie robota	bez sprzężenia zwrotnego na poziomie ECP $f^{*k}_{e_{j_{ECP}}}(c_j, \bullet, -, \bullet)$ $k = 0, 1, 4, 5$	ze sprzężeniem zwrotnym na poziomie ECP $f^{*k}_{e_{j_{ECP}}}(c_j, \bullet, e, \bullet)$ $k = 2, 3, 6, 7$

Tabela 5.1: Klasyfikacja generatorów procesu ECP

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

Tabela 5.2: Wpływ rodzajów współpracy robotów na generatory

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 zazwy-

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 5.3: Funkcje generatorów w zależności od realizowanego zadania.

czaj 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.

5.6. Warunki wstępne

Monitorowanie warunku wstępnego $f_I(c^i, e^i, v^i)$ (**??**) wykonania instrukcji ruchu (rys. 2.2) przyjęło się, jak już wspominano, rozpatrywać oddzielnie od samej instrukcji przemieszczającej efektory – czyni to instrukcja `Wait`.

```
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
```

5.7. Instrukcje `Move` i `Wait` na poziomie ECP

Oczywiście w przypadku obiektu klasy wywiedzionej z klasy `generator`, generator konkretny będzie wytwarzał trajektorię dla jednego robota, a nie dla wielu jak to było w przypadku MP, więc lista robotów może być zastąpiona przez pojedynczego robota konkretnego. Instrukcja `Move`, na tym poziomie, przyjmie następującą postać.

```
void Move (robot& the_robot, list<sensor>* sensor_list, generator& the_generator) {
// Funkcja 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 koncowy spelniony 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
   the_robot.execute_motion(); // zlecenie ruchu SET oraz odczyt stanu robota GET

   // 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, the_robot) ); // end: do
}; // end: Move()
```

Instrukcja `Wait` dla ECP również różni się jedynie od `Wait` dla MP drobnym szczegółem: zamiast listy robotów do funkcji realizującej badanie warunku początkowego przekazywany jest pojedynczy robot konkretny.

```
void Wait (robot& the_robot, list<sensor>* sensor_list, condition& the_condition) {
// Funkcja 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: do
}; // end: Wait()
```

5.8. Elementy dodatkowe

Do tworzenia programu użytkowego na poziomie ECP przydatne są następujące funkcje:

- `operator_reaction` — zadająca pytanie operatorowi systemu,
- `input_integer` — żądająca od operatora systemu wprowadzenia liczby całkowitej,
- `input_double` — żądająca od operatora systemu wprowadzenia liczby rzeczywistej,
- `load_file` — tworząca generator trajektorii na podstawie danych zawartych w pliku,
- `teach` — tworząca generator trajektorii poprzez uczenie,
- `save_file` — tworząca plik z nauczonymi pozycjami,
- `save_extended_file` — tworząca plik z nauczonymi pozycjami oraz z danymi kalibracyjnymi.

5.9. 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 6

Procesy EDP

6.1. Rola procesu EDP

Proces EDP spełnia rolę interpretera zleceń przysyłanych do procesu EDP przez proces ECP lub UI. Jego zadaniem jest:

- zdekodowanie rozkazu i sprawdzenie jego poprawności,
- wykonanie adekwatnych operacji na parametrach zlecenia (np. przeliczenie współrzędnych),
- wykonanie rozkazu (np.: zmiana definicji narzędzia, odczytanie wejść binarnych, zapisanie wyjść binarnych, odczyt aktualnego położenia ramienia)
- zlecenie wykonania ruchu, jeżeli przysłano rozkaz ruchu (`SET ARM`), odczytu położenia (`GET ARM`) lub synchronizacji (`SYNCHRO`),
- uformowanie przesyłki zwrotnej dla zleceniodawcy (ECP lub UI),
- wysłanie do ECP przesyłki zwrotnej w odpowiedzi na zlecenie `QUERY`.

6.2. Opis poleceń dla EDP

Lista poleceń dla procesu EDP składa się z następujących grup rozkazów (rys.6.1):

`SYNCHRO` — synchronizacji robota,

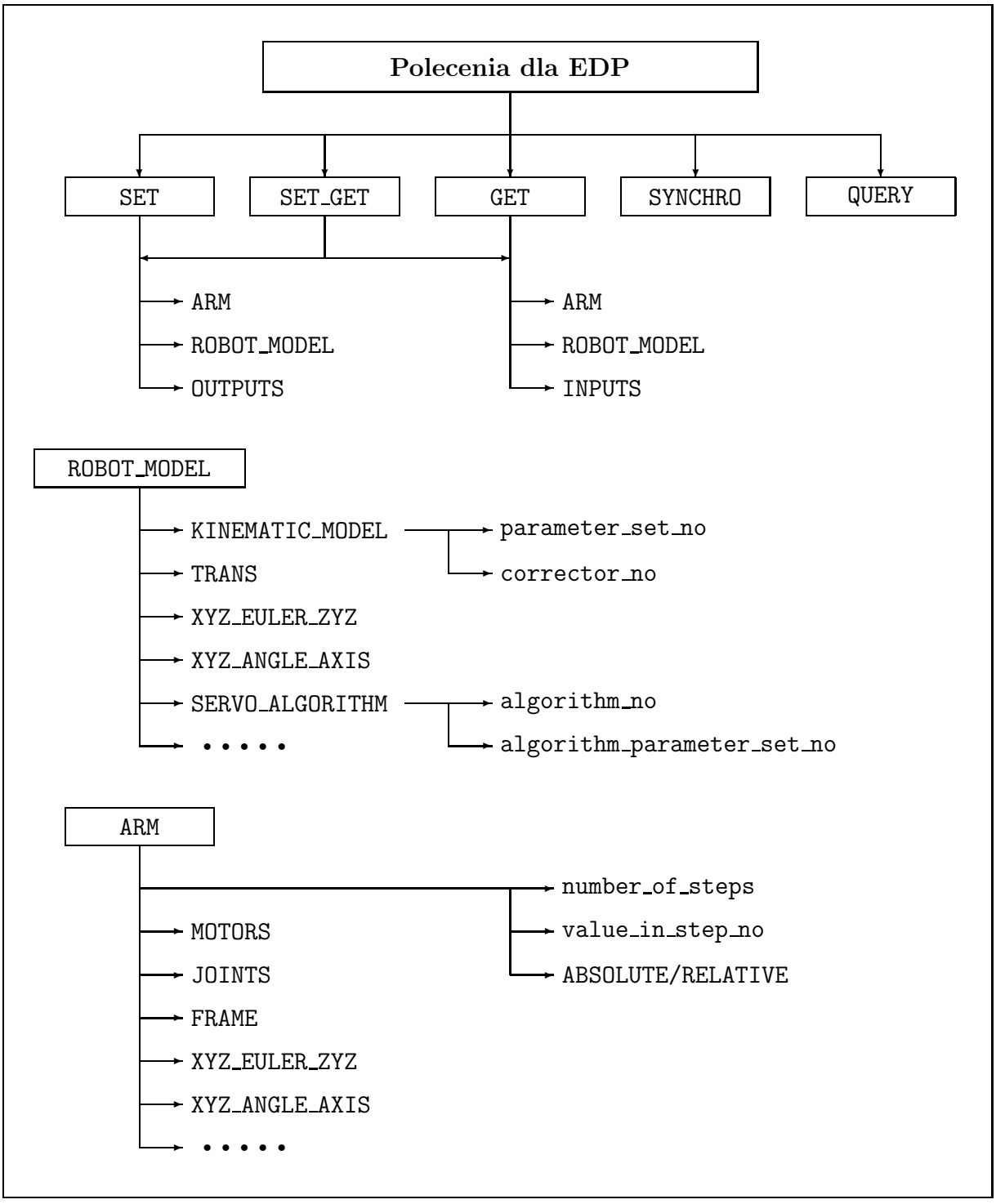
`SET` — zapisu: definicji narzędzia, stanu wyjść, parametrów regulatorów, parametrów modelu kinematycznego lub położenia manipulatora,

`GET` — odczytu: definicji narzędzia, stanu wejść, parametrów regulatorów, parametrów modelu kinematycznego lub położenia manipulatora,

`SET_GET` — jednoczesnego zapisu i odczytu powyższych składowych systemu,

`QUERY` – żądania przesłania odpowiedzi.

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).



Rys. 6.1: Lista poleceń dla EDP

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 MOTOR RELATIVE`). Inne formy rozkazu `SET ARM` są niedopuszczalne w tym stanie i powodują błąd. 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 jak 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 (rys.6.1):

- zmiany definicji 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 (tylko tych, które można zmieniać programowo),
- 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 (rys.6.1):

- odczytanie definicji 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).

Przyjęto wyrażać współrzędne kartezjańskie w *mm*, natomiast kąty w *rad*. 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 a następnie wysłany, do wyższych warstw sterownika, odczyt położenia. 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.

6.3. Struktura procesu EDP

Istnieją dwie metody współdziałania procesów:

- *peer to peer*,
- *client–server*.

Pierwsza nie wyróżnia żadnego z procesów – każdy ma prawo zwracania się do każdego innego. W drugiej jedynie klient może zwracać się do procesu obsługującego zlecenia. Obsługa zleceń przez *driver* odbywa się na zasadzie *client–server*. Wykonanie każdego z procesów tworzących *driver* robota składa się z dwóch zasadniczych faz:

- fazy inicjacyjnej,
- fazy właściwej – realizacji zleceń i obsługi zdarzeń zewnętrznych.

Po uruchomieniu procesu następuje faza inicjacji, która obejmuje:

- rejestrację procesu w systemie operacyjnym,
- powołanie procesów potomnych (jeśli takie istnieją dla danego procesu),
- lokalizację procesów, z którymi będzie nawiązana komunikacja (dotyczy to procesów składowych driver'a).

Po prawidłowym wykonaniu wstępnej inicjacji proces przechodzi do realizacji swych właściwych funkcji. W przypadku procesu EDP_MASTER polega to na tym, że proces zawiesza się w oczekiwaniu na zlecenie z wyższej warstwy sterownika (tj. procesu ECP lub UI). Klienci (procesy ECP lub UI), którzy chcą zlecić *driver'owi* wykonanie określonych usług przekazują podczas *spotkania* wiadomość zawierającą kod odpowiedniego zlecenia. Wynik wykonania zlecenia przesyłany jest do procesu zleceniodawcy, a *serwer* wykonuje dane zlecenie i ponownie zawiesza się oczekując na kolejne polecenie.

Proces wykonuje się w pętli nieskończonej realizując odpowiednie zlecenia i zmieniając swój stan dopóki nie zostanie wysłany sygnał wymuszający zakończenie jego działania.

6.3.1. Proces EDP_MASTER

6.3.1.1. Interpreter poleceń

Proces EDP_MASTER spełnia rolę interpretera poleceń przysyłanych do procesu EDP przez proces ECP. Jego zadaniem jest:

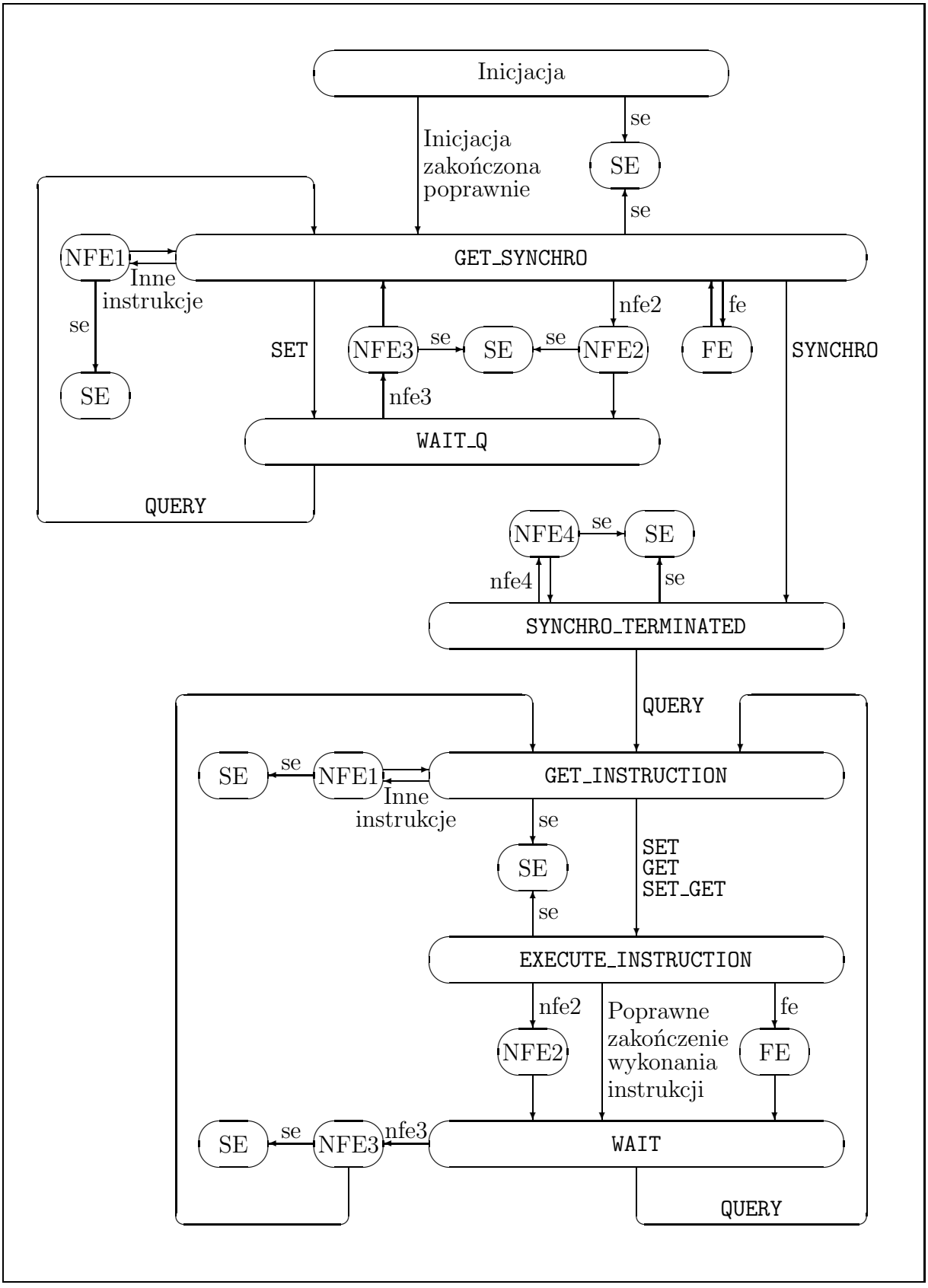
- rozkodowanie rozkazu i sprawdzenie jego poprawności,
- wykonanie adekwatnych operacji matematycznych na parametrach polecenia (np. przeliczenie współrzędnych),
- wykonanie rozkazu (np.: zmiana definicji narzędzia, odczytanie wejść binarnych, zapisanie wyjść binarnych, odczyt aktualnego położenia ramienia)
- zlecenie wykonania ruchu, jeżeli przysłano rozkaz ruchu (`SET ARM`) lub synchronizacji (`SYNCHRO`),
- odebranie informacji o sposobie wykonania ruchu przez procesy realizujące serworegulację,
- uformowanie przesyłki zwrotnej dla zleceniodawcy (ECP),
- wysłanie do ECP przesyłki zwrotnej w odpowiedzi na polecenie `QUERY`.

Proces EDP_MASTER ma strukturę automatu skończonego (rys. 6.2), zmieniającego swój stan pod wpływem napływających rozkazów. Obsługę błędów zrealizowano jako reakcję na wyjątki zgłaszane w różnych częściach programu. Reakcja zawsze następuje na tym samym poziomie – tzn. na poziomie najwyższym, niezależnie od miejsca gdzie wykryto błąd. Na najwyższym poziomie błąd jest analizowany i jest podejmowana decyzja, od którego stanu automat powinien wznowić działanie. Oczywiście zleceniodawca jest informowany o zaistniałej awarii.

Jak już wspomniano, proces EDP_MASTER stanowi automat skończony, który zmienia swój stan pod wpływem poleceń przychodzących od wyższych warstw sterownika. Ponieważ nie w każdym stanie automat może przyjąć dowolne polecenie, a ponadto w trakcie realizacji poszczególnych poleceń mogą zostać wykryte błędy, automat ma możliwość przechodzenia do stanów, w których następuje adekwatna reakcja na błędy. Wyróżniono trzy typy błędów:

- niefatalne (nfe),
- fatalne (fe),
- systemowe (se).

Błędy niefatalne powodowane są niewłaściwymi parametrami poleceń lub poleceniami, które nie mogą być obsłużone w danym stanie automatu. Powodują one przejście do jednego ze stanów NFE, przy czym wyróżniono cztery takie stany: NFE1, NFE2, NFE3 i NFE4. Różnią się one sposobem reakcji na błędy, wynikającym z tego, że automat musi się inaczej zachować w zależności od stanu, w którym się znajduje w momencie wykrycia błędu. Dla zachowania przejrzystości rysunku 6.2 zdecydowano się w grafie umieścić te same stany wielokrotnie, by nie powstała plątanina łuków dochodzących do tych stanów. Przyjęto, że stany o tej samej nazwie są tożsame. Błędy fatalne powodowane są awariami sprzętu sygnalizowanymi przez serwomechanizmy. Błędy te powodują przejście do stanu FE. Automat potrafi powrócić do poprawnego działania po sygnalizacji i obsłudze zarówno błędów fatalnych jak i niefatalnych. Błędy systemowe powstają wskutek wadliwego działania sieci komputerów wykonującej program sterowania. Wykrycie takiego błędu oznacza niesprawność samego sterownika, a więc jest sygnalizowany operatorowi, a sterownik kończy działanie. Automat przechodzi do stanu SE, który jest stanem terminalnym. Łuki w grafie etykietowane są albo poleceniami dla EDP albo typami wykrytych



Rys. 6.2: Graf stanów procesu EDP_MASTER

błędów (nfe1, nfe2, nfe3, nfe4, fe, se). Łuki bez etykiet oznaczają wszystkie pozostałe sytuacje, które mogą wystąpić w danym stanie.

6.3.1.2. Przeliczniki współrzędnych

Jednym z podstawowych zadań procesu EDP_MASTER jest transformacja współrzędnych. Dane potrzebne do tego przekształacenia oraz funkcje realizujące zleconą transformację tworzą pojedynczy obiekt klasy `transformer`. Klasa `transformer` jest *klasą bazową*. Obiekt klasy `transformer` jest odpowiedzialny za przechowywanie położenia ramienia wyrażonego w jeden z następujących sposobów:

- jako położenia wałów silników,
- jako współrzędne wewnętrzne, a więc kąty między członami ramienia lub długości charakterystyczne elementów łańcucha kinematycznego,
- jako współrzędne zewnętrzne aktualnie zamontowanego na ramieniu narzędzia, przy czym współrzędne te wyrażone są za pomocą macierzy przekształcenia jednorodnego.

Ponadto *klasa* ta ma *metody* służące do zmiany aktualnego narzędzia. Położenie i orientacja narzędzia względem kołnierza[1] mogą być wyrażone jako: macierz jednorodna trójścianu związanego z narzędziem, współrzędne kartezjańskie początku i kąty Eulera trójścianu z nim związanego, współrzędne kartezjańskie początku oraz wersor osi obrotu i kąt obrotu wokół niego trójścianu związanego z narzędziem.

Ostatnią funkcją *klasy bazowej* jest obsługa ewentualnych błędów, które mogą powstać w trakcie przeliczania współrzędnych. Dla uproszczenia programowania obsługa błędów jest realizowana jako *obsługa wyjątków*. Takie podejście czyni kod o wiele czytelniejszym, przez co staje się on łatwiejszy do napisania i modyfikacji. W miejscu powstania błędu wstawia się jego numer do *składowej* error oraz zgłasza się wyjątek (throw) zbiorczy `NonFatal_error`. Wyjątek ten jest wyłapywany (`catch`) przez funkcję `main` procesu, w którym zdefiniowano *obiekt klasy pochodnej* `transformer`.

Inne klasy zdefiniowane w procesie EDP_MASTER są klasami dziedzicznymi klasy bazowej `transformer`.

6.3.1.3. Korektory kinematyczne

W celu umożliwienia lokalnej modyfikacji modelu kinematycznego robota wprowadzono *korektory kinematyczne*. Przykładowo po rozwiązaniu odwrotnego zadania kinematyki, otrzymany wynik jest argumentem wejściowym korektora, który dokonuje dodatkowych przeliczeń i w efekcie otrzymuje się zmodyfikowane położenie i orientację końcówki. Stosowane są zazwyczaj proste korektory liniowe. Do zmiany korektora kinematycznego służy polecenie służy polecenie `SET ROBOT_MODEL KINEMATIC_MODEL corrector_no`, gdzie `corrector_no` oznacza numer korektora. Oczywiście korektor o zadanym numerze musi być wcześniej wprowadzony. Sprawdzenia aktualnego numeru korektora można dokonać zleceniem `GET ROBOT_MODEL KINEMATIC_MODEL`. Wprowadzenie nowego korektora wiąże się z napisaniem funkcji dokonującej odpowiednich obliczeń i nadanie temu korektorowi kolejnego numeru i rozszerzenie instrukcji wyboru o wprowadzony numer.

[1]Kołnierz jest końcowym fragmentem ramienia manipulatora, do którego mocowane jest wymienne narzędzie.

6.3.1.4. Przełączanie parametrów modelu kinematycznego

Jakość modelu kinematycznego robota zależy od znajomości parametrów kinematycznych robota. Dokładane określenie wartości parametrów kinematycznych jest często bardzo trudne. Z wielu względów (np. przy kalibracji robota, dla uzyskania odpowiedniej dokładności pozycjonowania końcówki) przydatna jest możliwość zmiany zestawu parametrów modelu kinematycznego robota. Może okazać się ponadto, że uzyskuje się lepszą dokładność w pewnych obszarach przestrzeni roboczej zmieniając parametry modelu kinematycznego. Zmianę zestawu parametrów modelu kinematycznego umożliwia zlecenie `SET ROBOT_MODEL KINEMATIC_MODEL parameter_set_no`, gdzie `parameter_set_no` numer istniejącego zbioru parametrów. Podobnie odczyt aktualnego zestawu parametrów uzyskuje się poprzez zlecenie `GET ROBOT_MODEL KINEMATIC_MODEL`.

Wprowadzenie nowego zestawu parametrów wymaga podania nowych wartości poszczególnych parametrów nadania kolejnego numeru temu zestawowi i rozszerzenie instrukcji wyboru o wprowadzony numer.

6.3.1.5. Rola przy synchronizacji

Rola EDP_MASTER przy realizacji zlecenia `SYNCHRO` sprowadza się do sprawdzenia dopuszczalności tego zlecenia w aktualnym stanie i przesłaniu do realizacji przez proces SERVO_GROUP. Całość synchronizacji jest wykonywana przez proces SERVO_GROUP, zaś wynik jest przekazywany do procesu EDP_MASTER. Jeśli synchronizacja przebiegła pomyślnie, EDP_MASTER przechodzi do innego stanu, właściwej pracy w której możliwe jest przyjmowanie i wykonanie pozostałych zleceń. W przypadku wystąpienia błędu w trakcie synchronizacji, proces proces EDP_MASTER po otrzymaniu informacji o tym fakcie przesyła komunikat do SRP o wystąpieniu błędu fatalnego i w odpowiedzi na zlecenie `QUERY` z żądaniem podania wyniku realizacji zlecenia SYNCHRO przesyła numer odpowiedniego błędu do procesu klienta (ECP lub UI).

6.3.2. Obsługa błędów

Dla systemów robotycznych szczególnie istotna jest analiza powstałych błędów. Ponieważ system sterujący jest tworzony przez użytkownika, wskutek nieuwagi może on wprowadzić do sterownika błędy albo mogą one być konsekwencją złego działania sprzętu.

W systemie `MRROC++` wyróżniono następujące przyczyny błędów:

- **błędy niefatalne** – niewłaściwe polecenia lub ich argumenty,
- **błędy fatalne robota** – kłopoty ze sprzętem sterującym robota (np. przeciążenie),
- **błędy systemowe** – kłopoty z siecią sterującą (np. niemożność nawiązania komunikacji międzyprocesowej).

Przewidziano następujące reakcje na błędy:

- niefatalne – informacja dla warstw nadrzędnych i przywrócenie zdolności obsługi dalszych poleceń,
- fatalne robota – informacja dla warstw nadrzędnych i przywrócenie zdolności obsługi dalszych poleceń,
- fatalne systemowe – informacja dla operatora oraz zakończenie działania sterownika.

Zdecydowano się reakcję na błędy zrealizować jako obsługę wyjątków. W ten sposób kod w przypadku bezbłędnego funkcjonowania systemu oraz kod obsługujący błędy są w dużym stopniu rozdzielne, co znacznie upraszcza pisanie programów i zwiększa ich czytelność.

6.3.2.1. Obsługa błędów w SERVO_GROUP

Sterowniki sprzętowe, prócz informacji o aktualnym położeniu wałów silników, przekazują procesowi SERVO_GROUP informacje o swoim stanie. Na rysunku 6.3 przedstawiono format przesyłanych danych. Są to:

- `overcurrent` — ustawiony, gdy przekroczono dopuszczalny prąd w silniku,
- `upper limit switch` — ustawiony, gdy najechano na górny wyłącznik krańcowy,
- `lower limit switch` — ustawiony, gdy najechano na dolny wyłącznik krańcowy,
- `synchro switch` — ustawiony, gdy najechano na wyłącznik synchronizacji,
- `synchro T` — ustawiony, gdy impulsator wygeneruje impuls T (robi to raz na obrót).

Jedynie trzy pierwsze bity sygnalizują błąd, więc dwa ostatnie są programowo filtrowane. Powyższe trzy bity z każdego serwomechanizmu są łączone w jedno słowo i wraz z dodatkową informacją o błędzie zależną od stanu samego procesu SERVO_GROUP przesyłane są do procesu EDP_MASTER (rys. 6.3) w pakiecie komunikacyjnym, w skład którego wchodzi struktura `edp_error`. Powyższe słowo stanowi składową `error0` tej struktury. Składowa `error1` wypełniana jest, gdy procesowi SERVO_GROUP przekazane zostaną przez EDP_MASTER błędne numery algorytmów regulacji bądź zestawów parametrów tych regulatorów.

6.3.2.2. Obsługa błędów w EDP_MASTER

Błędy wykryte na poziomie SERVO_GROUP przesyłane są pakietem komunikacyjnym do EDP_MASTER. Tu składowa `error0` struktury `edp_error` uzupełniana jest informacją o stanie procesu EDP_MASTER, w którym wykryto błąd. W tym samym miejscu umieszczane są kody błędów wykrytych w trakcie działania EDP_MASTER , a nie spowodowanych działaniem SERVO_GROUP lub sprzętu. Najczęściej są to błędy numeryczne związane z rozwiązywaniem prostego i odwrotnego zagadnienia kinematyki lub błędną zawartością pakietów komunikacyjnych przysyłanych z ECP.

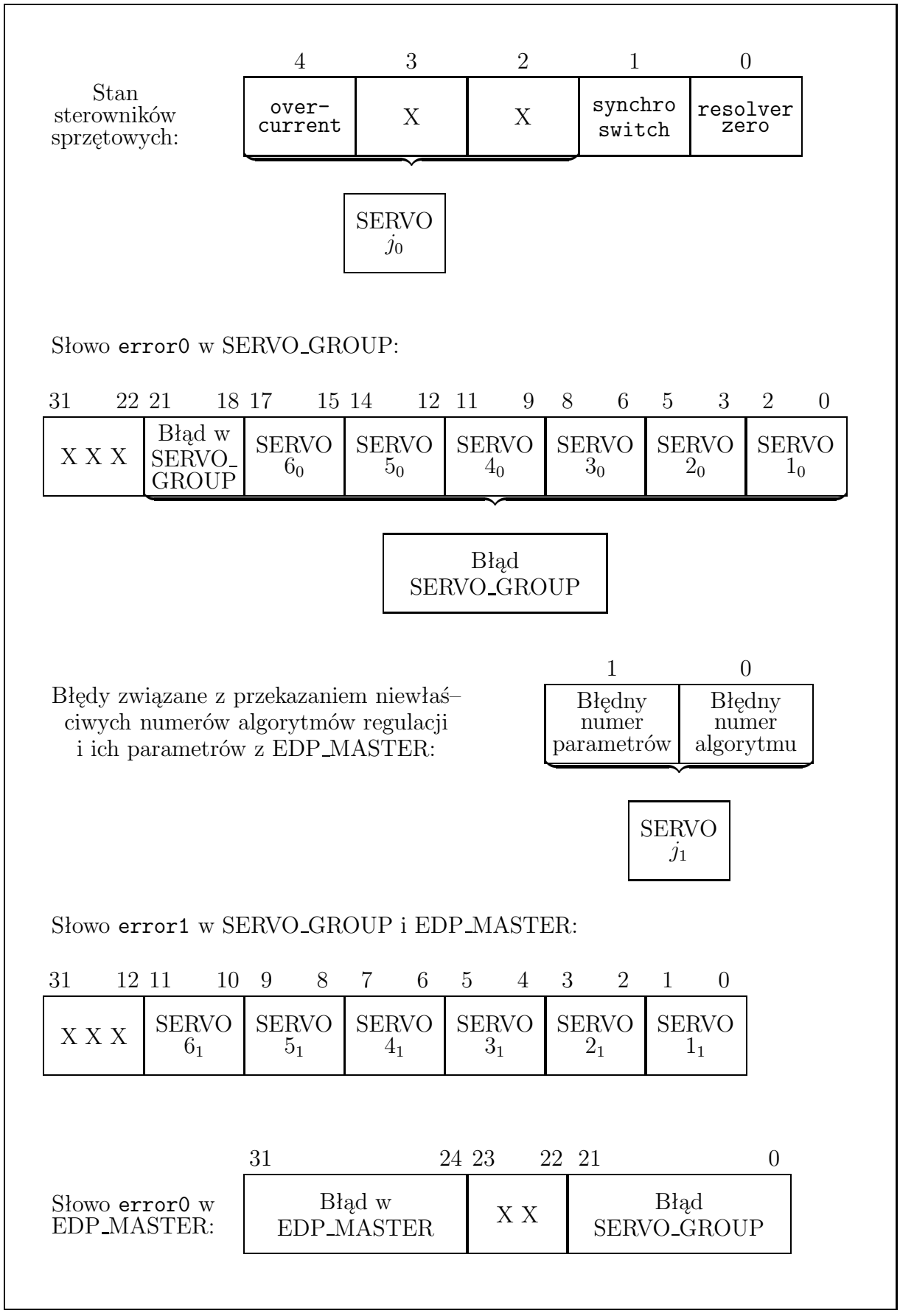
6.3.3. Ingerencja użytkownika w proces EDP

6.3.3.1. Wprowadzenie nowego algorytmu regulacji

Dołączenie nieistniejącego jeszcze w systemie algorytmu regulacji polega na wpisaniu adekwatnego kodu do instrukcji `switch` w metodzie `compute_set_value` klasy wywiedzionej z `NL_regulator`. Ponadto należy uaktualnić adekwatny konstruktor, tak aby w momencie uruchamiania systemu ten regulator był włączany (chyba, że nie jest to podstawowa wersja algorytmu regulacji).

6.3.3.2. Wprowadzenie nowego zestawu parametrów algorytmu regulacji

Dołączenie nieistniejącego jeszcze w systemie zestawu parametrów do istniejącego już algorytmu polega na wpisaniu adekwatnego kodu do instrukcji `switch` w metodzie `compute_set_value` klasy wywiedzionej z `NL_regulator`. Ponadto należy uaktualnić adekwatny



Rys. 6.3: Format błędów wykrywanych w EDP

konstruktor, tak aby w momencie uruchamiania systemu ten regulator był włączany (chyba, że nie jest to podstawowa wersja algorytmu regulacji).

6.3.3.3. Wprowadzenie nowego zestawu parametrów modelu kinematycznego

Aby wprowadzić nowy zestaw parametrów modelu kinematycznego należy w procesie EDP_MASTER w obiekcie klasy `transformer` utworzyć nową metodę o nazwie `set_kinematic_model_`i`_parameters`, gdzie i jest kolejnym numerem zestawu. W metodzie tej należy nadać wartość wszystkim parametrom modelu kinematycznego, tzn.:

Obecnie istnieją w systemie dwie metody o identycznych parametrach. Jeżeli nowo wprowadzany zestaw parametrów modelu kinematycznego ma być zestawem podstawowym, należy to zaznaczyć w konstruktorze obiektu klasy `transformer`. W tym celu należy wstawić do tego obiektu instrukcje: `kinematic_model=`i`;` oraz `set_kinematic_model_`i`_parameters();` zamiast poprzedniej wersji. Ponadto należy rozszerzyć instrukcje `switch` o nowe przypadki `case` i`:` w metodzie `set_rmodel` obiektu klasy `edp_buffer`. Tak przygotowany wariant parametrów modelu kinematycznego może być uaktywniony poleceniem SET ROBOT_MODEL KINEMATIC_MODEL wysłanym do procesu EDP_MASTER.

6.3.3.4. Wprowadzenie nowego korektora modelu kinematycznego

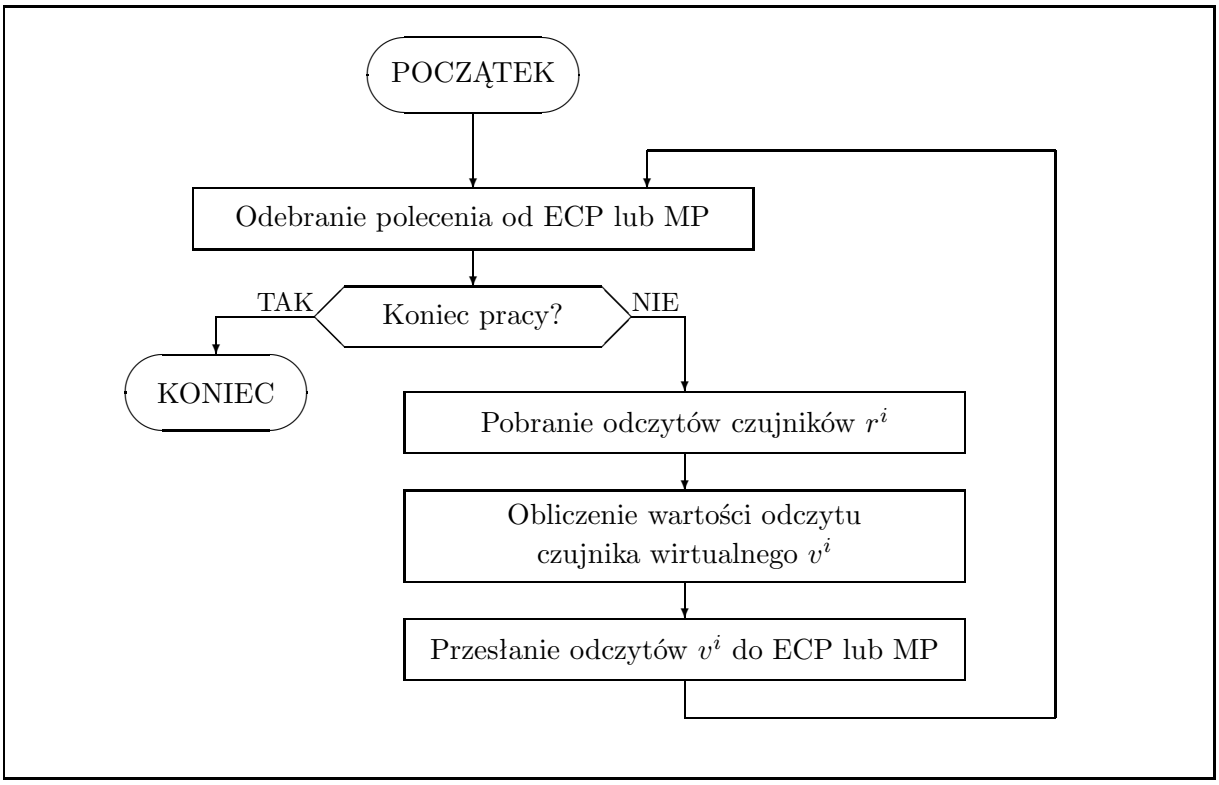
Aby wprowadzić nowy korektor modelu kinematycznego należy w procesie EDP_MASTER w metodzie `set_rmodel` w obiekcie klasy `edp_buffer` rozszerzyć instrukcje `switch` o nowe przypadki `case` i`:`, gdzie i jest kolejnym numerem korektora. Wewnątrz tych przypadków należy umieścić albo kod korektora albo wywołać funkcję (metodę) spełniającą rolę korektora. Jeżeli nowo wprowadzany korektor modelu kinematycznego ma być zestawem podstawowym, to należy to zaznaczyć w konstruktorze obiektu klasy `transformer`. W tym celu należy wstawić do tego obiektu instrukcję: `kinematic_corrector=`i`;`. Tak przygotowany wariant korektora modelu kinematycznego może być uaktywniony poleceniem SET ROBOT_MODEL KINEMATIC_MODEL wysłanym do procesu EDP_MASTER.

Rozdział 7

Procesy VSP

7.0.4. Schematy współpracy z ECP

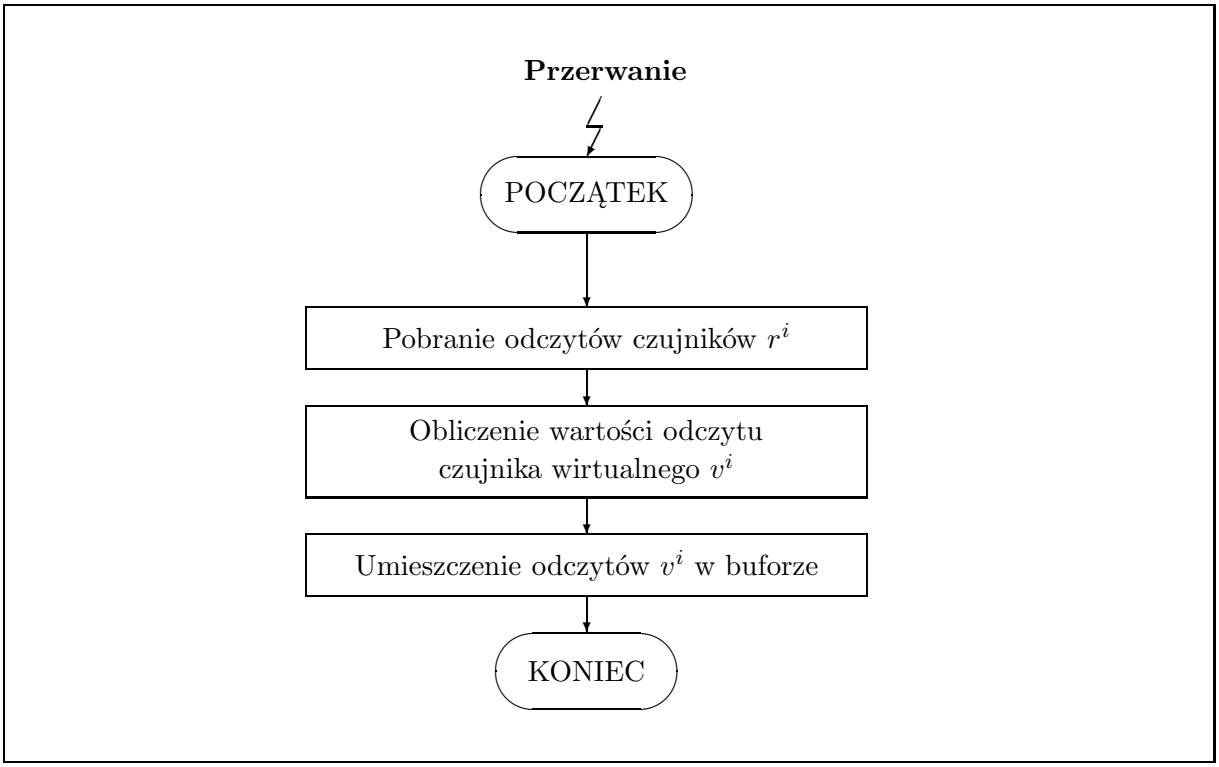
Przyjęto, że każdy z procesów VSP (*Virtual Sensor Process*), w zależności od tego czy współpracuje interaktywnie czy nieinteraktywnie z procesami ECP, albo na żądanie albo wskutek upłynięcia określonego czasu, dokonuje odczytu czujników rzeczywistych, agreguje dane z nich otrzymane oraz wysyła wynik, czyli odczyt czujnika wirtualnego, do ECP. Procesy ECP oraz proces MP wykorzystują te odczyty za pośrednictwem swych instrukcji `Wait` oraz `Move`. Ściślej rzecz ujmując, to czujniki konkretne wskutek zastosowania odpowiednich metod żądają nadesłania nowych danych oraz odbierają przesyłki, a następnie przechowują otrzymany odczyt i udostępniają go instrukcjom ruchowym.



Rys. 7.1: Sieć działań procesu VSP współpracującego interaktywnie z ECP lub MP

Każdy proces VSP ma standardową postać. W przypadku współpracy interaktywnej z procesami ECP lub MP jego struktura jest taka jak pokazano na rysunku 7.1, natomiast w przypadku pracy nieinteraktywnej algorytm jego działania przedstawia rysunek 7.2.

Oczywiście do realizacji tych algorytmów można również użyć czujników konkretnych wywiedzionych z `sensor`. W tym jednak przypadku metody virtualne będą odwoływały się do sprzętu (czujników rzeczywistych) bezpośrednio, a nie wysyłały żądania i odbierały odczyty od innych procesów.



Rys. 7.2: Sieć działań procesu VSP współpracującego nieinteraktywnie z ECP lub MP

7.0.5. Proces VSP: przykład realizacji

Klasa konkretna czujnika ma, dla omawianego przykładu klawiszowego czujnika dwustanowego, po stronie VSP, postać:

```
class vsp_bin_sensor {
 public:

   // bufory przesylowe
   VSP_ECP_MSG from_vsp; // Wiadomosc przesylana pomiedzy VSP - ECP
   ECP_VSP_MSG to_vsp;   // Wiadomosc przesylana pomiedzy ECP - VSP

   // wewnetrzne dane czujnika
   pid_t pid;
   char VSPname[50];
   unsigned int cap; /*16bits: read word */
   int error;
   unsigned int  low_hword, high_hword;// 8 bits data


   vsp_bin_sensor ( void );
     // konstruktor

   ~vsp_bin_sensor ( void ) {};
     // destruktor
```

```
   void GetSensorReading ( void );
     // odczyt z hardwaru

  BOOLEAN  get_ecp_demand ( void );
    // kontakt z ECP

}; // end: class vsp_bin_sensor
```

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

Funkcja `GetSensorReading` dokonuje odczytu czujników i podstawia odczyty do bufora komunikacyjnego `from_vsp` w postaci jednej danej typu integer (osiem bitów - osiem klawiszy) albo ośmiu liczb całkowitych, z których każda odpowiada stanowi jednego klawisza (liczba `0` albo `1`).

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 prawidłowa (`FLOAT_READING`: wtedy przygotowywane są dane skumulowane, albo `INTEGER_READING` - osiem pól danych) dokonywany jest odczyt poprzez wywołanie funkcji `GetSensorReading`. Jeżeli z VSP 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_fs.get_ecp_demand())`) dopóty dopóki zwraca ona wartość `TRUE`, natomiast przekazanie przez nią wartości `FALSE` powoduje zakończenie programu głównego.

Rozdział 8

Proces UI

8.1. Struktura i funkcje procesu User Interface

Warstwa komunikacji użytkownika ze sterownikiem robota składa się z dwóch modułów: procesu obsługującego polecenia operatora User Interface (UI) oraz procesu wyświetlania stanu systemu sterującego System Response Process (SRP). Oba procesy realizują komunikację z operatorem za pośrednictwem *okienkowego środowiska graficznego*. Procesy UI i SRP mają zamkniętą strukturę i nie podlegają modyfikacji ani rekompilacji przez użytkownika.

Z punktu widzenia użytkownika istotne są mechanizmy komunikacji pomiędzy stworzonym przez niego nadrzędnym procesem sterującym Master Process (MP) a procesem User Interface. Mechanizmy te pozwalają na sterowanie przebiegiem wykonania procesów tworzących sterownik robota z poziomu interfejsu użytkownika. Podstawowym mechanizmem komunikacji między procesem UI a procesem MP jest asynchroniczny przekaz stałych komunikatów za pomocą tzw. *pośredników* (ang. *proxies*). Każdy z *pośredników* służy do przesyłania jednego polecenia wybranego ze skończonej listy dostępnych poleceń operatora. Powołanie *pośredników* zapewniających właściwą komunikację z procesem UI następuje w procesie sterującym MP pisanym przez użytkownika, zaś identyfikatory *pośredników* są przekazywane do procesu UI jako wiadomość w trakcie spotkania. Taki sposób tworzenia narzędzi komunikacji wynika z faktu, że odebrać wiadomość od *pośrednika* może jedynie jego właściciel, czyli proces powołujący go. Wzbudzić *pośrednika* może natomiast dowolny proces, który zna jego identyfikator. Ponadto mechanizm spotkań jest wykorzystywany do bezpośredniego kontaktu pomiędzy procesami ECP i procesem UI. Dla pewnych zadań, np. dla „uczenia" robotów, konieczna jest taka bezpośrednia komunikacja. Po stronie UI istnieją procedury obsługi odpowiednich zleceń przesyłanych z procesów ECP. Po wykonaniu zlecenia odpowiedź jest przesyłana do właściwego procesu ECP. A zatem poza przesyłaniem zleceń operatora do procesu MP, proces UI odbiera i obsługuje zlecenia wysłane przez ECP i wymagające w czasie wykonywania zadania stosownych reakcji operatora.

8.1.1. Środowisko graficzne

Najprostszy tryb komunikacji użytkownika z programem polega na podawaniu przez użytkownika odpowiedzi na kolejno zadawane przez program pytania. Ten tzw. tryb *pytań i odpowiedzi* jest wysoce nieelastyczny i obecnie jest coraz rzadziej stosowany.

Inny tryb interakcji użytkownik-program polega na wyborze przez użytkownika jednej z możliwych opcji wchodzących w skład hierarchicznie zorganizowanych *menu*. Jego przewagą jest wizualizacja dostępnych możliwości wyboru. Sam wybór odbywa się w sposób graficzny – np. przez wskazanie opcji za pomocą kursora myszki.

Obecnie powszechnie stosowanym sposobem dialogu programu i użytkownika jest *tryb interakcji graficznej*. W trybie tym użytkownik wprowadza dane do programu wskazując na ekranie określone obiekty graficzne. Najczęściej stosowaną wersją interakcji graficznej jest *tryb okienkowy*. Użytkownik może sam wybrać okno aktywne, w którym w danej chwili prowadzi konwersację z programem. Programy posługujące się trybem okienkowym mogą wykorzystywać w poszczególnych oknach inne tryby interakcji (np. menu, polecenia, blankiety). W ten sposób forma dialogu może być dobrana odpowiednio do określonych dla danej funkcji wymagań. Ostatni z przedstawionych trybów komunikacji użytkownika z programem został wykorzystany przy tworzeniu programu procesu UI.

Podstawowym narzędziem programistycznym służącym do tworzenia okienkowego systemu komunikacji użytkownika ze sterownikiem robota jest środowisko *QNX Windows*. Podobnie jak inne środowiska *Graphical User Interface* (GUI) pakiet *QNX Windows* pozwala na związanie kodu funkcji z określonymi obiektami graficznymi takimi jak: okienka, ikony, dialogi, przyciski, suwaki itp. Podstawowym założeniem graficznego interfejsu użytkowego implementowanego w procesie UI jest ułatwienie obsługi sterownika poprzez wizualną komunikację za pośrednictwem przejrzystego systemu okienek, przywoływanych i rozwijanych menu.

8.1.2. Funkcje procesu User Interface

Proces User Interface obsługuje polecenia operatora systemu, zdarzenia będące wynikiem wykonania procesu MP i/lub procesów ECP oraz odpowiada za inicjację i zakończenie działania programowego sterownika robotowego. Funkcje procesu User Interface są następujące:

1. Inicjacja i zakończenie działania sterownika.
   1. Automatyczne uruchamianie oraz usuwanie procesu SRP
   2. „Zabijanie" procesu Effector Driver Process (EDP), jeśli nie został usunięty, po zakończeniu działania sterownika.
2. Obsługa poleceń operatora.
   - Ładowanie do pamięci operacyjnej i uruchomienie procesu EDP — polecenie *LOAD_EDP*
   - Konfigurowanie *driver'a* (procesów EDP i SERVO) (odczyt/zapis: nastaw regulatora, definicji narzędzia) — polecenie SETUP_EDP
   - Usuwanie procesów EDP z pamięci operacyjnej — polecenie *UNLOAD_EDP*
   - Ładowanie do pamięci procesu MP — polecenie *LOAD_MP*
   - Usuwanie procesu MP – polecenie *UNLOAD_MP*
   - Uruchamianie procesu MP — polecenie *START*
   - Zakończenie wykonania procesu MP — polecenie *STOP*
   - Wstrzymanie (w określonych miejscach programu) wykonania procesu MP — polecenie *PAUSE*
   - Wznowienie wykonania procesu MP (tylko po uprzednim wykonaniu polecenia *PAUSE*) — polecenie *RESUME*

- Przerywanie wykonania procesu MP (w dowolnej chwili) na poziomie systemu operacyjnego — polecenie *PAUSE_H*
- Wznowienie wykonania procesu MP na poziomie systemu operacyjnego (tylko po uprzednim wykonaniu polecenia *PAUSE_H*) — polecenie *RESUME_H*
- Obsługa tzw. ruchów ręcznych — grupa poleceń:
  - synchronizacja robota — polecenie *SYNCHRO*
  - ruchy ręczne we współrzędnych położenia wałów silników — polecenie *MOVE_MOTORS*
  - ruchy ręczne we współrzędnych wewnętrznych ramion robota (kątach obrotu) — polecenie *MOVE_JOINTS*
  - ruchy ręczne we współrzędnych zewnętrznych robota (przestrzeń kartezjańska) — polecenie *MOVE_END_EFFECTOR*
- Obsługa uczenia robota/robotów.
- Obsługa reakcji opertora.
- Uzyskanie pomocy w posługiwaniu się narzędziami dostępnymi w środowisku okienkowym UI — polecenie *HELP*.
- Zakończenie działania systemu — polecenie *QUIT*

8.1.3. Opis poleceń operatora

LOAD_EDP — załadowanie do pamięci operacyjnej i uruchomienie procesu Effector Driver Process obsługi danego robota. Pliki wykonywalne uruchamianych procesów stanowiących *driver* robota muszą znajdować się w katalogu podanym jako jeden z parametrów konfiguracyjnych (zapisanych w pliku `ui.cnf`). Nadawana procesowi EDP nazwa globalna jest również zdefiniowana w danych konfiguracyjnych. Proces EDP może być uruchamiany w dowolnym węźle sieci. Numer węzła, w którym uruchamiany jest proces EDP jest definiowany za pomocą polecenia *CONFIG* z poziomu aplikacji UI. Jeśli wybrany węzeł jest nieaktywny, to operator jest informowany komunikatem o niemożności uruchomienia EDP w tym węźle. Przed każdą próbą załadowania procesu EDP jest sprawdzana aktualna konfiguracja. Powołane do życia procesy EDP mają zerwaną relację pokrewieństwa z procesem UI (z punktu widzenia systemu operacyjnego), gdyż do ich uruchomienia wykorzystywana jest funkcja `spawn()` z parametrem $mode = P\_NOWAIT$. Operacja uruchamiania procesu EDP może się nie powieść w przypadku braku pliku wykonywalnego w odpowiednim katalogu lub w sytuacji, gdy proces o danej nazwie jest już zarejestrowany w systemie. Procesy EDP rejestrowane są pod nazwami globalnymi, unikatowymi w całym systemie, przez co próba rejestracji procesu pod już wykorzystaną nazwą jest interpretowana jako błąd. Jeżeli operacja zakończy się błędem, proces UI nie zmieni stanu i nadal będzie dostępna możliwość ładowania procesów EDP. W sytuacji, gdy proces EDP został uruchomiony, operacja ponownego jego uruchomienia będzie dostępna dopiero po wykonaniu polecenia *UNLOAD_EDP*.

UNLOAD_EDP — polecenia zabicia procesu EDP. Operacja usunięcia procesu EDP jest dostępna dopiero po zakończonej sukcesem operacji jego załadowania. Zabicie procesów EDP realizowane jest poprzez wysłanie sygnału *SIGTERM* do procesu EDP. Wykorzystano sygnał *SIGTERM*, ponieważ może on być przechwycony przez proces, do którego został wysłany i przed zakończeniem procesu można wykonać takie czynności jak: usunięcie procesów potomnych (w danym przypadku

procesu SERVO_GROUP) odłączenie procedur obsługi przerwań, zamknięcie łączy komunikacyjnych, itp. Po prawidłowym wykonaniu operacji usunięcia procesu EDP ponownie zostaje udostępniona operacja ładowania danego procesu EDP. Operacja zabicia procesu EDP kończy się niepowodzeniem, jeśli proces, który ma być usunięty, został z jakichś powodów zabity wcześniej.

LOAD_MP – polecenie załadowania procesu Master Process do pamięci operacyjnej. Proces może zostać uruchomiony w dowolnym węźle sieci. Jeśli wybrany węzeł jest nieaktywny, to operator jest o tym fakcie informowany stosownym komunikatem. Aby uniknąć ładowania przypadkowych plików wykonywalnych przyjęto, że plik wykonywalny Master Process powinien mieć nazwę rozpoczynającą się od `mp_m*` (gdzie `*` oznacza dowolną liczbę znaków dopuszczalnych w nazwie pliku). Plik ten może znajdować się w dowolnym katalogu oraz węźle sieci. Wszystkie te dane są umieszczone w odpowiednich parametrach konfiguracyjnych. Operacja uruchamiania i nawiązania komunikacji z procesem MP składa się z kilku faz:

- Ładowanie do pamięci operacyjnej programu Master Process — odbywa się poprzez wywołanie funkcji systemowej `spawn()` z parametrem *mode* = *P_NOWAIT*, czyli zrywana jest relacja pokrewieństwa z procesem UI. Proces MP wywoływany jest z następującymi argumentami:
  - *argv[1]* – przekazuje do procesu MP nazwę, pod którą ma on zarejestrować się – w celu uniemożliwienia istnienia w pamięci drugiego procesu MP (ze względów bezpieczeństawa zakłada się, że proces MP jest rejestrowany zawsze pod tą samą nazwą globalną);
  - *argv[2]* – numer węzła, w którym uruchomiony jest proces UI, parametr ten umożliwia procesowi MP utworzenie wirtualnych *proxy* przypisanych do węzła, na którym wykonuje się proces UI.
- Lokalizacja procesu MP (poprzez poszukiwanie procesu o danej nazwie globalnej) w celu uzyskania jego identyfikatora. Znajomość identyfikatora procesu MP jest konieczna do nawiązania *spotkania* z procesem MP oraz do jego zabicia.
- Odebranie wiadomości od procesu MP (funkcja systemowa `Creceive()` wraz z obsługą przeterminowania). Wiadomość zawiera identyfikatory wirtualnych *proxy*, za pomocą których proces UI komunikuje się z procesem MP. Odbiór wiadomości jest natychmiast potwierdzany w celu odblokowania procesu MP.

Nieprawidłowe zakończenie którejkolwiek z faz powoduje błąd ładowania procesu MP – stan procesu UI nie zmienia się i operacje związane z wykonaniem procesu MP będą niedostępne. Przyczyną wystąpienia błędu podczas ładowania procesu MP może być istnienie w pamięci procesu o tej samej nazwie lub błąd przy nawiązywaniu komunikacji z procesem UI.

UNLOAD_MP — polecenie zabicia procesu MP. Operacja usunięcia procesu MP realizowana jest za pomocą systemowego mechanizmu sygnałów. Do procesu MP wysyłany jest sygnał *SIGTERM*. Operacja ta może nie powieść się, gdy proces MP został już usunięty z pamięci.

START — polecenie uruchomienia programu sterującego stanowiącego fragment ciała procesu MP. Operacja ta jest realizowana przez wzbudzenie odpowiedniego *proxy*, którego właścicielem jest proces MP, a identyfikowanego jako polecenie startu programu sterującego (realizującego właściwe zadanie użytkownika).

STOP — polecenie przerwania wykonania programu sterującego przez proces MP. Wykonanie tej operacji polega na wzbudzeniu *proxy* należącego do MP identyfikowa-

nego jako polecenie przerwania wykonania programu sterującego. Przy ponownym uruchomieniu programu sterującego, program sterujący stanowiący fragment ciała procesu MP będzie wykonywany od początku.

PAUSE — polecenie wstrzymania wykonania procesu MP. Operacja realizowana jest przez wzbudzenie *proxy* należącego do procesu MP, które jest interpretowane jako polecenie przerwania wykonania programu sterującego.

RESUME — polecenie wznowienia wykonania, wstrzymanego poleceniem *PAUSE*, procesu MP. Wykonanie tego polecenia przez proces UI polega na wzbudzeniu odpowiedniego *proxy*, którego właścicielem jest proces MP.

PAUSE_H — polecenie wstrzymania wykonania procesu MP. Operacja realizowana jest z wykorzystaniem sygnałów systemowych. Do procesu MP wysyłany jest sygnał *SIGSTOP*, który powoduje natychmiastowe wstrzymanie wykonania procesu MP, a tym samym programu sterującego.

RESUME_H – polecenie wznowienia wstrzymanego poleceniem *PAUSE_H* procesu MP. Operacja *RESUME_H* polega na wysłaniu sygnału *SIGCONT* do procesu MP, co powoduje przejście procesu MP ze stanu HELD do stanu READY (stany procesów w systemie QNX) i wznowienie wykonania programu sterującego od miejsca, w którym został zatrzymany po wykonaniu operacji *PAUSE_H*.

MANUAL OPERATIONS — grupa zleceń odnoszących się do tzw. ruchów ręcznych:

SYNCHRO – polecenie do procesu lub procesów EDP sterujących robotami wykonania synchronizacji.

MOVE_MOTORS – polecenie umożliwia wykonanie ruchów ręcznych robotem z we współrzędnych wałów silników z jednoczesnym odczytem położenia. Polecenie to może być wykonane również przed synchronizacją robota, wówczas przemieszczenie robota odbywa się poprzez zadawanie kolejnych przyrostów położenia wałów silnika. Wykorzystywane jest do tego celu zlecenie dla procesu EDP (SET ARM MOTOR RELATIVE). Po synchronizacji robota do przemieszczania wałów silników może być wykorzystane zarówno zlecenie (SET_GET ARM MOTOR RELATIVE) gdy zadawane są przyrosty położenia lub (SET_GET ARM MOTOR ABSOLUTE), gdy chcemy osiągnąć określone położenie. W obu przypadkach odczyt bieżącego położenia jest bezwzględny. Możliwe są dwa sposoby wykonywania ruchów ręcznych. Pierwszy polega na przemieszczaniu pojedynczej osi o pewien zadany przyrost (krok ruchu), którego wielkość może być ustawiana. Drugi sposób polega na ustawieniu pozycji docelowej dla poszczególnych osi, a następnie wybranie opcji *Move* co odpowiada wysłaniu zlecenia (SET_GET ARM MOTOR ABSOLUTE). Ruch odbywa się z pewną ustaloną prędkością, zatem liczba kroków ruchu jest dobierana w taki sposób, aby w żadnej osi prędkość nie była większa od ustalonej prędkości.

MOVE_JOINTS – polecenie umożliwia wykonanie ruchów ręcznych we współrzędnych wewnętrznych robota. Przemieszczenia robota zadawane na poziomie położenia ogniw jest możliwe dopiero po synchronizacji robota. Możliwe są jak powyżej, dwa sposoby zadawania położeń ogniw: przyrostowo względem bieżącego położenia – za pomocą zlecenia (SET_GET ARM JOINTS RELATIVE)

lub bezwzględnego położenia korzystając ze zlecenia (SET_GET ARM JOINTS ABSOLUTE). Odczyt położeń ogniw jest zawsze podawany w wartościach bezwzględnych.

MOVE_END_EFFECTOR – polecenie umożliwia wykonanie ruchów ręcznych we współrzędnych zewnętrznych robota. Możliwe jest zadawanie trzech współrzędnych położenia oraz trzech kątów Euler'a Z-Y-Z opisujących orientację końcówki/narzędzia w globalnym układzie współrzędnych. Możliwe są jak poprzednio, dwa sposby zadawania położeń ogniw: przyrostowo względem bieżącego położenia/orientacji – za pomocą zlecenia (SET_GET ARM XYZ_EULER_ZYZ RELATIVE) lub bezwzględnego położenia/orientacji korzystając ze zlecenia (SET_GET ARM XYZ_EULER_ZYZ ABSOLUTE). Odczyt położenia i orientacji końcówki/narzędzia jest zawsze podawany w wartościach bezwzględnych.

HELP — polecenie podania informacje na temat obsługi systemu. Funkcja ta jest dostępna we wszystkich stanach procesu UI i nie ma wpływu na stan procesu UI.

QUIT — polecenie zakończenia wykonania procesu UI. Jego wywołanie powoduje zabicie (poprzez wysłanie sygnału *SIGTERM*) wszystkich procesów powołanych do życia przez proces UI.

Operator steruje robotem/robotami korzystając z okienkowego menu, poruszając się w ramach skończonego zbioru dostępnych poleceń. Zakres dopuszczalnych poleceń jest zależny od aktualnego stanu systemu (stanu poszczególnych procesów sterownika) (np. nie można odczytać położenia robota przed jego synchronizacją).

Polecenia *START*, *PAUSE*, *RESUME* i *STOP* są realizowane przez *pośredników* przenoszących koumunikaty między procesami UI i MP. Rozwiązanie to pozwala na ścisłą kontrolę miejsca wstrzymania lub przerwania wykonania procesu MP. Wymaga jednak zapewnienia prawidłowej implementacji i obsługi mechanizmów komunikacji pomiędzy procesami UI i MP. Realizacja tych poleceń następuje nie natychmiastowo, lecz po dokończeniu wykonania przez proces MP każdej instrukcji tj. polecenia.

Wstrzymanie wykonania procesu MP jest możliwe także bezpośrednio na poziomie systemu operacyjnego QNX. Operacja ta przewidziana jest do użycia w sytuacjach awaryjnych, gdy trzeba natychmiast wstrzymać wykonanie procesu sterującego. Wykorzystuje się tu *sygnały* dostępne w systemie operacyjnym QNX. Zasadnicza różnica w stosunku do programowej wymiany komunikatów polega na natychmiastowym wstrzymaniu wykonania procesu MP bez „jego wiedzy" (przerywane jest wykonanie bieżącego polecenia).

8.1.4. Rola UI przy uczeniu oraz żądaniu reakcji operatora

W trakcie wykonania zadania użytkownika może wystąpić konieczność nauczenia robota pewnych określonych pozycji. Wymaga to przemieszczenia robota do żądanej pozycji i jej zapamiętaniu z możliwością poźniejszego odtworzenia nauczonej pozycji. Po stronie procesu UI konieczne jest stworzenie operatorowi możliwości poruszania robotem i zapamiętywania wybranych pozycji. Ponieważ uczenie dotyczy konkretnego robota, to żądanie uczenia zgłaszane jest przez proces odpowiedni proces ECP. Po odebraniu takiego żądania przez proces UI otwierane jest okienko uczenia robota. Możliwe jest uczenie robota w jednym z wybranych układów współrzędnych (położeń wałów silników, wewnętrznych lub zewnętrznych). Dostępne są opcje jak w menu ruchów ręcznych oraz dwie dodatkowe

opcje *Save* – zapamiętania nauczonej pozycji i *Finish* – zakończenia uczenia. Każda zapmiętana pozycja jest przesyłana do procesu ECP i zapamiętywana jako kolejny element listy nauczonych pozycji.

Przewidziano także możliwość wprowadzenia reakcji operatora w trakcie wykonania zadania. Polega to na tym, że w trakcie wykonania procesów ECP i procesu MP realizujących konkretne zadanie, operator jest zmuszony do odpowiednich reakcji. Reakcje to polegają na odpowiedzi TAK/NIE na postawione pytanie, wprowadzenie liczby całkowitej lub rzeczywistej. Zgłoszenie takiego żądania przez ECP lub MP powoduje pojawienie się stosownego okienka z żądaniem reakcji operatora. Po reakcji operatora odpowiedź jest przesyłana do procesu, który wystąpił z tym żądaniem.

8.2. Stany procesu User Interface

Sterowanie robotami możliwe jest na drodze manualnej oraz poprzez wykonanie programu sterującego użytkownika realizowanego w procesie MP i procesie/procesach ECP.

W procesie UI wyróżnić można następujące stany:

1. **Stan inicjacji** *INITIAL_STATE* — stan procesu UI po uruchomieniu, ale przed załadowaniem procesów EDP. W tym stanie niemożliwe wykonywanie ruchów robotem, ani załadowanie procesu MP, zaś dostępne są następujące polecenia:
   - *QUIT* – zabicie procesu SRP, zakończenie wykonania procesu UI i powrót do konsoli systemu operacyjnego.
   - *LOAD_EDP* – uruchomienie procesu EDP. O tym ile można uruchomić procesów EDP decyduje wartość odpowiedniego parameteru konfiguracyjnego procesu UI (dla wersji wielorobotowej).

   Proces UI może przejść do stanu *INITIAL_STATE* dopiero po prawidłowym wykonaniu następujących operacji: nadaniu procesowi UI niepowtarzalnej w sieci nazwy globalnej, uruchomieniu procesu SRP oraz odczytaniu danych konfiguracyjnych z pliku.
2. **Stan obsługi ręcznej** *MANUAL_OPERATIONS* — do tego stanu proces przechodzi po załadowaniu do pamięci co najmniej jednego procesu EDP. W stanie tym możliwe jest ładowanie kolejnych procesów EDP lub procesu MP, wykonywanie ruchów ręcznych robotem, którego proces EDP został uruchomiony oraz możliwa jest synchronizacja tego robota. Przed wykonaniem synchronizacji robota, możliwe jest jedynie przyrostowe przemieszczanie wałów silników napędowych poszczególnych osi. Dostępne są zatem następujące polecenia:
   - *LOAD_MP* – uruchomienie procesu Master Process
   - *LOAD_EDP* – uruchomienie procesu Effector Driver Process
   - *SYNCHRO* – synchronizacja robota
   - *MOVE_MOTORS* – ruchy ręczne robotem we współrzędnych wałów silników w sposób przyrostowy
   - *UNLOAD_EDP* – zabicie procesu Effector Driver Process
   - *QUIT* – zabicie procesu SRP oraz procesów EDP zakończenie dzialania UI i powrót do systemu operacyjnego

   a po synchroniacji robota polecenia ruchów ręcznych:
   - *MOVE_MOTORS* – ruchy ręczne robotem we współrzędnych wałów silników

- *MOVE_JOINTS* – ruchy ręczne robotem we współrzędnych wewnętrznych
- *MOVE_END_EFFECTOR* – ruchy ręczne robotem we współrzędnych zewnętrznych

3. **Stan załadowania procesu** MP *MP_LOADED_MANUAL_OPERATIONS* — stan ten odpowiada załadowaniu do pamięci operacyjnej procesu MP, który oczekuje w stanie zawieszenia na wykonanie. W stanie tym możliwe jest nadal wykonywanie ruchów ręcznych, ale nie jest już możliwe załadowanie innego procesu MP ani usunięcie procesów EDP. W stanie tym dostępne są oprócz ruchów ręcznych *SYNCHRO*, *MOVE_MOTORS*, *MOVE_JOINTS*, *MOVE_END_EFFCTOR*, następujące polecenia:
   - *START* – wysłanie rozkazu do procesu MP, po odebraniu którego przechodzi on ze stanu zawieszenia do realizacji programu użytkownika.
   - *UNLOAD_MP* – zabicie procesu Master Process i powrót do stanu poprzedniego procesu UI. Po wykonaniu tego polecenia proces może się znaleźć w jednym ze stanów typu *MANUAL_OPERATIONS_i*.
   - *QUIT* – zabicie procesu SRP, procesów EDP i powrót do systemu operacyjnego.
4. **Stan wykonania procesu MP** *MP_RUNNING* — stan UI odpowiadający wykonywaniu się procesu MP. Nie jest możliwe w tym stanie wykonywanie ruchów ręcznych, dostępne zaś są następujące polecenia:
   - *STOP* – polecenie powoduje zatrzymanie wykonania programu użytkownika. Proces MP wraca na początek programu wykonania zadania użytkownika i zawiesza się oczekując na dalsze polecenia (wywołuje funkcję systemową `Receive()` przechodząc do stanu RECEIVE BLOCKED)
   - *PAUSE* – polecenie powoduje wstrzymanie wykonania programu użytkownika i zawieszenie procesu MP (RECEIVE BLOCKED) w oczekiwaniu na dalsze polecenia.
   - *PAUSE_H* – wysłanie sygnału *SIGSTOP* do procesu MP, co powoduje natychmiastowe zawieszenie (przejście do stanu HELD na poziomie systemu operacyjnego) procesu MP a tym samym programu użytkownika.
5. **Stan zawieszenia procesu** MP *MP_PAUSED* – w stanie tym proces MP jest zawieszony (RECEIVE BLOCKED) w oczekiwaniu na polecenie od użytkownika przsyłane za pomocą stosownego *pośrednika* powodującego kontynuację wykonywania programu sterującego użytkownika. W stanie tym niemożliwe jest wykonywanie ruchów ręcznych. Dostępne jest polecenie *RESUME*.
6. **Stan zawieszenia procesu** MP *MP_PAUSED_H* – stan ten odpowiada zawieszeniu procesu MP. Zawieszenie procesu MP odbywa się bez jego wiedzy na poziomie systemu operacyjnego. W stanie tym proces jest wrażliwy tylko na sygnały (znajduje się w stanie HELD), tak więc realizacja polecenia odwieszenia procesu sprowadza się do wysłania do niego sygnału *SIGCONT*. W stanie tym nie jest możliwe wykonywanie ruchów ręcznych. Dostępne jest w stanie tym polecenie *RESUME_H*.
7. **Stan uczenia robota** *TEACH_IN_STATE* — w stan ten proces UI przechodzi po odebraniu od procesu ECP żądania uczenia robota. W stanie tym pojawia się okienko wykonywania ruchów ręcznych w wybranym układzie współrzędnych i zapamiętywaniu nauczonych pozycji i przesyłaniu ich procesowi ECP. Do stanu *TEACH_IN_STATE* proces może przejść tylko ze stanu *MP_RUNNING*.

W każdym z powyższych stanów procesu UI można wykonać polecenie HELP w celu uzyskania potrzebnych informacji, zgodnych z kontekstem aktualnie wykonywanej operacji.

Rozdział 9

Proces SRP

9.1. Struktura i funkcje System Response Process

Proces SRP jest powoływany do życia przez proces UI. Zostaje on uruchomiony w tym samym węźle co proces UI oraz zarejestrowany pod globalną nazwą np.: */IRP6/SRP*. Proces SRP działa do momentu zakończenia działania procesu UI.

9.1.1. Funkcje realizowane przez proces System Response Process

Proces SRP spełnia następujące funkcje:

- odbiera komunikaty od wszystkich procesów, które informują użytkownika o wystąpieniu sytuacji szczególnej (zmiana stanu systemu, informacje o błędach)
- formatuje i wyświetla komunikaty na ekranie monitora
- przechowuje w buforze nadchodzące komunikaty, tak aby można było śledzić zachowanie systemu od momentu jego uruchomienia.
- dekoduje numery błędów i przetwarza je na odpowiednie komunikaty

9.1.2. Algorytm procesu System Response Process

Proces SRP działa jako *server* wypisujący komunikaty na ekranie monitora. W stanie zawieszenia (RECEIVE BLOCKED) oczekuje na nadejście komunikatu. Dla odróżnienia komunikaty o stanie systemu wyświetlane są w innym kolorze niż informacje o błędach. W przypadku odebrania komunikatu o błędzie fatalnym okienko wyświetlające komunikaty staje się aktywne i jest widoczne jako pierwszoplanowe. Wyświetlane komunikaty mogą pochodzić od procesów EDP, ECP, MP oraz innych procesów stworzonych przez użytkownika.

Struktura komunikatów przesyłanych do SRP ma postać:

```
typedef struct {
  word16 code;           // Kod: SRP_MESSAGE lub SRP_ERROR
  word16 status;         // Status błędu (OK, EDP_FATAL_ERROR,...)
  union {
    unsigned word32 error_code;     // Szczegółowy kod błędu
    word08 message[128];            // Bufor na informację tekstową
```

```
    struct {
      unsigned  word32 error_code0; // Szczegółowy kod błędu - 0
      unsigned  word32 error_code1; // Szczegółowy kod błędu - 1
    } err2;
    struct {
      unsigned word32 error_code;   // Szczegółowy kod błędu
      word08 message[128];          // Bufor na informację tekstową
    } err_and_msg;
  } err_msg;
} srp_package;
```

Znaczenie poszczególnych pól w unii struktur przesyłanych do SRP jest następujące:

`code`	umożliwia rozróżnienie typu komunikatu: *SRP_MESSAGE* – komunikat o stanie procesu, który go wysyła lub inny komunikat np. o przebiegu wykonania zadania użytkowego *SRP_ERROR* – komunikat o błędzie; proces SRP wyświetla taki komunikat w innym kolorze (fatalne - czerwonym, niefatalne- niebieskim) niż zwykłe informacje;
`status`	definiuje klasę błędu
`error_code`, `error_code0`, `error_code1`	pola zawierają szczegółowy kod błędu danej klasy patrz kody błędów.
`message`	tablica umożliwia przesłanie dodatkowej informacji tekstowej, np. opisu zaistniałej sytuacji;

Dzięki wprowadzeniu do struktury komunikatu pola *message* użytkownik może umieszczać precyzyjny opis poszczególnych błędów. Rozwiązanie to uniezależnia proces SRP od ewentualnych zmian dokonywanych w innych procesach sterownika.

Funkcja umożliwia przeglądanie komunikatów, które wcześniej nadeszły. Realizowane jest to poprzez przewijanie zawartości okna, w którym są one zapisywane. Do realizacji przewijania wykorzystano standardowe właściwości okien w środowisku QNX Windows zdefiniowanych jako „przewijalne". Sama aplikacja ani nie przewija danego okna, ani nie buforuje wyświetlanych w nim danych. Czyni to odpowiedni serwer QNX Windows odpowiedzialny za obsługę okien.

9.2. Obsługa System Response Process

Po uruchomieniu procesu SRP dzieli on ekran monitora razem z procesem UI. Okno procesu SRP może zostać w dowolnym momencie przeskalowane (np. do rozmiarów ekranu w celu powiększenia liczby jednorazowo oglądanych komunikatów), przesunięte czy zmniejszone do postaci ikony. Obsługa procesu SRP sprowadza się do przewijania zawartości okna za pomocą suwaka znajdującego się w prawej, pionowej ramce okna.

9.3. Komunikaty o błędach i stanie systemu

Klasy błędów:

```
NEW_MESSAGE       0x1
SYSTEM_ERROR      0x2
FATAL_ERROR       0x3
NON_FATAL_ERROR   0x4
```

Błędy szczegółowe generowane przez ECP i MP:

```
INVALID_MP_COMMAND            0x1
INVALID_POSE_SPECIFICATION    0x2
INVALID_RMODEL_TYPE           0x3
INVALID_ECP_COMMAND           0x4
INVALID_EDP_REPLY             0x5
ECP_ERRORS                    0x6
INVALID_COMMAND_TO_EDP        0x7
ECP_UNIDENTIFIED_ERROR        0x8
MP_UNIDENTIFIED_ERROR         0x9
EDP_ERROR                     0xA
NON_EXISTENT_DIRECTORY        0xB
NON_EXISTENT_FILE             0xC
READ_FILE_ERROR               0xD
NON_TRAJECTORY_FILE           0xE
NON_COMPATIBLE_LISTS          0xF
ECP_STOP_ACCEPTED            0x10
MAX_ACCELERATION_EXCEEDED    0x11
MAX_VELOCITY_EXCEEDED        0x12
```

Kody błędów generowanych w EDP:

```
INVALID_INSTRUCTION_TYPE        0x01000000
INVALID_REPLY_TYPE              0x02000000
INVALID_SET_RMODEL_TYPE         0x03000000
INVALID_GET_RMODEL_TYPE         0x04000000
ERROR_IN_RMODEL_REQUEST         0x05000000
INVALID_HOMOGENEOUS_MATRIX      0x06000000
QUERY_EXPECTED                  0x10000000
QUERY_NOT_EXPECTED              0x11000000
NO_VALID_END_EFFECTOR_POSE      0x12000000
INVALID_MOTION_TYPE             0x13000000
INVALID_MOTION_PARAMETERS       0x14000000
INVALID_SET_END_EFFECTOR_TYPE   0x15000000
INVALID_GET_END_EFFECTOR_TYPE   0x16000000
STRANGE_GET_ARM_REQUEST         0x17000000
```

Przekroczenie zakresów ruchu wałów silników:

```
BEYOND_UPPER_LIMIT_AXIS_0   0x21000000
BEYOND_UPPER_LIMIT_AXIS_1   0x22000000
BEYOND_UPPER_LIMIT_AXIS_2   0x23000000
BEYOND_LOWER_LIMIT_AXIS_0   0x24000000
BEYOND_LOWER_LIMIT_AXIS_1   0x25000000
BEYOND_LOWER_LIMIT_AXIS_2   0x26000000
```

Błędy wykryte przy synchronizacji serwomechanizmów:

```
SYNCHRO_SWITCH_EXPECTED  0x00140000
SYNCHRO_ERROR            0x00180000
SYNCHRO_DELAY_ERROR      0x001C0000
```

Błędy wykrywane przez SERVO_GROUP:

SERVO_ERROR_IN_PASSIVE_LOOP	0x00040000
UNIDENTIFIED_SERVO_COMMAND	0x00080000
SERVO_ERROR_IN_PHASE_1	0x000C0000
SERVO_ERROR_IN_PHASE_2	0x00100000

Błędy przekroczenia zakresu we współrzędnych wewnętrznych:

Rozdział 10

Pomiary w systemie MRROC++

10.1. Rejestracja w czasie rzeczywistym wielkości opisujących ruch robota

Zastosowanie robota IRP6 do złożonych czynności takich jak precyzyjne frezowanie, szlifowanie stawia wysokie wymagania dokładnościowe i funkcjonalne zarówno części mechanicznej robota jak też układowi sterowania. W odniesieniu do części sterującej wiąże się to m.in. ze spełnieniem następujących wymogów:

- dużą sztywnością serwomechanizów,
- brakiem przeregulowań i zerowaniem (minimalizacją) uchybów statycznych,
- odpornością na zakłócenia (np. zmienne obciążenie, tarcie), szybkie zmiany prędkości i przyspieszeń zadanych,
- dokładnym odtwarzaniem zadanych trajektorii.

Doświadczalne sprawdzenie jakości działania układu sterowania wymaga bieżącego pomiaru wielkości opisujących ruch manipulatora robota. Ze względu na dokładność wykonania zadania najbardziej adekwatne i wiarygodne byłyby pomiary aktualnej pozycji (tj. położenia i orientacji) narzędzia w przestrzeni roboczej. Jest to szczególnie istotne, gdy programowanie robota metodą uczenia jest uciążliwe lub wręcz niemożliwe. Ma to miejsce np. przy frezowaniu skomplikowanych powierzchni/kształtów. Przy tego typu zadaniach trajektoria ruchu narzędzia jest zazwyczaj generowana automatycznie w czasie planowania procesu technologicznego w systemie CAD/CAM i jest często reprezentowana jako analityczna funkcja czasu. Zbyt mała dokładność odtwarzania takiej trajektorii uniemożliwia wykonanie zadania. Wykonanie pomiarów w przestrzeni roboczej (zadaniowej) wiąże się jednak z koniecznością stosowania złożonych czujników pomiarowych. Stosuje się je zazwyczaj w warunkach laboratoryjnych. Dlatego też, o ile to możliwe, należy starać się wykorzystać informacje z czujników w które wyposażony jest robot. Dla oceny jakości działania serwomechanizów osi robota może być to całkowicie wystarczająca informacja. Na jej podstawie można określić sposób ewentualnej modyfikacji struktury i/lub doboru parametrów regulatorów.

10.2. Warunki wykonania pomiarów

Pomiary wielkości opisujących ruch robota mogą być wykonywane w różnych warunkach i dla różnych celów. Na przykład przy identyfikacji modelu robota podstawowym celem jest rejestracja odpowiedzi obiektu na zadane wymuszenie. Pomiary aktualnych sygnałów wyjściowych takich jak położenie, prędkość, prąd są niezbędne dla działania układu sterowania z sprzężeniem zwrotnym. Zazwyczaj w typowych sterownikach przemysłowych zmierzone wartości chwilowe są bezpośrednio wykorzystywane (wewnętrz sterownika) do obliczania sterowań i nie ma możliwości ich rejestracji lub taka rejestracja zakłóca ruch robota (tak jest np. w przypadku sterownika przemysłowego robota IRp-6). Dane pomiarowe zebrane podczas normalnej pracy są szczególnie przydatne, ponieważ opisują zachowanie się całego układu (manipulatora robota i układu sterowania) w rzeczywistych warunkach wykonania zadania. W tym przypadku szczególnie istotne jest aby proces pomiarowy nie zakłócał przebiegu realiazcji zadania. Dlatego też przyjęto następujące założenia dotyczące warunków prowadzenia pomiarów:

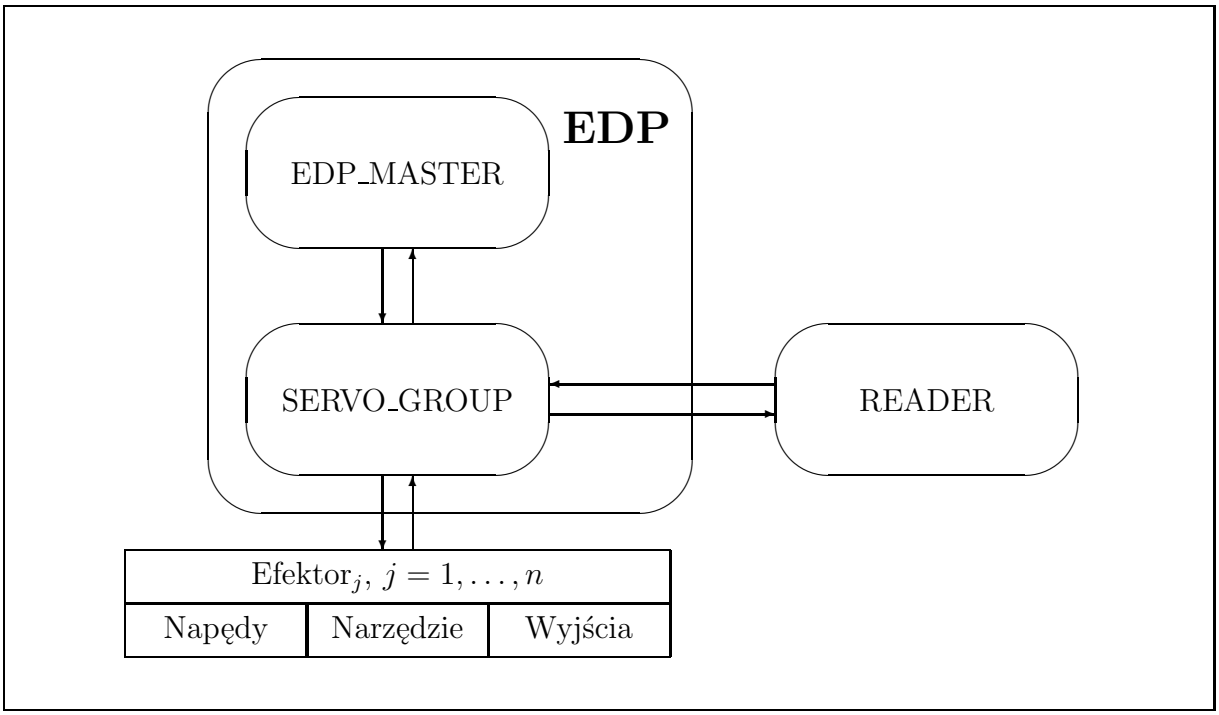
- Pomiary nie powinny zakłócać procesu regulacji, a zatem liczba obliczeń związanych z rejstracją danych pomiarowych powinna być jak najmniejsza.
- Nie wykonuje się dodatkowych odczytów wyjściowych rejestrów sprzętowych, korzysta się wyłącznie z danych używanych w procesie regulacji.
- Dane pomiarowe rejestrowane są na bieżąco, zaś okres próbkowania może być równy pojedynczemu przedziałowi sterowania lub może być jego wielokrotnością.
- Mogą być rejestrowane zarówno wielkości zadane tj. położenie/przyrost, sterowanie (wypełnienie przebiegu PWM), jak też zmierzone rzeczywiste położenia/przyrosty i rzeczywisty prąd wirnika silnika napędowego.
- Chwile rozpoczęcia i zakończenia pomiarów są określane przez operatora.
- Opertor może wybierać osie, dla których będą rejestrowane pomiary.
- Liczba pomiarów jest zadawana przez operatora i ograniczona od góry jedynie pojemnością wolnej pamięci operacyjnej.

W celu umożliwienia wykonywania i rejestracji pomiarów w czasie rzeczywistym zmodyfikowano proces SERVO_GROUP oraz napisano dodatkowy proces READER współpracujący z SERVO_GROUP.

10.3. Implementacja procesu pomiarowego READER

Proces READER jest powoływany automatycznie przez SERVO_GROUP i tylko z nim komunikuje się w chwilach rozpoczęcia i zakończenia cyklu pomiarowego. Na rysunku 10.1 przedstawiono umiejscowienie procesu READER w strukturze sterownika MRROC++. Po uruchomieniu procesu READER wyświetlane są pytania i zachęty dotyczące organizacji pomiarów. Operator może zadać liczbę pomiarów oraz zdecydowac, dla których osi będą rejestrowane dane. Określa również chwilę rozpoczęcia rejestracji pomiarów, wówczas z procesu READER wysyłany jest sygnał SIGUSR1 informujący SERVO_GROUP o konieczności odbioru wiadomości z poleceniem operatora. Następnie nawiązywane jest spotkanie między tymi procesami i w jego trakcie przekazywana jest do SERVO_GROUP wiadomość o następującej strukturze:

```
struct reader_command {
```



Rys. 10.1: Pomiar

```
  READER_COMMAND command;          // rodzaj polecenia dla SERVO_GROUP
  BYTE measured_axes;              // mierzone osie
  unsigned long int nr_of_samples; // liczba pomiarów
};
```

Jeśli polecenie zostało poprawnie odebrane do procesu READER przesyłana jest odpowiedź z potwierdzeniem. Po odebraniu pierwszego polecenia (BEGIN_MEAS) proces SERVO_GROUP rozpoczyna rejestrację danych pomiarowych. Tworzony jest bufor cykliczny o zadanym rozmiarze, do którego zapisywane są dane pomiarowe. Bufor ten jest tablicą struktur postaci:

```
struct io_data {  // Struktura z danymi pomiarowymi
 float desired_inc[NUMBER_OF_SERVOS];  // zadane przyrosty położenia
 float current_inc[NUMBER_OF_SERVOS];  // aktualne przyrosty położenia
 short int set_value[NUMBER_OF_SERVOS];// zadane wartości PWM (sterowanie)
};
```

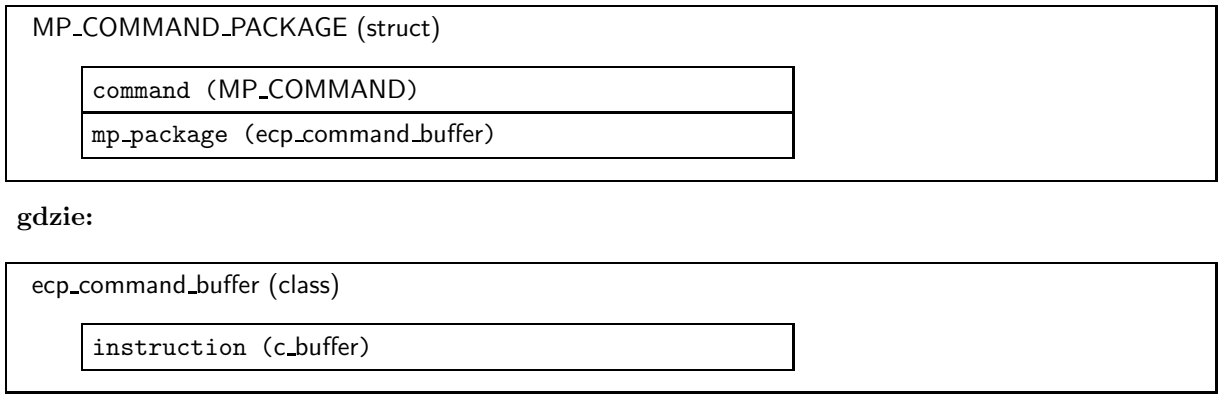
Rejestracja danych trwa do chwili wysłania z procesu READER polecenia zakończenia pomiarów i przesłania danych (GET_DATA). Proces READER działa z niskim priorytetem, jedynie w fazach komunikacji z SERVO_GROUP priorytet procesu jest zwiększany tak, aby czas poświęcony na wymianę danych między procesami był możliwie najkrótszy. Jest to istotne dla poprawnego (płynnego) działania regulatorów. Po odebraniu danych przez proces READER możliwe jest ich zapisanie na dysk do pliku tekstowego. Jeśli operator wybierze opcję zapisu, musi podać nazwę pliku, do którego zapisane będą pomiary. Ponieważ priorytet procesu READER jest niski, zatem zapisywanie odbywa się w chwilach, gdy inne procesy wchodzące w skład sterownika zwalniają procesor. Operacja zapisu na dysk nie zakłóca zatem działania sterownika. Po zakończeniu zapisu można rozpocząć kolejny cykl rejestracji danych.

Rozdział 11

Komunikacja między procesami: EDP, ECP, VSP, MP, UI i SRP

11.1. Komunikacja między procesami: ECP i MP

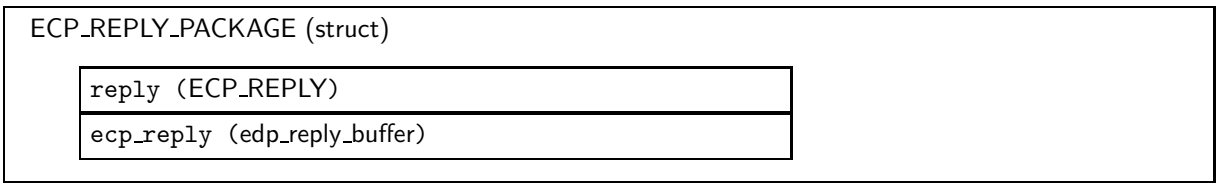
Na rysunkach 11.1 oraz 11.2 przedstawiono format pakietów komunikacyjnych przesyłanych między procesami MP i ECP.



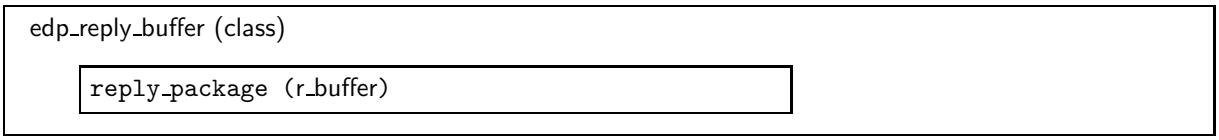
Rys. 11.1: Pakiet komunikacyjny przesyłany z MP do ECP.
Konwencja: `nazwa_składowej` (jej_typ)

11.2. Komunikacja wewnątrz EDP (EDP_MASTER z SERVO_GROUP)

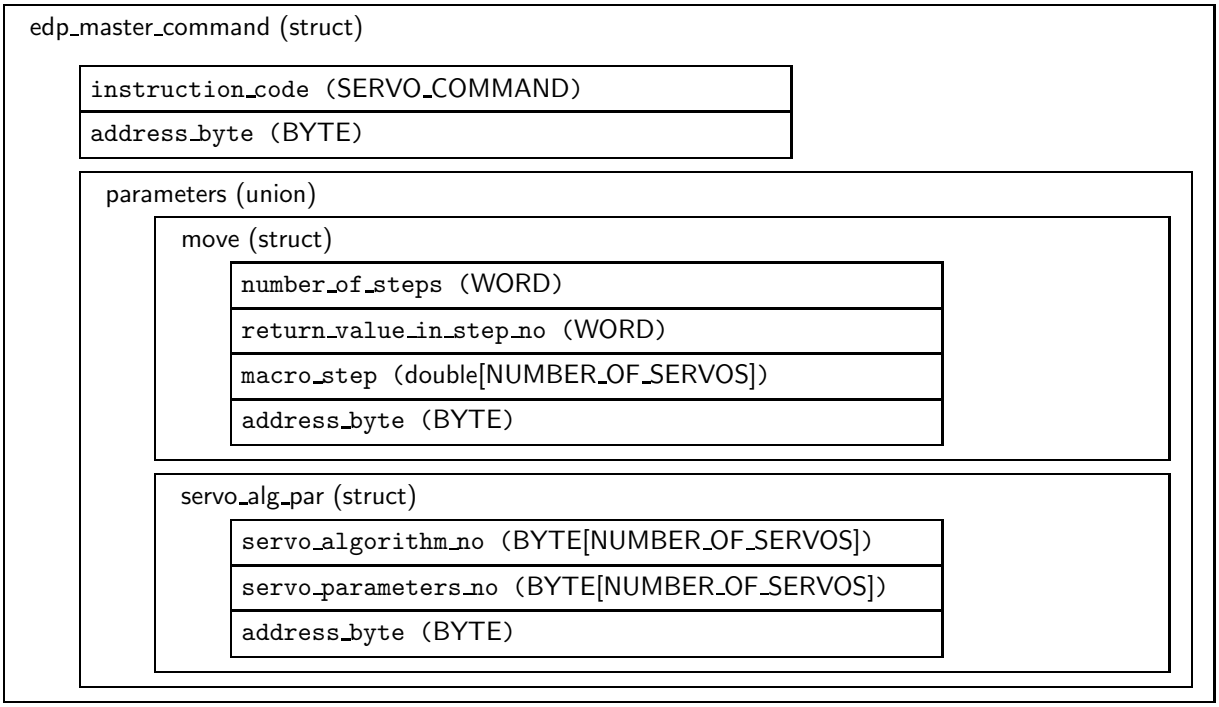
Na rysunkach 11.3 oraz 11.4 przedstawiono format pakietów komunikacyjnych przesyłanych między procesami EDP_MASTER i SERVO_GROUP.



gdzie:



Rys. 11.2: Pakiet komunikacyjny przesyłany z ECP do MP. *Konwencja:* `nazwa_składowej (jej_typ)`



Rys. 11.3: Pakiet komunikacyjny między EDP_MASTER a SERVO_GROUP. *Konwencja:* `nazwa_składowej (jej_typ)`

11.3. Komunikacja innych procesów z procesem SRP

Na rysunku 11.5 przedstawiono format pakietu komunikacyjnego przesyłanego między dowolnymi innymi procesami a procesem SRP.

11.4. Komunikacja procesu UI z procesem MP

Proces UI kontaktuje się z procesem MP poprzez pośredników (*ang.* `proxy`), czyli stałe komunikaty przyporządkowane poszczególnym poleceniom operatora. Poleceniami tymi są: *START*, *STOP*, *PAUSE*, *RESUME*.

servo_group_reply (struct)

error (edp_error)
position (double[NUMBER_OF_SERVOS])
PWM_value (word16[NUMBER_OF_SERVOS])
current (word16[NUMBER_OF_SERVOS])
algorithm_no (BYTE[NUMBER_OF_SERVOS])
algorithm_parameters (BYTE[NUMBER_OF_SERVOS])

Rys. 11.4: Pakiet komunikacyjny między SERVO_GROUP a EDP_MASTER. *Konwencja:* nazwa_składowej (jej_typ)

srp_package (struct)

process_type (word16)
message_type (word16)
process_name (word08[NAME_LENGTH])
description (word08[TEXT_LENGTH])

Rys. 11.5: Pakiet przesyłany do SRP z dowolnego procesu. *Konwencja:* nazwa_składowej (jej_typ)

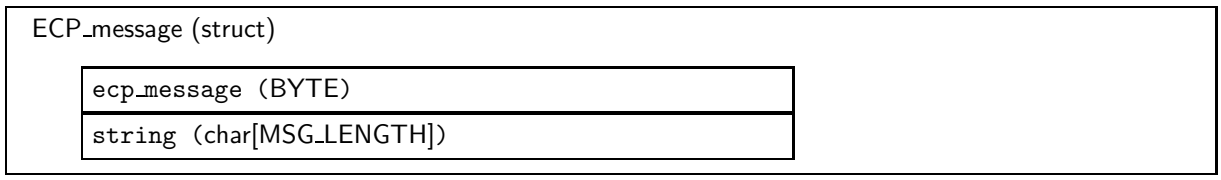
11.5. Komunikacja procesu UI z procesem ECP

Na rysunkach 11.6 oraz 11.7 przedstawiono format pakietów komunikacyjnych przesyłanych między procesami UI i ECP. Komunikacja ta ma miejsce w przypadku uczenia raobota trajektorii, którą ma odtwarzać oraz w przypadku, gdy wymagana jest specjalna interwencja operatora w tok wykonania programu użytkowego dla pojedynczego robota.

UI_reply (struct)

reply (BYTE)
integer_number (word16)
double_number (double)
coordinates (double[NUMBER_OF_SERVOS])
path (char[80])
filename (char[20])

Rys. 11.6: Pakiet komunikacyjny przesyłany z UI do ECP. *Konwencja:* nazwa_składowej (jej_typ)



Rys. 11.7: Pakiet komunikacyjny przesyłany z ECP do UI.
Konwencja: `nazwa_składowej (jej_typ)`

11.6. Komunikacja procesów ECP i VSP

Na rysunkach 11.8 oraz 11.9 przedstawiono wypełnienie pakietów komunikacyjnych przesyłanych między procesami VSP i ECP. Komunikacja ta dotyczy współpracy interaktywnej procesu ECP z VSP.

Przesyłana do VSP struktura danych składa się z pola zawierającego kod instrukcji `instruction_code`, która jest typu `VSP_COMMAND` zdefiniowanego jako typ `enum` obejmujący sekwencję dyrektyw: `VSP_INIT, VSP_GET_READING, VSP_TERMINATE`. Następne pole – `parameters` przeznaczone jest na parametr powyższej instrukcji (np. wymagany przez czujnik). Pole to jest unią `vsp_parameters` zawierajaca liczbę całkowitą albo zmiennoprzecinkową.

Przesyłana do ECP struktura danych – rys. 11.8, składa się z pola zawierającego odpowiedź procesu VSP – `vsp_report`, która jest typu `unsigned_int` i może zawierać kod: `VSP_reply_OK`, `INVALID_ECP2VSP_COMMAND`, `INVALID_VSP_READING_TYPE`. Następne pole – `reading_type` przeznaczone jest na kod typu odczytanych danych (zgodnie z definicją programisty: `MY_DEFINITION`). Kolejne pole – `vsp_reply_buffer` przeznaczone jest na dane z czujników. Pole to jest unią typu `vsp_reply`, w której definiuje się pakiety danych z czujników.

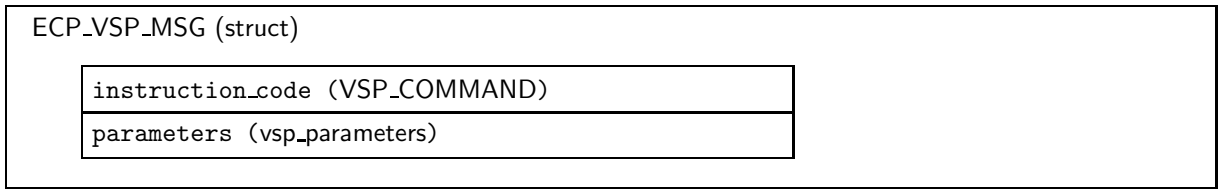
Przy pracy interaktywnej proces ECP inicjując odpowiedni proces VSP (co oznacza też przesłanie wiadomości `VSPinit`), odbiera dane z odpowiednim raportem, kodem danych i danymi. Tak samo się dzieje po przesłaniu żądania odczytu `VSP_GET_READING`. W poprawnej sytuacji VSP odpowiada wiadomością `VSP_reply_OK`. W sytuacji, gdy z ECP odebrano inny rodzaj wiadomości niż przewidziane (błąd), proces VSP odpowiada wiadomością `INVALID_ECP2VSP_COMMAND`.

W przykładowej realizacji procesu VSP odczytującego dane z czujnika klawiszowego struktura przesyłanych danych obejmuje osiem zmiennych typu `int` albo jedna zmienna `insigned int` zawierającą dane skumulowane. W drugim przypadku każdy bit odpowiada stanowi jednego klawisza. Dla pakietu ośmiu danych zdefiniowano typ strukturalny `int_8`. Dane z czujnika klawiszowego są unią `all_touch` zawierającą strukture `binary_sensor` typu `u_8` oraz dane skumulowane `all_bin` typu `unsigned int`. Zmienna `vsp_reply_buffer` jest unią typu `vsp_reply` zawierajacą unie (struktury) odpowiadające różnym czujnikom (różnym zbiorom czujników) – przykładowo zawiera ona unię typu `all_touch` dla czujnika klawiszowego i unię typu `force_sensors` dla czujników siły.

11.7. Komunikacja procesów ECP i EDP

Na rysunkach 11.10 i 11.11 przedstawiono format poleceń przesyłanych z procesu ECP do procesu EDP, a konkretnie do podprocesu EDP_MASTER stanowiącego rodzaj interfejsu *server'a* z *klientem*, czyli procesem ECP. Poniższą strukturę danych można traktować jako wielowariantowy pakiet danych przekazywany między komunikującymi się procesami.

Rys. 11.8: Pakiet komunikacyjny pomiędzy VSP a ECP. *Konwencja:* `nazwa_składowej (jej_typ)`



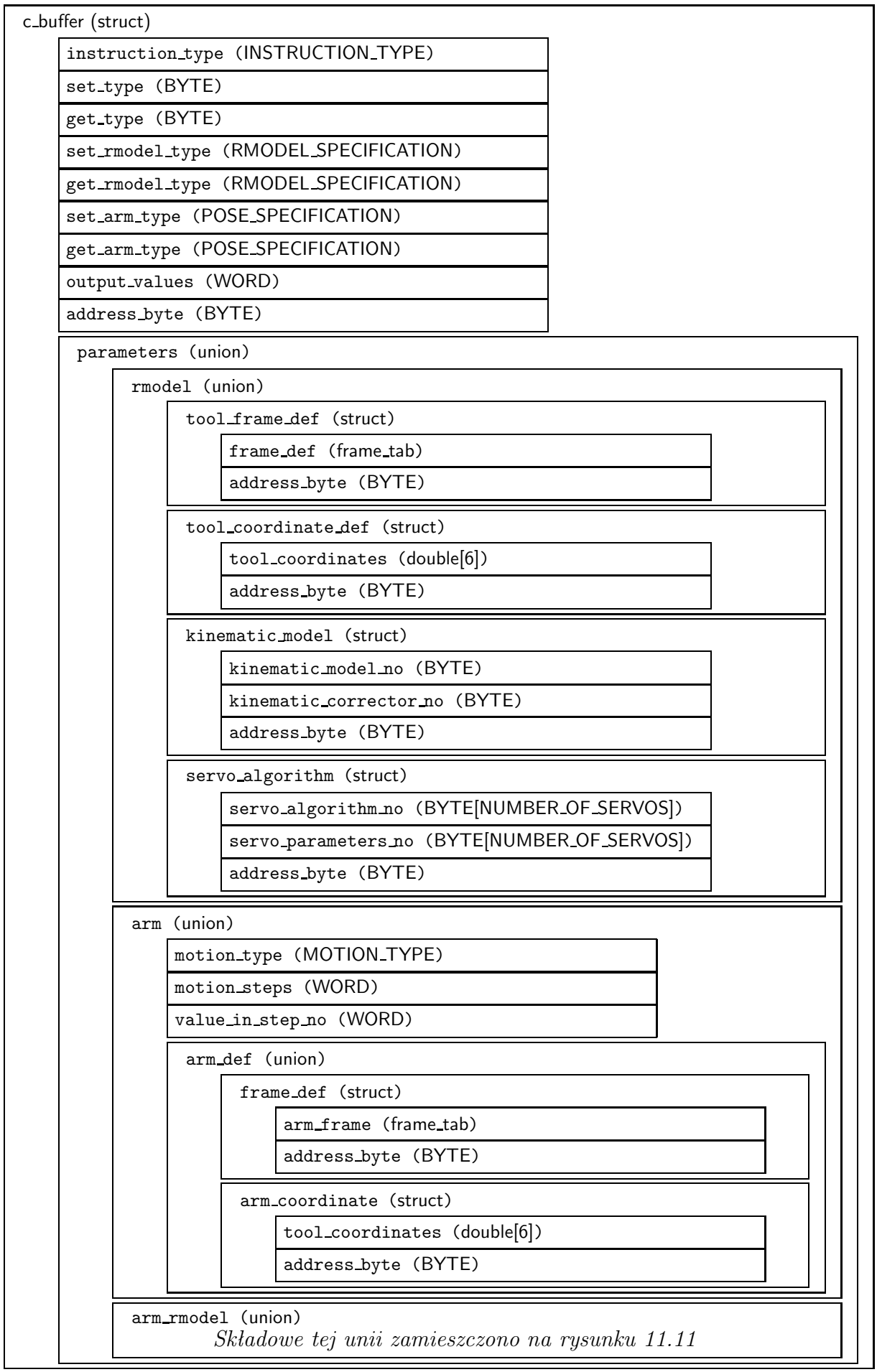
Rys. 11.9: Pakiet komunikacyjny pomiędzy ECP a VSP. *Konwencja:* `nazwa_składowej (jej_typ)`

Który wariant tego pakietu jest przesyłany, zależy od rodzaju polecenia (SYNCHRO, SET, GET, SET_GET, QUERY) oraz konkretnej kombinacji niezbędnych parametrów. Wszystkich możliwych kombinacji parametrów jest bardzo dużo, więc dostosowanie formatu pakietu do każdej kombinacji oddzielnie nie wchodzi w rachubę – bardzo by to utrudniło przyszłe korzystanie z niego. Dużo prościej byłoby skonstruować pakiet o stałym formacie, ale to prowadziłoby do przesyłania dużego nadmiaru bajtów dla rozkazów o niewielu, i to krótkich, parametrach (stosunek długości rozkazu najdłuższego do najkrótszego przekracza 100). Wybrano więc wariant pośredni – pakiet może przyjmować jedną z dziewięciu postaci.

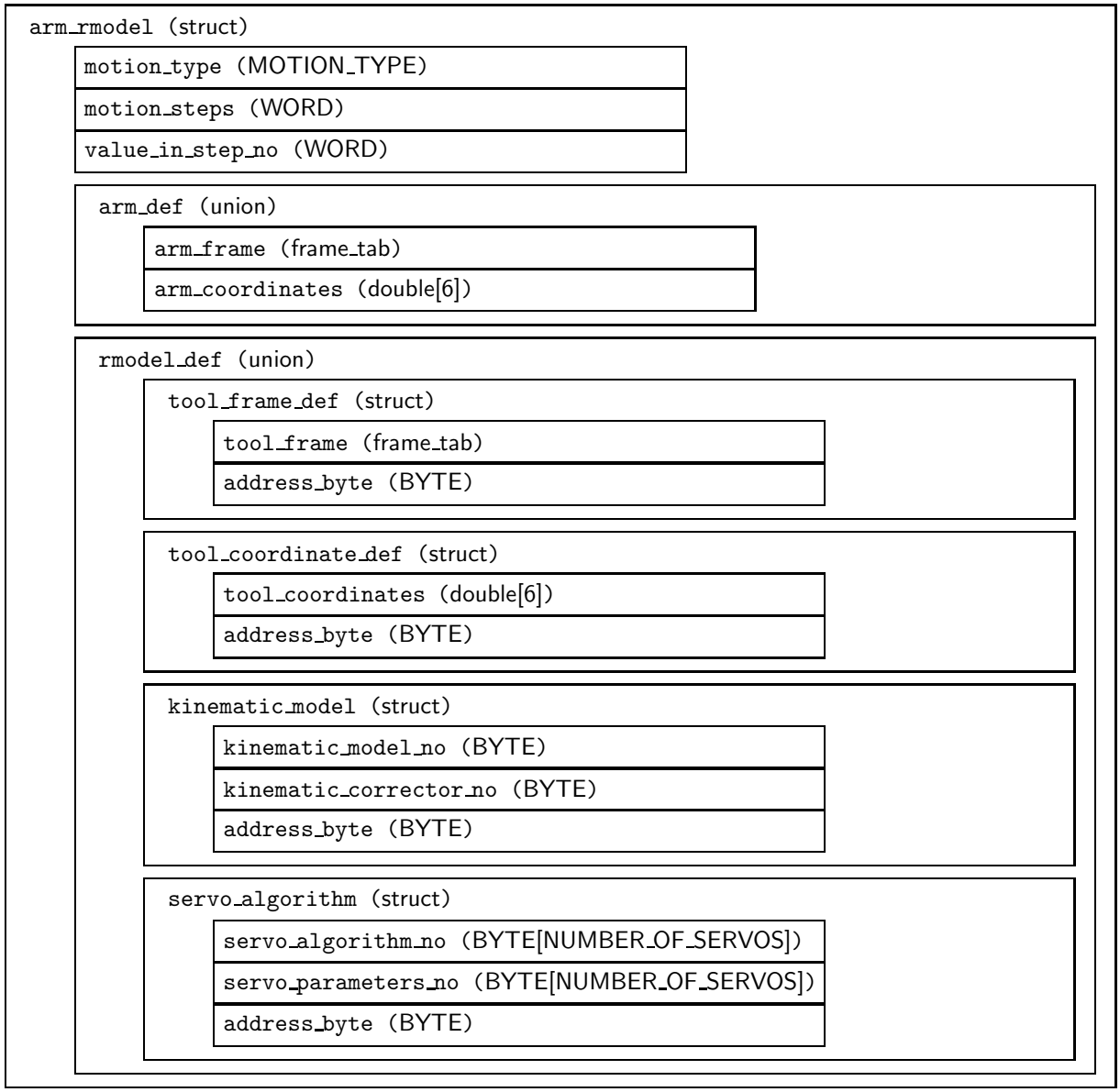
Z tych samych powodów co powyżej, format pakietu przesyłanego zwrotnie jest również wielowariantowy. Definicja formatu tego pakietu została przedstawiona na rys. 11.12 i 11.13.

Definicje typów użytych do określenia składowych pakietów są następujące.

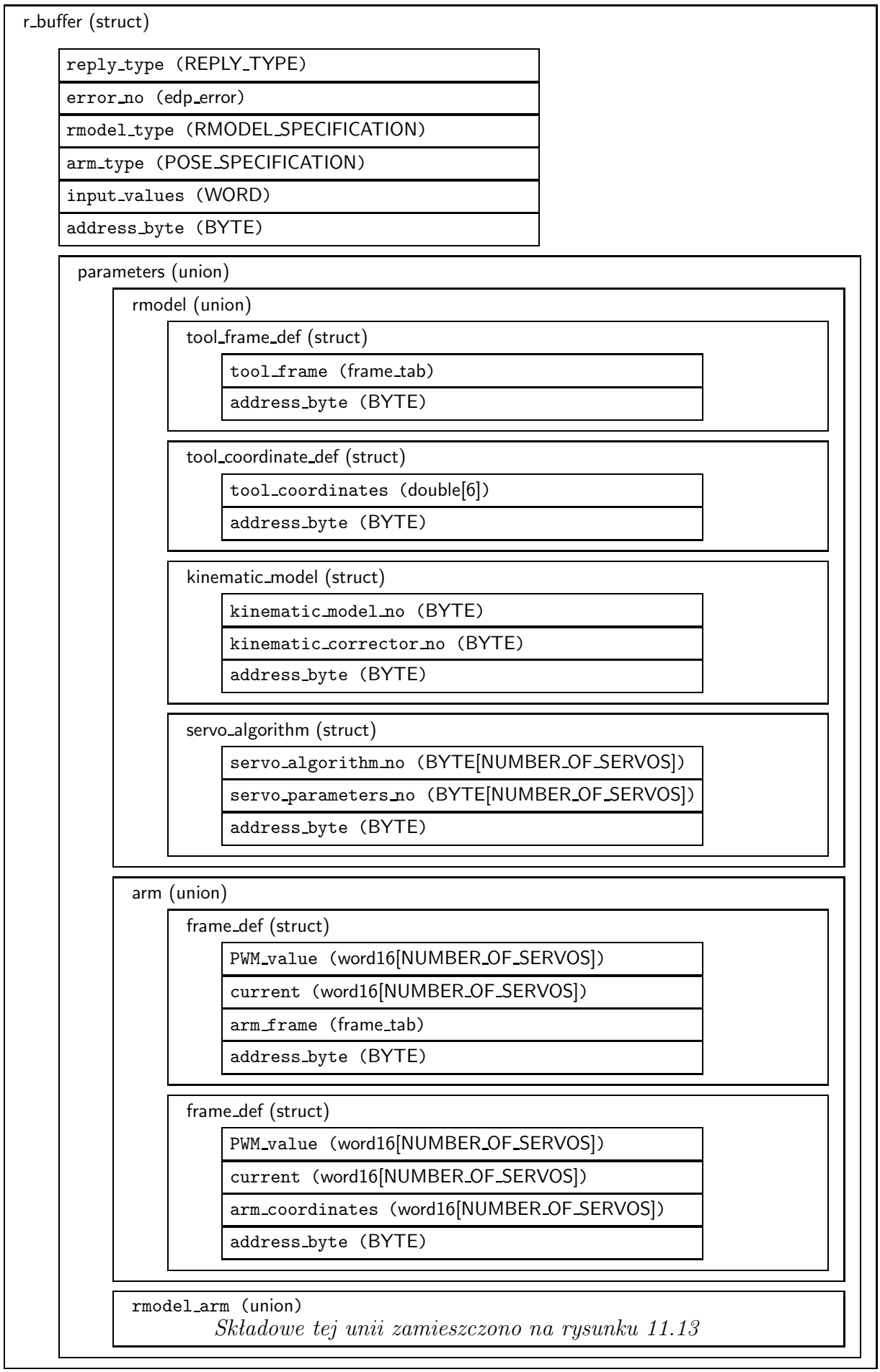
```
enum INSTRUCTION_TYPE     { INVALID, SET, GET, SET_GET, SYNCHRO, QUERY };

enum RMODEL_SPECIFICATION { INVALID_RMODEL, TOOL_FRAME, TOOL_XYZ_ANGLE_AXIS,
                            TOOL_XYZ_EULER_ZYZ, ARM_KINEMATIC_MODEL, SERVO_ALGORITHM };

enum POSE_SPECIFICATION   { INVALID_END_EFFECTOR, FRAME, XYZ_ANGLE_AXIS,
                            XYZ_EULER_ZYZ, JOINT, MOTOR };

enum MOTION_TYPE          { ABSOLUTE, RELATIVE };

enum REPLY_TYPE  { ERROR, ACKNOWLEDGE, SYNCHRO_OK, ARM, RMODEL, INPUTS, ARM_RMODEL,
                   ARM_INPUTS, RMODEL_INPUTS, ARM_RMODEL_INPUTS };
```



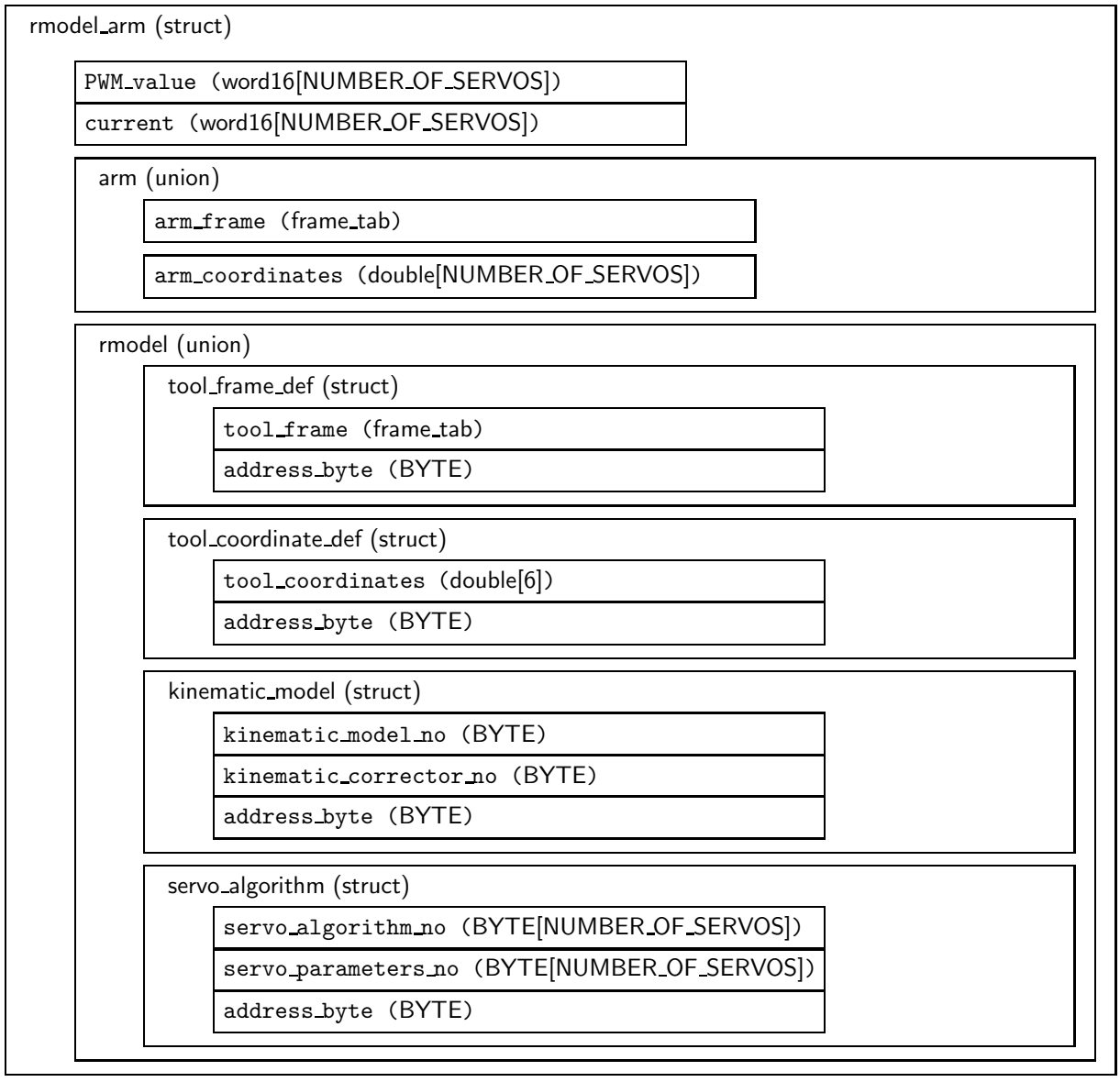
Rys. 11.10: Pakiet komunikacyjny przesyłany z ECP do EDP.
Konwencja: `nazwa_składowej` (jej_typ)



Rys. 11.11: Fragment pakietu komunikacyjnego przesyłanego z ECP do EDP. *Konwencja:* `nazwa_składowej (jej_typ)`



Rys. 11.12: Pakiet komunikacyjny przesyłany z EDP do ECP. *Konwencja:* `nazwa_składowej` (jej_typ)



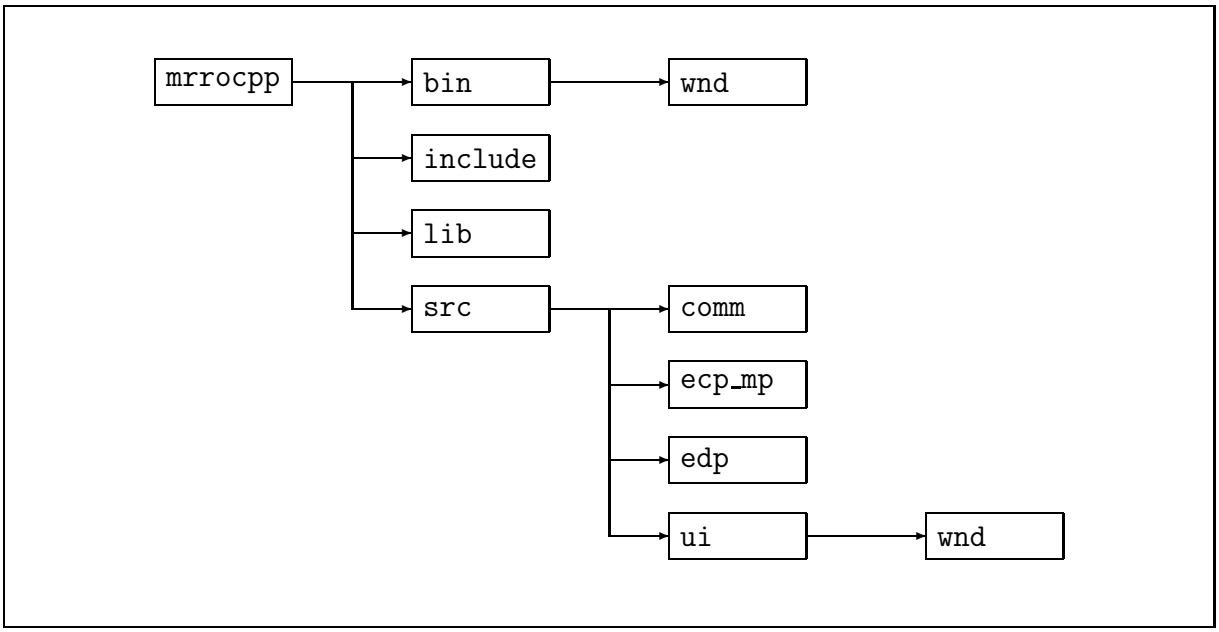
Rys. 11.13: Fragment pakietu komunikacyjnego przesyłanego z EDP do ECP. *Konwencja:* `nazwa_składowej` (jej_typ)

Rozdział 12

Programowanie systemu

12.1. Struktura katalogów oraz zawartość poszczególnych plików

Struktura katalogów pakietu MRROC++ jest następująca: Katalog główny o nazwie `mrrocpp` może znajdować w dowolnym miejscu w struktury katalogów systemu QNX. Jednakże, ze względów porządkowych powinien ten katalog znajdować się w katalogu systemowym `home`, a zatem ścieżka dostępu powinna być `/home/mrrocpp`. Struktura podkatalogów `mrrocpp` jest przedstawiona na rys.12.1:



Rys. 12.1: Struktura katalogów

Katalog:

`bin` – zawiera pliki wykonywalne: `ui`, `mp_m`, `ecp_m`, `edp_m`, `servo` konieczne do uruchomienia sterownika MRROC++

`wnd` – zawiera pliki *.pict *.wnd będące elementami składowymi okienkowego interfejsu użytkownika (okienka, menu, ikony) ładowane gdy wybierana jest odpowiednia opcja z menu

`include` – zawiera wszystkie pliki nagłówkowe (*.h) wchodzące w skład MRROC++

`lib` – zawiera biblioteki funkcji wykorzystywanych do pisania sterowników

`src` – zawiera pliki źródłowe (w języku `C` i `C++`) z programami kodów źródłowych procesów składowych MRROC++:

- `comm` – zawiera pliki źródłowe funkcji realizujących komunikację pomiędzy procesami składowymi sterownika MRROC++
- `ecp_mp` – zawiera pliki źródłowe funkcji służących do pisania ECP oraz MP `ecp.cc` i `mp.cc`, a także przykładowe funkcje `main` procesów ECP i MP `ecp_m.cc` i `mp_m.cc` oraz kilka przykładowych programów (funkcji `main`) ECP i MP wykonujących konkretne zadania użytkowe (patrz **??**)
- `edp` – zawiera pliki źródłowe funkcji służących do tworzenia procesów EDP (procesy składowe EDP_MASTER i SERVO_GROUP) (`edp.cc servo_gr.cc`, `hi_rydz.cc`, `edp_m.cc`, `servo_ms.cc`) *driver'a* dla robota IRp-6
- `ui` – zawiera pliki źródłowe okienkowego interfejsu użytkownika UI oraz programu SRP działających w środowisku QNX Windows
  - `wnd` – zawiera pliki *.pict *.wnd będące elementami składowymi okienkowego interfejsu użytkownika (okienka, menu, ikony)

12.2. Listy

Ponieważ przy konstruowaniu sterownika korzysta się z wielu różnorakich list (n.p. czujników, robotów, serwomechanizmów), a operacje przeprowadzane na każdej z list są identyczne (tworzenie listy, wstawianie elementu, usuwanie elementu, etc.), więc, dla redukcji kodu oraz zwiększenia jego niezawodności, celowym jest utworzenie szablonu listy, a następnie stworzenie list konkretnych typów na podstawie tego szablonu. Szablon listy ma następującą postać.

```
template <class E>
class list {
 public:
   E* E_ptr;              // wskaźnik elementu listy
   list<E>* next;         // wskaźnik następnego węzła listy

  list (E* el_ptr) { E_ptr=el_ptr; next = NULL; }; // konstruktor
  ~list (void) { if(next) delete next; };          // destruktor
  // dodanie elementu na początek listy
  void add_list_element (E* el_ptr, list<E>*& head)
                        { list<E>* prev=head;
                          head = new list<E>(el_ptr);
                          head->next = prev; };
  void insert_list_element (E* el_ptr);
  void delete_next_list_element (void);
  void delete_nth_list_element (int n);
  int list_length (void);
  void delete_list_elements (void);
}; // end: class list template
```

Deklarując listę odwołujemy się do **obiektu abstrakcyjnego** (czujnika, robota, etc.). Natomiast tworząc listę umieszczamy na niej **obiekty konkretne**. Obiekty klasy konkretnej wywodzi się z klasy abstrakcyjnej. W ten sposób programista tworząc swój program może użyć dowolnych czujników lub robotów korzystając z list obiektów zadeklarowanych przez twórców systemu, którzy nie wiedzieli jakie konkretne obiekty będą mu potrzebne.

Taka konstrukcja listy umożliwia umieszczanie na niej **czujników konkretnych** przekazujących odczyty różnego typu. Czujnik konkretny musi zawierać pole danych odpowiedniego typu przechowujące ostatni odczyt. Klasa `sensor`, dzięki użyciu formy syntaktycznej "`= 0`" po deklaracji metody (funkcji wirtualnej), stanowi klasę abstrakcyjną. Oznacza to, że obiekt tej klasy nie może być użyty w programie, natomiast obiekt klasy pochodnej (czujnik konkretny), w której czyste funkcje wirtualne zostaną zastąpione konkretnymi definicjami funkcji, może być wykorzystywany. Zadaniem użytkownika będzie dostarczenie definicji czujników konkretnych. Definicje te, między innymi, będą określały jak MP zawierający odwołania do czujnika konkretnego ma się kontaktować z adekwatnym procesem VSP oraz w jakiej postaci będą przechowywane odczyty. W ten sposób można zdefiniować procedurę `Move`, stosując `sensor`, natomiast to z jakimi konkretnymi czujnikami będzie ona wykonana będzie zależeć od programisty dostarczającego listę `sensor_list`. Oczywiście czujniki konkretne muszą wywiedzione z `sensor`.

12.3. Sugestie uruchomieniowe

Najprostszy sposób stworzenia nowego sterownika dedykowanego jakiemuś konkretnemu zadaniu polega na skopiowaniu gotowych procesów ECP i MP, które realizują najbardziej podobne zadanie. Następnie należy je tak modyfikować, by powstał sterownik wykonujący nowe zadanie.

W trakcie uruchamiania nowego sterownika nieuniknione są błędy programistyczne. Aby je wykryć i usunąć można korzystać z następujących środków. Dowolnym miejscu programu użytkowego można umieszczać instrukcje `cprintf` powodujące wyświetlanie dowolnych komunikatów oraz wartości zmiennych w oknie powłoki (shell) sytemu operacyjnego. Używając w łańcuchu znakowym stanowiącym format instrukcji `cprintf` znaków "`\A`" możemy w trakcie pracy sterownika uzyskać dźwięk wskazujący na miejsce w programie, w którym została umieszczona taka instrukcja. Wtedy nie zaburzamy pracy innych okienek ich przełączaniem. Inną metodą jest przesłanie wiadomości (message) procesowi SRP. Wtedy on tę wiadomość umieści w okienku komunikatów. Robi się to za pomocą funkcji `msg->message`. Można również wykorzystać przy uruchamianiu debugger `wd`, lecz należy pamiętać o ostrych wymaganiach czasowych, jakie muszą być spełnione przy pracy z rzeczywistym robotem.

Uruchomienie procesów sterownika należy rozpocząć od uruchomienia w katalogu `/home/-mrrocpp/bin` środowiska QNX Windows poleceniem `windows`. Po uruchomieniu QNX Windows trzeba otworzyć okienko terminala `wterm` naciskając prawy przycisk myszki i wybierając z menu odpowiednią opcję. Po ukazaniu się okienka terminala należy uruchomić z linii polecenia (z shell'a) proces UI poleceniem `ui` lub `./ui`. Po uruchomieniu interfejsu okienkowego można korzystać z odpowiednich poleceń dostępnych dla użytkownika zgodnie z opisem przedstawionym w rozdziale 8.

Rozdział 13

Wnioski

System MRROC++ jest jednocześnie biblioteką modułów, z których konstruowany jest sterownik dedykowany zadaniu, przeznaczonemu do wykonania przez robot lub system wielorobotowy, oraz językiem programowania, w którym to zadanie jest wyrażane (oprogramowywane). Istotną cechą przyjętej struktury jest jej otwartość, umożliwiająca dołączenie do systemu dowolnych efektorów i czujników. Teoretyczne uzasadnienie struktury systemu MRROC++ zostało zamieszczone w pierwszej części tego opracowania. U podłoża systemu MRROC++ leży język C++ oraz system operacyjny czasu rzeczywistego QNX. System MRROC++ jest ulepszoną i rozszerzoną wersją systemu MRROC. Ulepszenia głównie wynikają z zastosowania całkowicie innego podejścia do jego implementacji, w tym przypadku podejścia obiektowego zamiast proceduralnego do konstrukcji biblioteki. Implementacja systemu MRROC++ pokazała, że wystarczy jedna, ale wygodnie jest użyć dwóch instrukcji ruchowych (Move i Wait), aby w pełni móc sterować dowolnym systemem wielorobotowym. W systemie MRROC istniało bardzo wiele instrukcji ruchu – czyli procedur realizujących różne typy ruchu (rodzaje trajektorii). Natomiast w systemie MRROC++ zaimplementowano jedynie dwie instrukcje ruchu i mają one postać niezależną od:

- typu i liczby robotów,
- typu i liczby czujników,
- realizowanego ruchu oraz zadania.

Cała zmienność instrukcji, wynikająca z różnorodności wykonywanych ruchów, została skoncentrowana w dwóch parametrach: generatorze trajektorii (dla Move) oraz warunku początkowym (dla Wait). Stałą liczbę parametrów instrukcji ruchowych uzyskano przez zastosowanie list robotów i czujników. Następujące cechy programowania obiektowego są szczególnie przydatne do realizacji systemu MRROC++: polimorfizm i klasy abstrakcji, kapsułkowanie, obsługa wyjątków, wzorce (zastosowano je do tworzenia list). Paradygmat programowania obiektowego może być wykorzystany jedynie w ramach każdego z procesów systemu wieloprocesowego jakim jest MRROC++. Obsługa sytuacji awaryjnych została oddzielona od kodu sterującego normalnym wykonaniem instrukcji i jest realizowana jako obsługa wyjątków.

Sterownik robota IRp-6 został skonstruowany jako urządzenie mogące sterować wieloma efektorami i korzystać z informacji pochodzącej z różnorodnych czujników. Sam robot IRp-6 stanowi jeden z efektorów. Za bezpośrednie sterowanie efektorem odpowiedzialny jest proces Effector Driver Process EDP. Ze względów technicznych proces ten został podzielony na podprocesy. Wydzielono proces koordynujący, czyli EDP_MASTER, oraz proces realizujący regulację, czyli SERVO_GROUP.

W wyniku realizacji pracy uzyskano sterownik o całkowicie otwartej strukturze. Nadaje się on zarówno do badań algorytmów regulacji, wykorzystania różnorodnych czujników w sterowaniu ruchem ramienia jak i do prac nad różnymi aplikacjami robotów. Powstało narzędzie badawcze. Teraz należy zastosować je do badań możliwości technicznych skonstruowanego robota oraz potencjalnych jego aplikacji.

W celu sprawdzenia przydatności stworzonego narzędzia badawczego postanowiono zrealizować sterownik przeznaczony do konkretnych zadań. Były to sterownik "przemysłowy" — umożliwiający nauczenie lub wczytanie z pliku trajektorii do zrealizowania przez końcówkę robota (trajektoria może być wyrażona w różnych współrzędnych).

W wyniku przeprowadzonych badań okazało się, że tworzenie różnych sterowników nie stanowi problemu, a sterowniki realizują postawione przed nimi zadania. Ponadto przeprowadzono wiele badań dotyczących jakości regulacji w poszczególnych stopniach swobody. Ze względu na wysoki stopień nieliniowości oraz duże tarcie w układzie mechanicznym badano różnorodne algorytmy regulacji w celu uzyskania odpowiedniego kompromisu między dokładnością i prędkośćią regulacji.

Bibliografia

[1] Ambler A. P., Corner D. F.: *RAPT1 User's Manual*. Department of Artificial Intelligence, University of Edinburgh, 1984.

[2] Backes P., Hayati S., Hayward V., Tso K.: *The KALI Multi-Arm Robot Programming and Control Environment*. Proceedings of the NASA Conference on Space Telerobotics, 1989.

[3] Bidziński J., Mianowski K., Nazarczuk K., Słomkowski T.: *A manipulator with an arm of serial-parallel structure*. Archives of Mechanical Engineering, Vol.34 , 1992.

[4] Blume C., Jakob W.: *PASRO: Pascal for Robots*. Springer-Verlag, Berlin 1985.

[5] Blume C., Jakob W.: *Programming Languages for Industrial Robots*. Springer-Verlag, 1986.

[6] Connel J. H.: *A BehaviorB̄ased Arm Controller*. IEEE Transactions on Robotics and Automation, Vol.5, No.6, December 1989. str.784–791.

[7] Corke P., Kirkham R.: *The ARCL Robot Programming System*. Proceedings of the International Conference Robots for Competitive Industries, Brisbane, Australia, 14-16 July 1993. str.484-493.

[8] Dewhurst S. C., Stark K. T.: *Programming in C++*. Prentice Hall, Englewood Cliffs, 1989.

[9] Gini G., Gini M.: *ADA: A Language for Robot Programming?*. Computers In Industry, Vol.3, No.4, 1982, str.253–259.

[10] Gosiewski A.: *Dynamic Interactions in IRb Robots*. 7-th Italian-Polish Symposium on Theory and Mathematical Economics. Control and Cybernetics, Vol.15, No.3-4, 1986.

[11] Gosiewski A., Wieczorek W.: *Interakcje Dynamiczne w Robocie IRb*. Materiały I-ej Krajowej Konferencji Robotyki, Wrocław, Wrzesień 1985.

[12] Gosiewski A.: *Dynamic Interactions in 3-Main-Axes-Robot and Their Influence on CP Control*. Proceedings of the IFAC Symposium on Theory of Robots, Vienna, 1986.

[13] Gosiewski A.: *Roboty Przemysłowe*. Charakterystyka Prac Zrealizowanych w Ramach Celu Nr 203: *Rozwój Teorii Sterowania, Programowania oraz Modeli Dynamicznych Robotów i Manipulatorów Przemysłowych* w CPBR 7.1., Materiały III-tej Krajowej Konferencji Robotyki, (referat plenarny), Wrocław, wrzesień 1990.

[14] Gosiewski A.: *Dynamic Interactions in Robot Manipulators*. Rozdział w monografii: *Robotics Research and Applications*. (Selected topics presented at the 1-st, 2-nd and 3-rd National Conferences on Robotics), WNT, Warszawa, 1992.

[15] Gosiewski A., Szynkiewicz W.: *Zastosowanie algorytmu ruchu ślizgowego do układu sterowania manipulatora robota*. II Krajowa Konferencja Robotyki, Prace Naukowe ICT Politechniki Wrocławskiej, nr 75, 1988, str. 141-150.

[16] Gosiewski A., Grodecki A.: *On a New Structure of PID Controller for Robot Motion Control*. IASTED Conf. CONTROL 90, 1990, Lugano, Szwajcaria.

[17] Grodecki A., Gosiewski A.: *Hybrydowy sterownik siły/położenia dla robota IRP-6. Cz.I: opis kinematyczny*. IV Krajowa Konferencja Robotyki, 22-24.09.1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.126-133.

[18] Grodecki A.: *Hybrydowy sterownik siły/położenia dla robota IRP-6. Cz.II: struktura układu i wyniki eksperymentalne*. IV Krajowa Konferencja Robotyki, 22-24.09.1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.134-141.

[19] Grodecki A., Niezgoda J.: *Implementacja sprzężeń "feedforward" od prędkości i przyspieszenia w robocie IRp-6*. IV Krajowa Konferencja Robotyki, 22-24.09.1993, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1993, Vol.1, pp.142-149.

[20] Grodecki A.: *Analiza porównawcza układów sterowania siłowego robotów przemysłowych*. Opracowanie wewnętrzne, Instytut Automatyki, Politechnika Warszawska, Maj 1993.

[21] Grodecki A., Gosiewski A.: *Sprzężenia typu "feedforward" w regulatorze położenia ramion robota IRp-6*. III Krajowa Konferencja Robotyki, Prace Naukowe Instytutu Cybernetyki Technicznej Politechniki Wrocławskiej, Wrocław, 1990.

[22] Halang W. A., Sacha K. M.: *Real-Time Systems: Implementation of Industrial Computerised Process Automation*. World Scientific Publishing Co., Singapore 1992.

[23] Hayward V., Paul R. P.: *Robot Manipulator Control Under Unix RCCL: A Robot Control C Library*. International J. Robotics Research, Vol.5, No.4, Winter 1986. str.94-111.

[24] Hayward V., Hayati S.: *KALI: An Environment for the Programming and Control of Cooperative Manipulators*. Proceedings of the American Control Conference, 1988. str.473-478.

[25] Hayward V., Daneshmend L., Hayati S.: *An Overview of KALI: A System to Program and Control Cooperative Manipulators*. In: *Advanced Robotics*. Ed. Waldron K., Springer-Verlag, 1989.

[26] Hollerbach J. M.: *A Survey of Kinematic Calibration*. **In:** The Robotics Review 1. Eds: O. Khatib, J. J. Craig, T. Lozano-Perez, The MIT Press, Cambridge 1989.

[27] Kierzenkowski K.: *Wyznaczanie optymalnej sciezki przejscia przez labirynt*. II Sympozjum Naukowe: Techniki Przetwarzania Obrazu, 18-20 listopad 1993, Serock.

[28] Kręglewska U.: *Interfejs bezposredniego dostepu IBM AT do zasobow sterownika robota IRp-6*. III Krajowa Konferencja Robotyki, Prace Naukowe ICT, Politechnika Wrocławska, 1990.

[29] Kruszyński H., Kręglewska U., Szynkiewicz W.: *Stanowisko Badawcze dla robota IRb-6*. II Krajowa Konferencja Robotyki, Prace Naukowe ICT Politechniki Wrocławskiej, nr 78, 1988, str. 189-193.

[30] Lieberman L. I., Wesley M. A.: *AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly*. IBM Journal of Research and Development, Vol.21, No.4, July 1977, str.321–333.

[31] Lloyd J., Parker M., McClain R.: *Extending the RCCL Programming Environment to Multiple Robots & Processors*. Proceedings of the IEEE International Conference Robotics & Automation, 1988. str.465-469.

[32] Malcolm C., Smithers T.: *Programming Assembly Robots in Terms of Task Achieving Behavioural Modules: First Experimental Results*. Department of Artificial Intelligence, Research Paper No.410, University of Edinburgh, 1988.

[33] Mianowski K., Nazarczuk K., Wojtyra M.: *Badania własności dynamicznych oraz lokalna korekcja modelu kinematyki prototypu robota RNT* W: *Konstrukcja, sterowanie i programowanie złożonych systemów robotycznych* . Red.: Zieliński C., Zielińska T., Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 1997, pp.11–23.

[34] Mujtaba S., Goldman R.: *AL Users' Manual*. Stanford Artificial Intelligence Laboratory, 1979.

[35] Nilakantan A., Hayward V.: *The Synchronisation of Multiple Manipulators in Kali*. Robotics and Autonomous Systems, No.5, 1989.

[36] Paul R.: *WAVE: A Model Based Language for Manipulator Control*. The Industrial Robot, March 1977, str.10–17.

[37] Popplestone R.J., Ambler A.P., Bellos I.: *RAPT: A Language for Describing Assemblies*. The Industrial Robot, September 1978, str.131–137.

[38] Sacha K.: *Systemy czasu rzeczywistego*. Oficyna Wydawnicza Politechniki Warszawskiej. Warszawa 1993.

[39] Sacha K.: *Laboratorium systemu QNX*. Oficyna Wydawnicza Politechniki Warszawskiej. Warszawa 1995.

[40] Stroustrup B.: *Język C++*. WNT, Warszawa 1995.

[41] Szynkiewicz W., Gosiewski A., Janecki D.: *Weryfikacja doświadczalna modelu dynamiki manipulatora IRb-6*. III Krajowa Konferencja Robotyki, Prace Naukowe ICT, Politechnika Wrocławska, 1990, str. 255-260.

[42] Szynkiewicz W.: *Opis funkcjonalny stanowiska do testowania robotów*. Biuletyn Przemysłowego Instytutu Automatyki i Pomiarów PIAP Nr 1 (159), 1992 str. 29-46.

[43] Szynkiewicz W.: *Admissible Path Planning for Two Cooperated Robot Arms*, MELECON'94 (IEEE Mediterranen Electrotechnical Conference), Antalaya, Turkey, April 1994, pp.699-702.

[44] Szynkiewicz W.: *Zastosowanie funkcji sklejanych do aproksymacji ścieżek ruchu dla dwóch współpracujących robotów*. IV Krajowa konferencja robotyki, Tom 1, Prace Naukowe ICT Politechniki Wrocławskiej, Wrocław, 1993, str. 317-324.

[45] Szynkiewicz W.: *Planowanie skoordynowanych trajektorii ruchu dla dwóch współpracujących robotów*. w *Projekty badawcze granty w dziedzinie robotyki*, Biletyn PIAP, Warszawa, 1994.

[46] Szynkiewicz W., Gosiewski A.: *Coordinated Trajectories Planning for Two Cooperating Robots*. 1-st IFAC Workshop on New Trend in Design of Control Systems, Smolenice, Slovak Republik, September 1994.

[47] Szynkiewicz W., Zieliński C., Kierzenkowski K., Zielińska T., Grodecki A.: *Sterownik wielorobotowy do celów badawczych MRROC*. Sprawozdanie z wykonania pracy w ramach Program Automatyki, Technik Informacyjnych i Automatyzacji PATIA, Politechnika Warszawska, maj 1995.

[48] Szynkiewicz W., Zieliński C., Kierzenkowski K., Zielińska T., Grodecki A.: *Rozproszony sterownik wielorobotowy MRROC*, V Krajowa Konferencja Robotyki, 25-27 wrzesień 1996, Szklarska Poręba, Polska.

[49] Szynkiewicz W., Zieliński C.: *Sterownik dwóch współpracujących robotów o pięciu stopniach swobody*, V Krajowa Konferencja Robotyki, 24-26 wrzesień 1996, Świeradów Zdrój, Polska.

[50] Szynkiewicz W.: *Algorytmy planowania trajektorii w systemach wielorobotowych*, rozprawa doktorska, Politechnika Wrszawska 1996.

[51] Taylor R. H., Summers P. D., Meyer J. M.: *AML: A Manufacturing Language*. International Journal of Robotics Research, Vol. 1, No. 3, 1982.

[52] Topper A.: *A Computing Architecture for a Multiple Robot Controller*. M.Sc. Thesis, Department of Electrical Engineering, McGill University, Montreal, Canada, 1991.

[53] C. Pelich, F. M. Wahl: *A Programming Environment for a Multiprocessor-Net Based Robot Control Unit*. Proceedings of the 10th International Conference on High Performance Computing, Ottawa, Canada, June 5-7, 1996. (on CD-ROM)

[54] Zieliński C.: *TORBOL - język programowania robotów zorientowany na przemieszczanie obiektów*. X Krajowa Konferencja Automatyki, Wydawnictwo Politechniki Lubelskiej, Lublin 1988.

[55] Zieliński C.: *Środowisko programowe języka programowania robotów zorientowanego na przemieszczanie obiektów*. II Krajowa Konferencja Robotyki, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1988.

[56] Zieliński C.: *Klasyfikacja i metody definiowania języków programowania robotów: zastosowanie do sformułowania języka zorientowanego na przemieszczanie obiektów*. rozprawa doktorska, Politechnika Warszawska, Wydział Elektroniki, Warszawa 1988.

[57] Zieliński C.: *TORBOL - język programowania robotów przeznaczonych do wykonywania zadań transportowo-montażowych*. Archiwum Automatyki i Telemechaniki, Zeszyt 3, 1989.

[58] Zieliński C., Grodecki A., Kręglewska U., Śluzek A., Zielińska T.: *Propozycja stworzenia sterownika dla robotów, przeznaczonego do celów badawczych*, opracowanie wewnętrzne, Instytut Automatyki, 1989.

[59] Zieliński C., Śluzek A.: *Kinematyczne aspekty sterowania robotami wyposażonymi w narzędzia osiowo symetryczne*. III Krajowa Konferencja Robotyki, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990.

[60] Zieliński C.: *Przegląd cech istniejących języków programowania robotów*. Archiwum Automatyki i Telemechaniki. Zeszyt 3, 1989.

[61] Zieliński C.: *Opis semantyki rozkazów języków programowania robotów*. Archiwum Automatyki i Telemechaniki, Zeszyt 1-2, 1990.

[62] Zieliński C., Grodecki A., Kręglewska U., Śluzek A., Zielińska T.: *Koncepcja sterownika robotów przeznaczonego do celów badawczych*. III Krajowa Konferencja Robotyki, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.1, pp.336-341.

[63] Zieliński C.: *Zastosowanie sieci Petriego do opisu działania robota wyposażonego w czujniki*. III Krajowa Konferencja Robotyki, 19-21.09.1990, Wrocław, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1990, Vol.1, pp.330-335.

[64] Zieliński C.: *Incorporation of Sensors in Object-Level Robot Programming Language*. Proceedings of the 8-th CISM IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy'90, 2-6 July 1990, Cracow, Poland, Warsaw University of Technology Publications. pp.418-425.

[65] Zieliński C.: *TORBOL: An Object Level Robot Programming Language*. Mechatronics, Vol.1, No.4, Pergamon Press, 1991. str.469-485.

[66] Zieliński C.: *Description of Semantics of Robot Programming Languages*. Mechatronics, Vol.2 No.2, Pergamon Press, 1992.

[67] Zieliński C.: *Object Level Robot Programming Languages*. **In:** *Robotics Research and Applications*. Ed.: A. Morecki et.al., Warsaw 1992. str.221-235.

[68] Zieliński C., Grodecki A., Kręglewska U., Sobczyk J., Śluzek A., Zielińska T.: *Sterownik robotów przeznaczony do celów badawczych*. opracowanie dla KBN, Instytut Automatyki Politechniki Warszawskiej, grudzień 1992.

[69] Zieliński C.: *Flexible Controller for Robots Equipped with Sensors*. 9th Symp. Theory and Practice of Robots & Manipulators, Ro.Man.Sy'92, 1-4 Sept. 1992, Udine, Italy, Lect. Notes: Control & Information Sciences 187, Springer-Verlag, 1993. str.205-214.

[70] Zieliński C.: *Robot Object-Oriented Pascal Library: ROOPL*. J. of Theoretical and Applied Mechanics, Vol.31, No.3, 1993. str.525-535.

[71] Zieliński C.: *Controller Structure for Robots with Sensors*. Mechatronics, Vol.3, No.5, Pergamon Press, 1993. str.671-686.

[72] Zieliński C.: *Sterownik robotów wyposażonych w różnorodne czujniki*. Biuletyn PIAP, Zbiór referatów wygłoszonych na seminarium poświęconym grantom dotyczącym robotyki finansowanym przez KBN, 7-8 grudnia 1993, PIAP, Warszawa. pp.31-40.

[73] Zieliński C., Grodecki A.: *Reaktywne sterowanie robotem wyposażonym w czujnik sił*, Sprawozdanie z realizacji pracy badawczej – własnej 503/031/003/1, Instytut Automatyki, Politechnika Warszawska, styczeń 1994.

[74] Zieliński C.: *Sensory Robot Motions*. Archives of Control Sciences, Vol.3, no.1, 1994. str.5-20.

[75] Zieliński C., Zielińska T.: *Sensor-Based Reactive Robot Control*. Proceedings of the 10-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'94, September 1994, Gdańsk, Poland.

[76] Zieliński C.: *Reaction Based Robot Control*. Mechatronics, Vol.4, no.8, 1994. pp.843-860

[77] Zieliński C., Zielińska T.: *Software for Mechatronic Devices*. Joint Hungarian-British Mechatronics Conference, 21-23 September 1994, Budapest, Węgry. Mechatronics: the basis for new Industrial Development. Ed,: M. Acar, J. Makra, E. Penney, Computational Mechanics Publications. Southampton, Boston, 1994

[78] Zieliński C.: *Distributed Software for Mechatronic Systems*. International Journal of Intelligent Mechatronics: Design and Production, Vol.1, No.1, 1994. str.11-24.

[79] Zieliński C.: *Sterownik robotów przeznaczony do celów badawczych*. IV Krajowa Konferencja Robotyki, 22-24 wrzesień 1993, Wydawnictwo Politechniki Wrocławskiej, Wrocław 1994, Vol.1, pp.73-80.

[80] Zieliński C.: *Robot Programming Methods*. Publishing House of Warsaw University of Technology, 1995.

[81] Zieliński C.: *Control of a Multi-Robot System*, 2nd International Symp. Methods & Models in Automation & Robotics MMAR'95, 30 Aug.–2 Sept. 1995, Międzyzdroje, Poland. str.603-608.

[82] Zieliński C.: *Reactive Robot Control Applied to Acquiring Moving Objects*, 3rd International Symposium on Methods and Models in Automation and Robotics MMAR'96, 10-13 September 1996, Międzyzdroje, Poland.

[83] Zieliński C.: *Zastosowanie sterowania reakcyjnego do chwytania ruchomych przedmiotów*, V Krajowa Konferencja Robotyki, 24-26 wrzesień 1996, Świeradów Zdrój, Polska.

[84] Zieliński C., Szynkiewicz W.: *Control of Two 5 d.o.f. Robots Manipulating a Rigid Object*, IEEE International Symposium on Industrial Electronics ISIE'96, 17-20 June 1996, Warsaw, Poland.

[85] Zieliński C., Szynkiewicz W., Gosiewski A.: *Sterownik robota o dużej sztywności – ogólna struktura*. Sprawozdanie z wykonania pracy w ramach PATIA, Instytut Automatyki i Informatyki Stosowanej, Politechnika Warszawska, grudzień 1995.

[86] Zieliński C., Szynkiewicz W.: *Sterownik robota o dużej sztywności – MRROC+*. Sprawozdanie z wykonania pracy w ramach PATIA, Instytut Automatyki i Informatyki Stosowanej, Politechnika Warszawska, czerwiec 1996.

[87] Zieliński C. i inni: *Sterowanie i konstrukcja złożonych systemów robotycznych*. Raport Syntetyczny z wykonania pracy w ramach PATIA, Instytut Automatyki i Informatyki Stosowanej, Politechnika Warszawska, czerwiec 1996.

[88] Zieliński C., Szynkiewicz W., Gosiewski A.: *Wyższe warstwy sterowania sztywnym robotem —- koncepcja*. Sprawozdanie z wykonania pracy w ramach PATIA, Instytut Automatyki i Informatyki Stosowanej, Politechnika Warszawska, listopad 1996.

[89] Zieliński C., Szynkiewicz W., Gosiewski A.: *Oprogramowanie układu sterowania sztywnym robotem*. Sprawozdanie z wykonania pracy w ramach PATIA, Instytut Automatyki i Informatyki Stosowanej, Raport IAIS nr 97–05, Politechnika Warszawska, Raport IAIS nr 97–05, czerwiec 1997.

[90] Zieliński C.: *Reactive Robot Control Applied to Acquiring Moving Objects*, Proceedings of the 3rd International Symposium on Methods and Models in Automation and Robotics MMAR'96, 10–13 September 1996, Międzyzdroje, Poland. Vol.3, str.893–898.

[91] Zieliński C.: *Object–Oriented Approach to Programming Multi–Robot Systems*, Proceedings of the 11th CISM IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy'96, 1–4 July 1996, Udine, Italy. Ed.: Morecki A., Bianchi G., Rzymkowski C., Springer Verlag, Wien, New York, 1997. str.373–380.

[92] Zieliński C.: *Object–Oriented Robot Programming*, Robotica, Vol.15, 1997. str.41–48.

[93] Szynkiewicz W., Zieliński C., Kierzenkowski K., Zielińska T., Grodecki A., Gosiewski A.: *Środowisko programowe do tworzenia sterowników wielorobotowych dla złożonych zastosowań*, Automation'97, 5–7 marca 1997 Warszawa, Polska. Vol.1, pp.127–134.

[94] Zieliński C.: *Object–Oriented Programming of Multi–Robot Systems Utilising Sensory Information*, 3rd ECPD International Conference on Advanced Robotics, Intelligent Automation and Active Systems, 15–17 September 1997, Bremen, Germany, pp.176–181.

[95] Zieliński C.: *Object–Oriented Programming of Multi–Robot Systems*, Proceedings of the 4th International Symposium on Methods and Models in Automation and Robotics MMAR'96, 26–29 August 1997, Międzyzdroje, Poland, pp.1121–1126.

[96] Zieliński C., Szynkiewicz W.: *Systemy MRROC i MRROC++; Część I: Struktura*. W: *Konstrukcja, sterowanie i programowanie złożonych systemów robotycznych* . Red.: Zieliński C., Zielińska T., Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 1997, pp.25–40.

[97] Zieliński C., Szynkiewicz W.: *Systemy MRROC i MRROC++; Część II: Realizacja*. W: *Konstrukcja, sterowanie i programowanie złożonych systemów robotycznych* . Red.: Zieliński C., Zielińska T., Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 1997, pp.41–54.

[98] Zieliński C., Rydzewski A., Szynkiewicz W., Woźniak A.: *Układ sterowania robotem o strukturze szeregowo–równoległej* W: *Konstrukcja, sterowanie i programowanie złożonych systemów robotycznych* . Red.: Zieliński C., Zielińska T., Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 1997, pp.55–76.

[99] Zieliński C., Zielińska T. (Red.): *Konstrukcja, sterowanie i programowanie złożonych systemów robotycznych* . Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 1997,

[100] NEWSLINE: *New product equipment*. Robotics Today, vol.6, no.3, SME 1993, str.5

[101] *QNX System Architecture*. Quantum Software, 1992.

[102] *Turbo C++: Getting Started*. Borland International Incorporated, 1990.

[103] *User's Guide to VAL II: Programming Manual*. Ver.2.0, Unimation Incorporated, A Westinghouse Company, August 1986.

[104] *Using OS-9*. Ver.3.0, Microware Systems Corporation, November 1993.

[105] Kręglewska U., Sacha K., *Sprawozdanie z realizacji grantu KBN nr 8 T11A 028 09.*

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**.

Dodatek B

Rys historyczny

W latach 1983-90, w ramach Centralnego Programu Badawczo-Rozwojowego CPBR 7.1 cel 203, pod kierownictwem prof. A. Gosiewskiego, prowadzone były w Instytucie Automatyki Politechniki Warszawskiej prace nad tematem: *Metody analizy i syntezy sterowania ruchem robotów oraz system komputerowego wspomagania programowania robotów.* W ramach tych prac C. Zieliński opracował język programowania robotów zorientowany na przemieszczanie obiektów – TORBOL (Transformation Of Relations Between Objects Language) [54, 55, 56, 57, 65]. Ponadto zdobyto duże doświadczenie w zakresie programowania [60, 67, 61, 66], modelowania i sterowania robotów [10, 11, 12, 13, 14, 15, 29, 41, 42, 16, 21].

Kod wynikowy dla programu napisanego w języku TORBOL był generowany dla sterownika przemysłowego połączonego z komputerem klasy IBM/PC za pomocą interfejsu bezpośredniego dostępu opracowanego przez U. Kręglewską [28]. Do sterowania tym systemem wykorzystano koncepcję programu wirtualnego [65] polegającą na wstrzymywaniu działania procesora sterownika przemysłowego na czas zmiany przez komputer nadrzędny argumentów programu realizującego ruchy ramieniem. Aczkolwiek powyższy system bardzo dobrze nadawał się do realizacji zadań transportowych, to stwarzał poważne problemy przy dołączaniu czujników. Przemysłowy układ sterowania także nie nadawał się do tych celów. Dlatego też w październiku 1989 roku C. Zieliński przedstawił dyrektorowi Instytutu Automatyki prof. K. Malinowskiemu propozycję realizacji tematu: *Sterownik robotów do celów badawczych* oraz prośbę o pomoc w znalezieniu źródła finasowania dla proponowanych prac. W listopadzie 1989 rozpoczęły się poszukiwania środków na sfinansowanie przedsięwzięcia.

W trakcie poszukiwań funduszy stworzono zespół (C. Zieliński, A. Grodecki, U. Kręglewska, A. Śluzek, T. Zielińska), który rozwinął wstępną koncepcję sterownika badawczego [58, 62]. Początkowo zakładano, że oprogramowanie sterownika będzie pracowało na dwu procesorach: w oryginalnym sterowniku przemysłowym oraz komputerze IBM/PC. W sterowniku przemysłowym miały być realizowane następujące zadania: generacja trajektorii, przeliczanie współrzędnych, zlecanie wykonania ruchu serwomechanizmom oraz odczyt aktualnego położenia ramienia. Komputer nadrzędny miał interpretować polecenia dla robota wyrażone w języku programowania.

Poszukiwanie funduszy przez prof. K. Malinowskiego zakończyło się powodzeniem dopiero w lipcu 1990 roku – wtedy Ministerstwo Edukacji Narodowej zdecydowało się sfinansować fazę projektowania proponowanego systemu. Faza ta przebiegała w dwu etapach (finasowanych przez MEN), które zakończyły się w czerwcu 1991 roku. Zespół w składzie: C. Zieliński, A. Grodecki, U. Kręglewska, J. Sobczyk, A. Śluzek, T. Zielińska do czerwca

1991 roku stworzył pełną dokumentację oraz napisał większość programów tworzących sterownik badawczy. W trakcie prac C. Zieliński zrezygnował z pierwotnej koncepcji definiowania specjalizowanego języka programowania robotów dla sterownika, gdyż im bardziej uniwersalny jest język, tym trudniej go zaimplementować, natomiast rezygnacja z uniwersalności spotykanej w językach programowania komputerów niepotrzebnie ogranicza możliwości projektowanego urządzenia. Należy podkreślić, że nie można było czynić żadnych założeń co do sposobu wykorzystania sterownika w przyszłości, z racji jego przeznaczenia do celów badawczych. Ponadto, przy realizacji konkretnego zadania, zazwyczaj większość możliwości uniwersalnego języka jest niewykorzystana. Oczywiście interpreter tego języka musi stale rezydować w pamięci maszyny, w ten sposób ograniczając zasoby pamięciowe przeznaczone do innych celów. Interpretacja rozkazów, natomiast, zabiera cenny czas potrzebny do sterowania. W związku z tym zdecydowano, że należy tak skonstruować sterownik, aby do realizacji każdego zadania trzeba było go modyfikować, z tym że niezbędne modyfikacje będą ściśle określone i łatwe do przeprowadzenia. Przyjęto, że sterownik będzie składany z wywołań procedur bibliotecznych, niczym z klocków Lego. Dokumentacja opisuje sposób konstruowania takiego sterownika. Sposób ten spełnił postawione przed nim wymagania. Ponieważ oprogramowanie sterownika składało się z wielu procesów, które miały pracować równolegle, potrzebny był współbieżny system operacyjny. Zdecydowano się wykorzystać system pierwotnie opracowany przez J. Sobczyka do celów przetwarzania numerycznego. System ten umożliwiał wykonanie wielu procesów na jednym komputerze w podziale czasu.

Jednocześnie w 1991 C. Zieliński i A. Śluzek realizowali tzw. pracę własną przyznaną przez IA PW, w ramach której rozwiązali proste i odwrotne zagadnienie kinematyczne dla robota IRp-6 wyposażonego w narzędzie osiowo-symetryczne [59], natomiast W. Szynkiewicz opracował generator trajektorii liniowej. Dzięki tej pracy szczegółowo przetestowano wszystkie przeliczniki współrzędnych stosowane w sterowniku.

Równolegle w 1991 roku zgłoszono na pierwszy konkurs Komitetu Badań Naukowych propozycję implementacji sterownika badawczego. W październiku 1991 KBN przyznał grant nr 7 1083 91 01 na powyższą pracę. Do grudnia 1992 roku zespół ukończył prace nad sterownikiem RORC [68, 69, 71, 79, 72]. Obecnie dwa egzemplarze takiego sterownika pracują w Pracowni Robotyki Instytutu Automatyki i Informatyki Stosowanej Politechniki Warszawskiej oraz, od kwietnia 1993, jeden egzemplarz w Instytucie Automatyki Politechniki Śląskiej.

Sterownik, bądź pokaźne fragmenty biblioteki, z której się składa, został wykorzystany do realizacji wielu prac badawczych. Między innymi: A. Grodecki badał algorytmy sterowania siłowego [17, 18, 20] oraz algorytmy sterowania pozycyjnego [19], C. Zieliński koncepcję sterowania reaktywnego [76, 75, 76, 90], natomiast K. Kierzenkowski wykorzystał kamerę do sterowania robotem [27]. Ponadto zrealizowano wiele prac magisterskich wykorzystując ten system. Od 1989 roku, W. Szynkiewicz zajmował się zagadnieniami związanymi ze współpracą dwu robotów [43, 44, 45, 46].

W połowie 1994 roku, z inicjatywy prof. K. Malinowskiego oraz prof. W. Włosińskiego powołano do życia Program Automatyki, Technik Informacyjnych i Automatyzacji (PATIA). Jednym z głównych celów tego programu było stworzenie dogodnych warunków integracji naukowców w celu realizacji badań interdyscyplinarnych. Radzie Programowej PATIA zaproponowano sfinansowanie prac nad dość szerokim tematem z zakresu robotyki, którego elementem było stworzenie sterownika systemu wielorobotowego (*Organizacja Środowiskowego Laboratorium Robotyki oraz projekt systemu wielorobotowego zawierającego szybkiego robota*). W ramach tej pracy uogólniono koncepcje wykorzystane przy realizacji sterownika badawczego na systemy wielorobotowe [78, 77, 93].

Do sterowania systemem wielorobotowym potrzebne było znaczne zwiększenie mocy obliczeniowej sterownika, w stosunku do tej, która była potrzebna RORC. Obliczenia realizowane przez sterownik należało rozproszyć na wielu procesorach. Badania K. Sachy wskazały na przydatność systemu QNX-4 do sterowania różnorodnymi systemami złożonymi [22, 38]. Dlatego zdecydowano się zrezygnować z poprzedniego systemu operacyjnego i wykorzystać system operacyjny czasu rzeczywistego QNX-4. System ten pracuje na komputerach połączonych siecią Ethernet, przez co moc obliczeniowa tworzonego sterownika może być w łatwy sposób powiększana, gdy wykonywane zadanie tego wymaga. Zmiana systemu operacyjnego pociągnęła za sobą konieczność zmiany sposobu komunikowania się procesów oraz metody powoływania ich do życia. Zmianę tę wykonał K. Kierzenkowski. Ponadto zdecydowano się ulepszyć interfejs z użytkownikiem. System ten składa się dwu robotów IRp-6, o pięciu stopniach swobody każdy, toru jezdnego, na którym może poruszać się jeden z robotów, taśmociągu, systemu wizyjnego, czujnika sił i momentów oraz kilku innych czujników binarnych. Z. Buśko oraz J. Frączek ITLiMS PW rozwiązali proste i odwrotne zagadnienie kinematyczne dla robota IRp-6 posadowionego na torze jezdnym, traktowanym jako urządzenie o sześciu stopniach swobody. W. Szynkiewicz dołączył dostarczone procedury do całości systemu wielorobotowego, więc obecnie jeden z robotów, w zależności od potrzeb realizowanego zadania, może mieć 5 lub 6 stopni swobody. Prace nad systemem wielorobotowym, nazwanym MRROC[1], zakończono w maju 1995 roku.

W latach 1995–1998 w ramach trzech kolejnych tematów realizowanych pod kierownictwem C. Zielińskiego w programie PATIA skonstruowano sterownik [85, 86, 87, 88, 89] robota RNT charakteryzującego się dużą sztywnością [3]. Robot ten został zaprojektowany i skonstruowany przez K. Nazarczuka i K. Mianowskiego z Instytutu Techniki Lotniczej i Mechaniki Stosowanej w ramach wcześniejszych prac prowadzonych w tym instytucie. Początkowo korzystano z prostego sterownika sprzętowego wykonanego przez M. Nazarczuka, ale wkrótce okazało się, że do badań potrzebna jest zupełnie inna elektronika. W związku z tym nowe układy elektroniczne zostały zaprojektowane i wykonane przez A. Rydzewskiego [98]. Algorytmami regulacji dla serwomechanizmów zajął się A. Woźniak [98], natomiast nową strukturę oprogramowania całości opracował C .Zieliński [91, 92, 94, 95, 96, 97]. Implementację całości oprogramowania przeprowadzili W. Szynkiewicz i C .Zieliński [98]. Procedury rozwiązujące proste i odwrotne zagadnienie kinematyki oraz koncepcję korektorów modelu kinematycznego opracował M. Wojtyra [33]. W odróżnienia od podejścia proceduralnego do programowania stosowanego przy realizacji systemu `MRROC` tym razem zdecydowano się użyć podejście obiektowe i w związku z tym system ochrzczono `MRROC++`. Prace programistyczne rozpoczęto od warstw najniższych, z czasem rozbudowując system. Początkowo próbowano zrealizować każdy serwomechanizm jako oddzielny proces, ale czas przełączania procesów był tak znaczny, że wpłynęło to na wydłużenie czasu próbkowania. Dlatego w kolejnej wersji wszystkie serwomechanizmy zgrupowano w jednym procesie. Po trzech latach prac udało się doprowadzić do stanu, w którym robot potrafił rysować i frezować w drewnie. Kalibracją tego systemu zajęli się J. Frączek oraz Z. Buśko. Ponieważ, podobnie jak `MRROC`, `MRROC++` jest jedynie biblioteką modułów przeznaczoną do programowania systemów wielorobotowych, dla zweryfikowania przydatności tego systemu należało stworzyć kilka sterowników do konkretnych zadań. W tym celu W. Szynkiewicz i C .Zieliński stworzyli wersję "przemysłową" sterownika oraz sterowniki do kalibrowania robota i frezowania w materiałach niezbyt twardych. W. Szynkiewicz i i A. Woźniak przeprowadzili wiele eksperymentów identyfikacyjnych w celu określenia najwłaściwszego algorytmu regulacji do tego robota charakteryzującego się dużym tarciem oraz nieliniowością. Efekt dwóch pierwszych lat prac opisano w książce wydanej pod redalcją C. Zielińskiego oraz T. Zielińskiej [99].

[1] *Ang.* – Multi-Robot Research-Oriented Controller

W roku 1998 A. Rydzewski wykonał nowe sterowniki osi dla robota IRp-6 posadowionego na torze jezdnym oraz interfejs łączący te sterowniki z komputerem klasy IBM PC. W odróżnieniu od sterowników wykorzystywanych przez robota RNT tutaj pomiar położenia osi pochodził z rezolwera, a nie impulsatora. Parametry regulatorów, o strukturze takiej samej jak w przypadku robota RNT, dobrał A. Woźniak. Proces EDP dla robota IRp-6 posadowionego na torze jezdnym został napisany i uruchomiony przez C. Zielińskiego oraz W. Szynkiewicza. Wyższe warstwy sterownika zostały takie same jak dla robota RNT – potwierdzając uniwersalność stworzonego oprogramowania. W ten sposób pojawił się drugi robot wykorzystujący system `MRROC++`.