Programowe struktury ramowe do tworzenia sterowników robotów

Cezary Zieliński
Przemysłowy Instytut Automatyki i Pomiarów PIAP, 02-486 Warszawa, Al. Jerozolimskie 202

Tomasz Kornuta
Instytut Automatyki i Informatyki Stosowanej, Politechnika Warszawska, 00-665 Warszawa, ul. Nowowiejska 15/19, Gmach Elektroniki

Streszczenie: Liczba typów robotów oraz wykorzystywanych przez nie czujników niezmiennie wzrasta, otwierając wciąż nowe pola ich zastosowań. Różnorodność ta wpływa na zapotrzebowanie na narzędzia ułatwiające ich programowanie. W artykule skupiono uwagę na narzędziach programistycznych umożliwiających tworzenie wysokopoziomowych sterowników robotów. Pokrótce omówiono ewolucję metod programowania robotów, od języków specjalizowanych, przez biblioteki, aż po programowe struktury ramowe. W szczególności omówiono szereg popularnych programowych struktur ramowych, umożliwiających tworzenie złożonych sterowników robotów.

Słowa kluczowe: metody programowania robotów, języki programowania robotów, programowe struktury ramowe, sterowniki robotów

1. Wstęp

Od pojawienia się pierwszych robotów w przemyśle na początku lat 60. XX wieku, liczba typów robotów oraz ich zastosowań stale rośnie. Od przełomu lat 70. i 80. zaczęto powszechnie stosować układy mikroprocesorowe do budowy systemów sterowania złożonych urządzeń elektromechanicznych, w tym i robotów.

W wyniku tego nastąpiło przesuniecie znacznej części wysiłku projektowego z budowy sprzętu na opracowywanie oprogramowania. Lawinowo rosnąca liczba tego typu systemów spowodowała, że każdorazowe rozpoczynanie pisania programu sterującego od zera stało się nieopłacalne. Co więcej, na gruncie informatyki juz wcześniej zajęto się problemem powtórnego użycia oprogramowania (ang. *reuse*). Zaletą takiego podejścia jest nie tylko zmniejszenie kosztów tworzenia oprogramowania, ale również zwiększenie jego niezawodności, głównie dzięki dłuższemu testowaniu i to przez wielu użytkowników. Przy tworzeniu układu sterowania robota, prócz kwestii powtórnego użycia istniejących modułów, istotne są wskazówki dla projektanta, jak zdekomponować złożony system, jaką nadać mu strukturę i jak skomunikować wyróżnione podsystemy. Zanim zostaną przedstawione współczesne rozwiązania tych problemów warto jest spojrzeć na ewolucję podejścia do tworzenia oprogramowania sterującego robotami.

Autor korespondujący:
Cezary Zieliński, c.zielinski@ia.pw.edu.pl

Artykuł recenzowany
nadesłany 16.05.2014 r., przyjęty do druku 17.10.2014 r.

1.1. Języki specjalizowane

Pierwsze roboty przemysłowe były programowane mechanicznie przez ustawianie konfiguracji matryc diodowych oraz potencjometrów, lub zderzaków określających zakresy ruchu poszczególnych silników [49]. Następnie stosowano programowanie nietekstowe zwane uczeniem, w trakcie którego operator przestawiał lub wodził manipulator między punktami, które są wpierw zapamiętywane, a następnie odtwarzane przy realizacji programu. Budowa układów sterowania z wykorzystaniem coraz wydajniejszych mikroprocesorów umożliwiła rozszerzenie programowania nietekstowego o pewne elementy programowania tekstowego, dzięki czemu powstały formy hybrydowe, stosowane obecnie na szeroką skalę [71]. Większość języków programowania robotów była wzorowana na języku BASIC, przy czym dodatkowo wbudowano typy danych związane z opisem geometrycznym pozycji końcówki robota oraz miejsc charakterystycznych otoczenia. Dlatego języki te operowały m.in. zmiennymi globalnymi oraz wykorzystywały instrukcje skoku do etykiety. Pojawiły się również języki programowania robotów, które były wzorowane na językach strukturalnych, a w szczególności na języku Pascal. Obecnie rozgraniczenie języków na te dziedziczące po BASICu i Pascalu nie jest ostre – mamy raczej do czynienia z płynnie przesuwającą się granicą związaną z dziedziczeniem po obu przodkach. Należy zwrócić

uwagę, że ta forma programowania robotów wymaga stworzenia: oprogramowania sterującego sprzętem, interpretera języka programowania robota oraz programu czynności robota, wytworzonego przez użytkownika w interpretowanym języku. Pojawiające się nowe typy robotów oraz nowe ich zastosowania wymagały stałej modyfikacji oprogramowania sterującego, a w szczególności języka programowania, a więc i interpretera, co było dość uciążliwe. Niemniej jednak w latach 70. oraz na początku lat 80. XX wieku ten sposób podejścia do budowy systemów sterowania i programowania robotów był faworyzowany. Tak więc uwaga konstruktorów skoncentrowana była na tworzeniu specjalizowanych języków. Jednym z pierwszych języków przeznaczonych do sterowania manipulatorami był WAVE [52] opracowany w Stanford Artificial Intelligence Laboratory w początkach lat 70. XX wieku. Bardziej rozwiniętą formą tego jezyka stał się AL [45] – opracowany w tym samym laboratorium. O ile w pierwszym rozkazy ruchu dotyczyły końcówki manipulatora, to w drugim odnosiły się do obiektów znajdujących się w otoczeniu robota. Programista musiał dbać o to, aby działania robota w rzeczywistym środowisku były adekwatnie odzwierciedlane w modelu środowiska rezydującym w pamięci układu sterowania. Potomkiem języka AL był SRL [10], ale mimo tego, że został rozbudowany, nie usunięto w nim podstawowej niedogodności, jaką jest możliwość powstania rozbieżności między faktycznym stanem środowiska i jego odbiciem w modelu. Ta niedogodność została usunięta w języku TORBOL [61, 62]. O ile w językach AL, SRL i TORBOL obiekty były modelowane jako trójściany reprezentowane macierzami jednorodnymi, to w języku RAPT [54, 5] reprezentowano je jako kształty składające się z podstawowych brył geometrycznych. Prace badawcze nad specjalizowanymi językami programowania robotów prowadzone w latach 70. i 80. XX wieku koncentrowały się na geometrycznym modelowaniu środowiska. Podstawowym problemem było dokładne określenie pozycji modelowanych obiektów, tudzież ich rozpoznawanie. Wspomniane języki w małym stopniu były przystosowane do korzystania z percepcji. Należy pamiętać, że w tamtych czasach moc obliczeniowa komputerów nie umożliwiała przetwarzania w czasie rzeczywistym odczytów uzyskanych z takich czujników, jak np. kamery. Gdy moc obliczenia stosowanych komputerów wzrosła, wysiłek badawczy został skierowany na wykorzystanie czujników. Wtedy okazało się, że dotychczas opracowane języki specjalizowane raczej są przeszkodą niż pomocą w dalszym rozwoju robotyki.

Dlatego też uwaga została skierowana na bardziej elastyczne rozwiązania, omówione w kolejnej części artykułu. Warto dodać, iż poza językami specjalizowanymi nadal dostarczanymi przez producentów robotów przemysłowych, obecnie języki specjalizowane wykorzystywane są również jako narzędzie do integracji rozdzielnie stworzonych komponentów w spójny system – najczęściej są to języki skryptowe.

1.2. Biblioteki

Wspomniane niedogodności związane ze zmianami w interpreterze, przy każdorazowej potrzebie dodania nowego rozkazu do języka specjalizowanego, a w szczególności rozkazów odwołujących się do czujników, których różnorodność stale wzrastała, spowodowały przeniesienie uwagi projektantów na biblioteki funkcji i procedur napisanych w językach ogólnego przeznaczenia. Przykładowymi bibliotekami były PASRO (Pascal for Robots) [9, 10] – na bazie języka Pascal oraz RCCL (Robot Control C Library) [34] – stanowiąca zestaw funkcji w języku C. W RCCL tworzono równania pozycyjne [53] przez wywołanie odpowiednich procedur bibliotecznych operujących na macierzach jednorodnych. W skład tak konstruowanego równania wchodziła macierz jednorodna reprezentująca końcówkę manipulatora. Zadaniem generatora trajektorii było takie przemieszczenie końcówki, aby równanie zostało spełnione. Istniały funkcje wprowadzające podatność mechaniczną, powodujące wywieranie siły lub przerywające ruch przy napotkaniu oporu. Pewne zmienne zostały stowarzyszone z czujnikami. Ich wartości były uaktualniane z częstotliwością próbkowania serwomechanizmów. Funkcje biblioteczne za pomocą tych zmiennych sterowały ramieniem.

Podobną biblioteką do RCCL była ARCL (Advanced Robot Control Library) [25]. Koncepcje leżące u podstaw biblioteki RCCL zostały rozwinięte w bibliotece RCI (Robot Control Interface) [42], która umożliwiała sterowanie wieloma robotami. Innym rozwinięciem RCCL była biblioteka KALI [33, 6, 32, 50], w której każdy ruch był implementowany jako oddzielny proces. Synchronizację między ruchami robotów osiągano przez zastosowanie zmiennych synchronizujących (ang. *flag*) oraz odpowiedni dobór parametrów ruchu.

1.3. Programowe struktury ramowe

Ze względu na opracowywane nowe typy robotów i czujników, a także coraz bardziej złożone zastosowania, stopniowo wagi zaczęły nabierać takie czynniki jak wygoda korzystania z danej biblioteki czy powtórne użycie kodu. Z tego powodu biblioteki, korzystając z rozwiązań zaczerpniętych z inżynierii oprogramowania, stopniowo zaczęły ewoluować w programowe struktury ramowe (ang. *programming framework*). W odróżnieniu od bibliotek (które są jedynie zestawem funkcji lub obiektów) struktury ramowe składają się z dwóch zasadniczych elementów: bibliotek modułów (funkcji umożliwiających sterowanie konkretnymi robotami, a przez ich wywoływanie tworzenie programów użytkowych, czyli programów zleconych systemowi przez użytkownika) oraz właściwego trzonu samej struktury ramowej (części niezmiennej, niezależnej od zadania czy sprzętu, zawierającej np. mechanizmy komunikacji między modułami czy zarządzania konfiguracją). Dodatkowo struktury ramowe narzucają wzorce, zarówno tworzenia modułów użytkowych, jak i ich użycia oraz łączenia. Mówiąc skrótowo, struktura ramowa to biblioteka oraz wzorce jej wykorzystania. Ponadto, wraz ze strukturami ramowymi zaczęto dostarczać również zestawy narzędzi (ang. *toolchains*) użytecznych podczas implementacji nowych modułów czy całych sterowników. Przykładowymi narzędziami są symulatory, debuggery oraz całe środowiska programistyczne [43]. Efektem ubocznym wykorzystania programowych struktur ramowych był zanik trójpodziału na oprogramowanie sterujące sprzętem, interpreter języka programowania robota oraz program użytkowy. Program użytkowy tworzony jest w tym samym języku, co oprogramowanie sterujące sprzętem. Zanika więc potrzeba stosowania interpretera, a całość oprogramowania może być przetworzona przez kompilator języka bazowego bezpośrednio na kod wykonalny komputera sterującego robotem. W tej sytuacji raczej należy mówić o tworzeniu sterownika robota niż o programowaniu robota, aczkolwiek stworzony sterownik oczywiście będzie realizował program użytkowy. Niemniej jednak program użytkowy i oprogramowanie sterujące sprzętem będą w tym przypadku stanowiły integralną całość. Gwoli ścisłości, należy dodać, że tworzenie sterownika takim sposobem nie wyklucza inkorporacji interpretera dodatkowego języka specjalizowanego, za pomocą którego użytkownik może modyfikować działanie systemu [64], ale nie jest to rozwiązanie powszechne.

Cechą charakterystyczną programowych struktur ramowych jest stosowanie wzorców. Literatura informatyczna wskazuje na istnienie trzech rodzajów wzorców [35]. Wzorzec architektury określa podstawową strukturę systemu. Wzorzec projektowy stanowi rozwiązanie często spotykanego problemu w pewnej dziedzinie projektowania. Wyróżnia się również idiomy, czyli wzorce odnoszące się do konkretnego języka programowania, wskazujące jak zaimplementować w konkretnym języku zachowanie części

składowej większego systemu. Powyższe wzorce rozróżnia stopień ich abstrakcyjności. Wzorzec architektury określa wysokopoziomową strukturę systemu oraz jego globalne właściwości. Wzorzec projektowy określa strukturę wewnętrzną i zachowanie części składowej systemu oraz związek z innymi częściami. Idiomy zależą od wykorzystywanego paradygmatu programowania oraz stosowanego języka programowania, a więc stanowią najbardziej konkretną formę wzorca.

Rozpatrywane w artykule programowe struktury ramowe mogą służyć do tworzenia systemów sterowania robotami traktowanymi jako całość, ale też mogą być wyspecjalizowane w tworzeniu pewnych ich podsystemów, np. percepcyjnych, bądź koncentrować się tylko na komunikacji między podsystemami. Stanowią one wtedy tzw. warstwę pośrednią (ang. *middleware*), co można również przetłumaczyć na "oprogramowanie pośredniczące". Oprogramowanie to umożliwia komunikację między podsystemami lub systemami [13, 14]. Przykładem takiego oprogramowania może być CORBA [51, 3]. Zazwyczaj nad warstwą pośredniczącą nadbudowuje się właściwą programową strukturę ramową określającą strukturę systemu sterującego robotem lub robotami.

W pracy [18] wskazano istotniejsze zastosowania programowych struktur ramowych używanych w robotyce. Istnieją struktury ramowe, które służą tylko jednemu z poniższych zastosowań, ale są też i takie, za pomocą których można rozwiązać kilka problemów. Tak więc wyróżniono struktury ramowe do tworzenia:

- programowych sterowników sprzętu (ang. *device driver*), jakimi są efektory (siłowniki) i receptory (czujniki),
- mechanizmów komunikacji między komponentami, z których składany jest system (czyli oprogramowanie pośredniczące),
- opakowanych komponentów realizujących specyficzne funkcje systemu, np. do tworzenia podsystemów percepcyjnych wraz z mechanizmami umożliwiającymi dołączenie takiego podsystemu do pozostałych komponentów całości, a w konsekwencji dających możliwość, z jednej strony – niezależnej pracy tylko nad percepcją, a z drugiej strony – testowania percepcji w jej naturalnym środowisku, jakim jest zestaw elementów wchodzących w skład tworzonego systemu robotycznego,
- systemów składanych z dobrze przetestowanego oprogramowania, ale stworzonego w starej technologii.

Większość programowych struktur ramowych wykorzystuje podejście do ich konstrukcji rodem z inżynierii oprogramowania, w niewielkim stopniu opierając się na wiedzy związanej z dziedziną, w której mają być stosowane, czyli robotyką. Niewiele z nich wskazuje projektantowi za pomocą swoich wzorców, architekturę systemu sterowania specyficzną dla robotów. Oczywiście postulowana architektura nie powinna ograniczać swobody projektanta w zakresie funkcji, które system powinien realizować, ale ułatwiać podjęcie decyzji co do właściwej struktury systemu.

W przeważającej liczbie przypadków programowe struktury ramowe stosowane w robotyce powstają na bazie istotnego doświadczenia zdobytego przez ich twórców przy konstruowaniu oprogramowania różnego typu. Innymi słowy, proponowane wzorce są wynikiem doświadczenia i intuicji ich twórców. Należy jednak odnotować, że prowadzone są również prace teoretyczne, które dążą do zaprojektowania ogólnych wzorców przez formalną ich specyfikację na podstawie głębokiej znajomości dziedziny zastosowań, do której się odnoszą [68, 76, 75, 70, 41]. Tak powstałe wzorce wykorzystywane są następnie w tworzonej programowej strukturze ramowej. Taka formalna specyfikacja może być przekształcona w metajęzyk, który umożliwia zwięzłą definicję struktury sterownika. Wtedy kompilator metajęzyka generuje program sterujący na podstawie tej definicji. Przykładem tego podejścia może być GenoM [30, 4]. Ta technika jest obecnie rozwijana przez formalne definiowanie ogólnego modelu systemu, który ma powstać. Prace związane z tym podejściem kryją się pod zbiorczymi angielskimi hasłami: *Model Driven Architecture* lub *Model Driven Engineering* (podejście wykorzystujące model dziedziny) [27, 15]. Niektórzy autorzy podkreślają istotne trudności, które trzeba pokonać na drodze do stworzenia uniwersalnej struktury ramowej dla robotyki [16]. Trudności te wynikają z faktu, że systemy robotyczne zazwyczaj są heterogeniczne, ich konstrukcja wymaga wiedzy z wielu dziedzin, roboty są urządzeniami wielozadaniowymi i wielopostaciowymi, przyjmującymi różne formy i działającymi w różnych środowiskach. Ponadto zarówno sam system jak i jego otoczenie wymagają reprezentacji na różnych poziomach abstrakcji. Niemniej jednak brak odpowiednich wzorców lub nadmierna swoboda wynikająca z przesadnej ogólności i oderwania się od dziedziny zastosowań uważane są za cechy negatywne niektórych struktur ramowych [23]. Warto też nadmienić, że podejście wykorzystujące model dziedziny może też być stosowane do tworzenia specjalizowanych języków programowania [38].

2. Wybrane programowe struktury ramowe do tworzenia sterowników robotów

Dalsza część artykułu jest poświęcona opisowi szczególnych cech wybranych, bardziej popularnych współczesnych struktur ramowych. Należy pamiętać, że wiele z opisanych tu programowych struktur ramowych jest nadal w fazie tworzenia, modyfikacji i eksperymentowania z różnymi rozwiązaniami technicznymi [77]. Dlatego podany tu opis należy traktować jako ilustrację różnych podejść i pomysłów, a nie jako kompendium wiedzy na temat stabilnych już rozwiązań.

2.1. Player, Stage, Gazebo

Struktura ramowa Player [31, 24, 58] powstała na University of Southern California. W przypadku jej stosowania układy sterowania robotów konstruowane są z tzw. sterowników sprzętowych (ang. *driver*), komunikujących się ze sobą przez predefiniowane, osobno kompilowane interfejsy. Programy tworzone za pomocą Playera składają się z modułów bezpośredniego dostępu do sprzętu robota mobilnego (takich jak np. układ napędowy, dalmierze laserowe, kamera) oraz klientów, z których jeden zazwyczaj pełni rolę sterownika nadrzędnego, realizującego zadanie sformułowane dla robota przez użytkownika. Jest to typowa architektura klient-serwer. Sterownik nadrzędny (klient) łączy się z serwerem urządzeń za pośrednictwem prostego protokołu wykorzystując połączenie TCP/IP. Sterowniki sprzętowe, stanowiące serwer, można łączyć hierarchicznie. Podstawową zaletą stosowania prostej komunikacji TCP/IP jest możliwość komunikowania się z robotem przez klientów napisanych z wykorzystaniem różnych języków programowania, np. C, C++, Java, Python, Ruby, Lisp oraz MATLAB. Jednocześnie możliwość tworzenia sterowników sprzętu, jako wątków wykonywanych w ramach procesu serwera, ograniczona jest do języka C++. Komunikacja między sterownikiem nadrzędnym i sterownikami sprzętu odbywa się w postaci kolejki komunikatów. Zadaniem serwera jest ich obsługa i przekazywanie danych między sterownikami sprzętu a klientami połączonymi protokołem TCP/IP. Podstawowy dostęp do urządzeń w systemach stworzonych za pomocą Playera wykorzystuje standardowe mechanizmy komunikacji systemu operacyjnego UNIX, tzn. wszelkie operacje realizowane są jako zapis (*write*), odczyt (*read*), sterowanie urządzeniem (*ioctl*). Obecnie standardowe traktowanie sprzętu przez system UNIX, jako plików, jest rozszerzone na inne formy komunikacji.

Zaletą tej struktury ramowej jest przedstawienie sprzętu, który wchodzi w skład robota, takiego jak czujniki i silniki, w postaci użytecznej dla twórcy oprogramowania realizującego zadanie zlecone przez użytkownika. Przykładowo, sterownik

sprzętowy manipulatora może go reprezentować przez pozycję końcówki wyrażoną w przestrzeni kartezjańskiej, a nie jako wektor położeń rotorów silników. Dodatkową zaletą jest intuicyjny model komunikacji klient-serwer oraz przejrzysty interfejs sterowania robotem. Natomiast wadami architektury systemów powstających na bazie oprogramowania Player są ograniczenia związane z transportem sieciowym, a w szczególności niedeterminizm czasowy oraz wzajemna komunikacja wszystkich modułów przez tę samą sieć, co może stanowić wąskie gardło. W zaproponowanej koncepcji brak jest mechanizmów zapewniających funkcjonowanie w czasie rzeczywistym. Z tego też względu użycie tego oprogramowana ograniczone jest do stosunkowo mało wymagających zastosowań w robotyce mobilnej, gdzie zarówno gwarancje czasów wykonania jak i częstotliwość wymiany informacji nie odgrywają kluczowej roli. Swoją popularność na gruncie robotyki mobilnej Player zawdzięcza przede wszystkim dostępności gotowych do użycia sterowników sprzętowych dla najbardziej typowych czujników (jak np. skanery laserowe) oraz istnieniu stowarzyszonych symulatorów graficznych: Stage (do symulacji środowisk dwuwymiarowych) oraz Gazebo (symulujący środowiska trójwymiarowe z modelowaniem pełnej dynamiki obiektów). Player jest klasycznym przykładem programowej struktury ramowej koncentrującej się na standaryzacji komunikacji między komponentami systemu oraz odpowiedniej reprezentacji sprzętu. Ta struktura ramowa korzysta z klasycznej architektury klient-serwer, nie odwołując się do dziedziny, jaką jest robotyka, pomimo że dla niej została powołana do życia. Brak jest tu wzorców architektonicznych sugerujących, jaką strukturę powinny mieć układy sterowania robotów.

2.2. OROCOS

Player przez wiele lat dominował na polu robotyki mobilnej, głównie za sprawą dużej liczby sterowników oraz symulatorów Stage oraz Gazebo, umożliwiających rozwijanie oraz testowanie programów użytkowych w środowisku symulacyjnym. Z kolei w zadaniach sterowania manipulatorami od wielu lat doskonale sprawdza się struktura ramowa OROCOS (ang. *Open RObot COntrol Software*) [19–21]. Za pomocą tej struktury ramowej systemy sterujące składane są z niezależnych modułów zwanych komponentami. Aby komponent mógł przejawić aktywność, musi być pobudzony. Do tego celu służą obiekty nazywane „aktywnościami" (ang. *Activity*). Komponenty przypisywane są aktywnościom, które mogą je pobudzać w następujący sposób:

- Cyklicznie (ang. Periodic Activity) – wtedy w wątku związanym z aktywnością periodycznie pobudzany jest do działania egzekutor komponentu (działanie synchroniczne),
- Niecyklicznie (ang. Non-Periodic Activity) – wtedy w wątku związanym z aktywnością pobudzany jest do działania egzekutor komponentu, ale jedynie gdy pojawią się nowe dane w jego portach wejściowych (działanie asynchroniczne związanie ze zdarzeniem),
- Pośrednio (ang. Master-Slave Activity) – wtedy w wątku związanym z komponentem nadrzędnym (Master) pobudzany jest do działania egzekutor komponentu podrzędnego (Slave) – to komponent nadrzędny decyduje kiedy egzekutor komponentu podrzędnego ma być uruchomiony,
- IRQActivity – egzekutor komponentu jest uruchamiany jako obsługa przerwania sprzętowego.

Egzekutor komponentu, gdy tylko zostanie pobudzony, wykonuje wszystkie asynchroniczne operacje i dołączone funkcje związane z tym komponentem oraz funkcje użytkownika przetwarzające dane z portów wejściowych na wartości wstawiane do portów wyjściowych komponentu. Ze wspomnianymi wątkami związane są priorytety. Funkcje mogą być grupowane i uruchamiane przez automaty skończone. Automat zleca egzekutorowi wykonanie zestawu funkcji przypisanych aktualnemu stanowi automatu. Przy każdym pobudzeniu egzekutora sprawdzany jest warunek przejścia do kolejnego stanu automatu, jeżeli jest on spełniony to następuje przejście do kolejnego stanu, określonego przez tablice przejść automatu, a więc i do wykonania innego zestawu funkcji.

Przez ponad dekadę rozwoju struktury ramowej OROCOS powstało szereg wzorców programów sterujących robotami. Wzorce te są składane z komponentów umożliwiających realizację często spotykanych w robotyce obliczeń takich jak: planowanie ścieżek, sterowanie pozycyjne, dostęp do sprzętu czy monitorowanie stanu systemu. Pojedynczy komponent może sterować wszystkimi funkcjami maszyny albo być odpowiedzialny jedynie za realizację jakichś specyficznych obliczeń (np. rozwiązanie prostego lub odwrotnego zagadnienia kinematyki).

Istotnym aspektem każdej struktury ramowej jest sposób wewnętrznego komunikowania się modułów. Komponenty OROCOS mogą korzystać z następujących sposobów wymiany danych:

- przez porty, przekazując w sposób nieblokujący dane, które mogą być buforowane lub nie,
- przez zmianę właściwości komponentów (parametry zapisane są w plikach XML),
- przez wywołanie metod,
- przez wysłanie polecenia.

Dany komponent nie musi korzystać ze wszystkich wymienionych sposobów komunikacji. Wtedy jego interfejs, definiujący sposób komunikacji z innymi komponentami, może być uproszczony.

Sam wzorzec komponentu zakłada istnienie w jego wnętrzu hierarchicznego automatu skończonego. Automat hierarchicznie wyższy określa stan wykonania, taki jak: przedoperacyjny, zatrzymany, wykonywany, wyjątkowy. Automat hierarchicznie niższy (jeżeli istnieje) w każdym ze swych stanów realizuje odpowiadające mu funkcje komponentu. Budowa programu sterującego robotem sprowadza się do zestawienia odpowiednich modułów obliczeniowych oraz odpowiedniego ich skomunikowania ze sobą. Możliwa jest również dystrybucja komponentów między procesy, potencjalnie wykonywane na różnych komputerach połączonych w sieć. Wtedy do wzajemnej ich komunikacji wykorzystywana może być CORBA (Common Object Request Broker Architecture) [3], POSIXowe kolejki komunikatów (jeżeli komunikujące się procesy rezydują na jednej maszynie) lub inne mechanizmy komunikacji dostarczone przez użytkownika. Generalnie architektura powstającego systemu jest pozostawiona inwencji jego twórcy – brak jest wzorców wynikających z dziedziny zastosowań, jaką w tym przypadku jest robotyka. Odniesienie do robotyki związane jest z rodzajem komponentów zawartych w wymienionych bibliotekach.

2.3. YARP

Struktura ramowa YARP (Yet Another Robot Platform) [29, 44] jest zestawem bibliotek, protokołów i narzędzi umożliwiających prostą dekompozycję systemu na niezależnie działające moduły, przy czym nie narzuca ona konkretnych sposobów łączenia tych modułów. Najczęściej była stosowana do tworzenia oprogramowania sterującego robotami humanoidalnymi (powstała jako otwarte oprogramowanie w ramach projektu iCub). Jej podstawowym składnikiem jest biblioteka klas z ogólnie dostępnymi źródłami napisanymi w języku C++. Biblioteka ta składa się z trzech podstawowych części: interfejsów do różnych systemów operacyjnych umożliwiających komunikację międzyprocesową, algorytmów przetwarzania sygnałów (obrazu i mowy) oraz interfejsów do typowych urządzeń używanych w robotach [28]. Tworzone oprogramowanie sterujące robotem przyjmuje ogólną strukturę typu klient-serwer. Ponieważ złożoność algorytmów sterowania robotów humanoidalnych wymaga zastosowania obliczeń rozproszonych realizowanych na wielu procesorach, dlatego twórcy struktury ramowej YARP skoncentrowali się na abstrakcyjnym modelu komunikacji między-

procesowej, który jest niezależny od specyficznego mechanizmu transportowego, czyli rodzaju sieci i protokołu. W konsekwencji komunikacja jest realizowana przez tzw. połączenia między aktywnymi obiektami zwanymi portami, które mogą znajdować się na różnych maszynach działających pod kontrolą różnych systemów operacyjnych (Linux, Solaris, QNX6, Windows, Mac OSX). Komputery te tworzą sieć YARP modelowaną jako graf skierowany, którego węzłami są porty, a krawędziami połączenia. Każdy port ma unikatową nazwę, pod którą jest rejestrowany w serwerze nazw. Dany port może obsługiwać wiele połączeń z innymi portami. Przyporządkowuje mu się kierunek transmisji, a więc może być wejściowy lub wyjściowy, odpowiednio wysyłając lub odbierając dane z wielu połączeń z różną częstością, wykorzystując różne protokoły komunikacyjne (TCP, UDP, QNet lub wspólną pamięć). Komunikacja jest całkowicie asynchroniczna bez gwarancji dostarczenia komunikatu, lecz zakłada się, że komunikaty są powtarzane i często wysyłane, zatem utrata jednego komunikatu nie ma istotnego wpływu na działanie systemu. Dla zapewnienia pracy w czasie rzeczywistym narzut sieciowy jest minimalizowany, dlatego też sieć YARP działa jako izolowana sieć lokalna – oczywiście jest to działanie bez gwarancji. Dostęp do sterowników urządzeń następuje przez interfejsy zdefiniowane dla konkretnej klasy urządzeń. Dzięki temu wymiana urządzenia na inne z danej klasy, niepociągająca zmiany interfejsu, nie powoduje modyfikacji programu klienta. Oprogramowanie stworzone za pomocą YARP wykorzystano do sterowania takich robotów jak Kismet, Cog, Domo oraz oczywiście iCub.

2.4. ORCA

ORCA jest programową strukturą ramowa wykorzystującą paradygmat programowania komponentowego. Jest ona przeznaczona dla robotów mobilnych [12, 11]. Szyperski [57] definiuje komponenty jako niezależnie wytwarzane, nabywane i instalowane moduły binarne, które współdziałając tworzą system. Istotne jest, że producent komponentu nie musi ujawniać jego użytkownikowi kodu źródłowego. Oznacza to, że komponenty muszą mieć ściśle zdefiniowane interfejsy (definiujące ich sposób użycia oraz zachowanie). Często zachowanie komponentu może być dostosowywane do potrzeb użytkownika, ale nie dzieje się to przez modyfikację kodu źródłowego, ale przez rekonfigurację w trakcie pracy (komponenty są odpowiednio parametryzowane) [17].

Przy wykorzystaniu struktury ramowej ORCA systemy składane są z gotowych oraz dodatkowo tworzonych komponentów, w opozycji do tradycyjnego podejścia opierającego się na tworzeniu kodu źródłowego programu. Komponenty struktury ramowej ORCA implementowane są jako procesy, które komunikują się przekazując sobie dane. ORCA określa wzorce komunikacji między procesami. Fizyczny mechanizm transportu danych początkowo był realizowany za pośrednictwem CORBA [51, 56], gniazd TCP/IP, a obecnie za pomocą biblioteki Ice (Internet Communication Engine) [2]. Z użytkowego punktu widzenia komponenty Orca stanowią algorytmy obliczeniowe oraz interfejsy sprzętowe. Mogą się one komunikować, jeżeli tylko ich interfejsy pasują do siebie, a więc jeżeli wzorzec komunikacji oraz typ przekazywanych danych są zgodne. To podejście zadecydowało o tym, że ORCA koncentruje się na wzorcach komunikacji, a nie sposobach tworzenia komponentów. Funkcje poszczególnych komponentów, ich granulacja i dobór zostały zostawione indywidualnej decyzji twórcy systemu i muszą uwzględniać opóźnienia wnoszone przez mechanizmy komunikacji oraz algorytmy realizowane przez komponenty. Zaletą tego podejścia jest duża elastyczność dotycząca architektury systemu, wadą natomiast brak wskazówek co do sprawdzonych rozwiązań architektonicznych. Orca odwołuje się do robotyki jedynie przez algorytmy zawarte w komponentach. Komponenty realizują algorytmy niezbędne do działania robota, natomiast struktura połączeń między komponentami nie wynika ze wzorców robotycznych.

2.5. Microsoft Robotics Studio

Komercyjne narzędzie Microsoft Robotics Studio MSRS [26, 23] swego czasu zyskało dość dużą popularność wśród hobbystów zajmujących się robotyką. MSRS stanowi zintegrowane środowisko do symulacji i programowania robotów. Umożliwia ono tworzenie oprogramowania zorientowanego na usługi (realizowane przez komponenty). Jako mechanizm komunikacji wykorzystywana jest biblioteka CCR (Concurrency and Coordination Runtime). Umożliwia ona budowanie wydajnych programów wielowątkowych, gdzie usługa definiowana jest jako zbiór wywołań funkcji (delegatur) w reakcji na dane napływające do tzw. portów. Możliwe jest definiowanie różnych schematów obsługi portów, jak np. blokowanie wykonania w oczekiwaniu na nadejście nowych danych z kilku portów [22]. Wymiana danych między komponentami odbywa się za pomocą otwartego protokołu DSS (Decentralized Software Services), wykorzystującego kodowanie danych w formacie XML i transmisję przy użyciu protokołu HTTP. Programowanie możliwe jest w językach C#, Visual Basic .NET, C++/CLI oraz Iron Python, które korzystają z platformy .NET. MSRS przystosowane jest do wykorzystania komercyjnego przez dodawanie pakietów rozszerzających możliwości systemu.

Przykładami darmowych pakietów są rozszerzenia do symulacji ligi robotów RoboCup oraz zawodów Sumo, dodatkowo wybrani producenci robotów przemysłowych oferują pakiety do symulacji i sterowania swoich produktów. Oczywiście powstałe oprogramowanie musi pracować pod kontrolą systemu operacyjnego z rodziny Windows, a w konsekwencji MSRS nie umożliwia tworzenia systemów działających w reżimie czasu rzeczywistego.

2.6. ROS

Analogiczne podejście (tzn. brak określonej z góry struktury sterownika) stosowane jest w jednej z najnowszych struktur ramowych do tworzenia rozproszonych, wieloprocesowych sterowników robotycznych: ROS (Robot Operating System) [55], opracowanym na Uniwersytecie Stanforda oraz przez wiele lat rozwijanym przez firmę Willow Garage. ROS jest obecnie jednym z najpopularniejszych, a zarazem najbardziej zaawansowanych narzędzi do tworzenia rozproszonych, wieloprocesowych sterowników robotycznych.

W ROS poszczególne elementy sterownika działają jako niezależne procesy w sieci (zwane węzłami, od ang. *Node*), a system dostarcza mechanizmów przesyłania komunikatów, zarządzania pakietami oraz monitorowania i wizualizacji stanu poszczególnych węzłów, kanałów komunikacyjnych, jak i stworzonego systemu. ROS, podobnie jak Orca, koncentruje się na wzorcach komunikacji, a nie na sposobach tworzenia modułów czy zestawiania modułów w sterownik.

ROS oferuje szereg mechanizmów służących do komunikacji i wymiany danych między poszczególnymi węzłami. Pierwszym z nich są kanały komunikacyjne zwane tematami (ang. *Topic*). Tematy oparte są na wzorcu projektowym publikacji subskrypcji oraz umożliwiają jednokierunkowe przesyłanie danych od nadawcy do odbiorcy, przy czym zarówno jednych jak i drugich może być dowolna liczba (komunikacja wiele do wielu). Głównym zastosowaniem tematów jest komunikacja strumieniowa następująca cyklicznie (np. wysyłanie kolejnych danych sensorycznych do przetworzenia). Tematy mają również mechanizmy synchronizacji wywoływania funkcji obsługi z danymi pojawiającymi się na wejściach kanałów komunikacyjnych danego węzła. Mechanizm ten łączy daną funkcję z listą wejść, dzięki czemu zostanie ona automatycznie wywołana, gdy dane znajdujące się w zarejestrowanych buforach będą spełniały odpowiednią zależność. Zależność ta może być zdefiniowana przez programistę. Dostępne są też dwie predefiniowane możliwości: wywoła-

nie funkcji w momencie, gdy wszystkie dane mają taka samą sygnaturę czasową bądź w momencie, gdy sygnatury te różnią się nie więcej niż o ustalony interwał. Kolejność wywołania funkcji w przypadku, gdy kilka z nich może być uruchomione (np. pokrywają się ich zestawy wejść) jest nieokreślona, nie ma również możliwości sprawdzenia, czy konkretne dane zostały juz wykorzystane w ramach bieżącego cyklu pracy komponentu.

Drugim mechanizmem umożliwiającym wymianę danych między węzłami są serwisy (ang. *Services*), które w odróżnieniu od tematów są synchronicznym mechanizmem wymiany danych. Poza wiadomością przesyłaną od nadawcy do odbiorcy oraz odsyłaną odpowiedzią, serwisy umożliwiają wywołanie odpowiedniej funkcji obsługi (a więc oferują mechanizm zdalnego wywołania funkcji, RPC).

Ponieważ wykonanie zdalnie uruchomionej funkcji może długo trwać, dlatego w ROS zaimplementowano trzeci mechanizm komunikacji zwany akcją (ang. *Action*), który umożliwia wywołanie funkcji bez konieczności zawieszania procesu zlecającego na czas jej wykonania. Akcje umożliwiają pytanie o stan wykonania wywołanej procedury, a także przerwanie jej wykonywania.

Poza wymienionymi ROS ma wbudowane mechanizmy do obsługi szeroko pojętej konfiguracji systemu. Najważniejszym z nich jest serwer parametrów, dający globalny dostęp do parametrów wszystkim węzłom ROS.

2.7. MRROC++

Układy sterowania systemami wielorobotowymi tworzone z wykorzystaniem programowej struktury ramowej MRROC++ (Multi-Robot Research Oriented Controller based on C++) [63, 77] mają hierarchiczną strukturę. Struktura ramowa MRROC++ powstała na Politechnice Warszawskiej [63, 73, 65-67]. W jej skład wchodzą: biblioteka modułów (klas, obiektów, procesów, wątków i procedur) oraz wzorce ich wykorzystania. Z modułów tych można skonstruować sterownik dowolnego systemu wielorobotowego (wieloefektorowego, gdzie efektorem jest dowolne urządzenie oddziałujące na środowisko). Zestaw modułów bibliotecznych może być rozszerzony przez utworzenie nowych modułów w języku C++. Niezależnie, czy nowy układ sterowania będzie miał za zadanie sterowanie pojedynczym efektorem, czy ich grupą, powstanie system z koordynatorem, a więc o strukturze umożliwiającej sterowanie wieloma efektorami. Ponadto zarówno koordynator, jak i poszczególne sterowniki efektorów, mogą być wyposażone w swoją własną grupę eksteroreceptorów (czujników zbierających informacje o stanie otoczenia robota) oraz środki do komunikacji z innymi systemami tego typu.

Współpracujące ze sobą procesy podzielono na warstwy. Każdej z warstw przyporządkowano ściśle określone funkcje – wyróżniono procesy zależne od sprzętu oraz zależne od zadania. Układy sterowania powstające na bazie MRROC++ składają się z następujących procesów:

UI – User Interface Process zależny jest od konfiguracji systemu; odpowiedzialny jest za komunikację z operatorem; tylko jeden taki proces egzystuje w systemie; w systemach w pełni autonomicznych nie jest on potrzebny, ale w systemach do celów badawczych, ulegających częstym modyfikacjom, podczas których mogą się wkraść błędy, jest on wręcz nieodzowny,

MP – Master Process koordynuje prace wszystkich efektorów systemu; system ma tylko jeden taki proces; proces ten może być uśpiony – wtedy składowe procesy ECP nie są koordynowane,

ECP – Effector Control Process odpowiedzialny jest za realizację zadania zleconego efektorowi; system ma tyle takich procesów, ile efektorów wchodzi w jego skład,

EDP – Effector Driver Process jest odpowiedzialny za bezpośrednie sterowanie efektorem; liczba procesów EDP równa jest liczbie procesów ECP,

VSP – Virtual Sensor Process jest odpowiedzialny za agregację danych z eksteroreceptorów – w ten sposób powstaje odczyt czujnika wirtualnego; zero lub więcej procesów tego typu może być skojarzonych z dowolnym ECP lub MP; każdy proces VSP oblicza wartość funkcji agregującej dane z dołączonych do niego czujników.

Każdy z tych procesów może składać się z wątków. Działanie procesów MP oraz ECP opisywane jest automatem skończonym, który w każdym ze swych stanów aktywuje odpowiednie zachowanie. Obecnie podsystemy percepcyjne, realizowane za pomocą czujników wirtualnych (VSP), tworzone są z wykorzystaniem dodatkowej struktury ramowej dedykowanej właśnie zadaniom percepcyjnym – DisCODe [40].

Wszystkie procesy MRROC++ mają jednakową strukturę ogólną. Składają się z powłoki oraz jądra. Powłoka jest odpowiedzialna za komunikację międzyprocesową, a ponadto za obsługę wykrytych błędów. Natomiast jądro zawiera kod stworzony przez użytkownika. Jest to kod związany z realizowanym zadaniem. Powłoka stanowi w dużej mierze niezmienną część procesu, natomiast jądro jest częścią wymienialną zależną od zadania bądź wykorzystywanego sprzętu, a więc dostarczana jest przez twórcę systemu. Komunikacja między procesami rezydującymi na jednym komputerze wykorzystuje przede wszystkim mechanizm spotkań (Send–Receive–Reply) dostępny w systemie operacyjnym czasu rzeczywistego QNX, natomiast komunikacja z procesami znajdującymi się na innych komputerach wykorzystuje protokół TCP/IP.

Dane przekazywane procesom przez ich powłokę trafiają do wewnętrznych struktur danych zwanych buforami, które ulokowane są w jądrach procesów. Funkcje przejścia definiujące zachowanie procesu działają na tych buforach. Obliczenie tych funkcji, dokonanie odpowiednich przesłań danych między powłoką a jądrem oraz powłoką a innymi procesami zajmuje czas – czas ten wyznacza okres pracy procesu (makrokrok). Zazwyczaj jest on wielokrotnością okresu (zwanego krokiem) działania serwomechanizmów realizowanych przez EDP. Dla manipulatorów makrokrok stanowi niewielka wielokrotność kroku (kilka), natomiast dla platform mobilnych może to być większa liczba. Serwomechanizm potrzebuje nowej wartości zadanej co krok swego działania. Zazwyczaj ten krok odmierzany jest przerwaniem zegarowym, a więc jest dość stabilny.

Funkcje definiujące sposób zachowania się procesu umieszczane są w tworze programowym (obiekcie języka C++) zwanym generatorem ruchu. Ów generator stanowi argument instrukcji ruchu Move. Poszczególne instrukcje ruchu przyporządkowane są węzłom grafu (stanom) automatu skończonego odzwierciedlającego sekwencję czynności wykonywanych przez efektor (jeżeli dotyczy to ECP) lub realizowanych przez system wieloefektorowy (jeżeli dotyczy to MP). Programista konstruuje jądro procesu MP lub ECP budując automat skończony z instrukcji Move, wspomagając się dodatkowymi instrukcjami języka C++ – w szczególności instrukcjami sterującymi wykonaniem programu. Tak więc definicja zadania, które ma być zrealizowane, rozbita jest na dwie części:

- zestaw ruchów (akcji), które trzeba wykonać, aby zrealizować zadanie (automat skończony) oraz
- opis sposobu wykonania każdego z tych ruchów – do tego służą generatory ruchu.

Podkreślić należy, że generatory ruchu nie tylko określają zachowanie efektora, ale również determinują interakcje z czujnikami wirtualnymi i zawiadują transmisjami do i z koordynatora (MP). Funkcja przejścia i warunek końcowy określający, kiedy realizacja instrukcji ruchu ma sie skończyć, zapisane są jako metody obiektu zwanego generatorem ruchu. Ponieważ użytkownik dołącza do kodu MRROC++ swój własny kod, więc powstanie błędów jest wysoce prawdopodobne. Obsługa błędów w MRROC++ oparta jest na zgłaszaniu i obsłudze wyjątków. Zarówno kod powłoki jak i kod użytkownika stanowiący jądro procesu mogą i powinny zgłaszać wyjątki, jak tylko wykryją sytuację awaryjną. Zgłoszone wyjątki przechwytywane są przez

powłokę, która stara się poradzić sobie z tą sytuacją najlepiej jak potrafi. Minimalnym wymaganiem jest zgłoszenie operatorowi komunikatu o powstaniu błędu, jego rodzaju oraz postawienie systemu sterującego w taki stan, aby możliwa była jego dalsza praca. Niestety nie jest to zawsze możliwe.

Struktura ramowa MRROC++ była wykorzystana do stworzenia wielu systemów. Z wykorzystaniem MRROC++ powstawały sterowniki np.: robotów przemysłowych [72], przemysłowych systemów wielorobotowych [69], jak i robotów usługowych, np. dwurękiego robota posiadającego wzrok i czucie siły, który układał podana mu kostkę Rubika [74].

2.8. CLARAty

CLARAty (Coupled-Layer Architecture for Robotic Autonomy) jest programową strukturą ramową opracowaną przez Jet Propulsion Laboratory, NASA Ames Research Center, Carnegie Mellon University oraz University of Minnesota, na potrzeby projektu Mars Technology Program [59, 48, 46]. Służy głównie do integracji oprogramowania wielokrotnego użytku tworzonego przez wiele instytucji dla różnego rodzaju pojazdów, w tym manipulatorów mobilnych, których zadaniem jest eksploracja powierzchni innych planet, w szczególności pojazdów już działających bądź przygotowywanych do działania na powierzchni Marsa.

W CLARAty, tak jak w MRROC++, również wydzielono warstwy, dokładnie dwie: decyzyjną oraz funkcjonalną. Warstwa funkcjonalna tworzona jest przez sterowniki urządzeń, natomiast warstwa decyzyjna odpowiedzialna jest za realizację i wykonanie planu, który tworzony jest na podstawie celów określonych przez zadanie sformułowane w kategoriach ograniczeń. Ograniczenia dotyczą dostępnych zasobów oraz czasu realizacji czynności przez poszczególne urządzenia. Podczas planowania stan warstwy funkcjonalnej uzyskiwany jest przez formułowanie i przekazywanie zapytań. Wykonanie planu i jego tworzenie przeplatają się, ponieważ przyjęte ograniczenia zmieniają się w czasie. Odejście od standardowej trójwarstwowej architektury (warstwa sprzętowa, egzekutor planu oraz planer) wynikało ze spostrzeżenia, że tworzenie planu wymaga wiedzy o stanie sprzętu i otoczenia, a więc bezpośredniego dostępu do warstwy sprzętowej. Dlatego powstała jedna warstwa odpowiedzialna za wykonanie i tworzenie planu. Warstwa decyzyjna korzysta jednocześnie z dwóch technik programowania: deklaratywnej i proceduralnej. Jak się później okazało próba połączenia tych dwóch podejść do programowania w jednej warstwie okazała się trudna, więc obecnie sugeruje się jednak podział tej warstwy na dwie [48].

Warstwę funkcjonalną tworzy się korzystając z programowania obiektowego. Obecnie warstwa ta zawiera szereg gotowych obiektów realizujących typowe czynności niezbędne robotom, takie jak lokomocja i manipulacja. Biblioteki zawierają ogólne algorytmy rozwiązywania prostego i odwrotnego zadania kinematyki, wykrywania i unikania kolizji, samolokalizacji, nawigacji, planowania ścieżki itp. [47]. U ich podstaw leżą obiekty umożliwiające realizację podstawowych działań matematycznych wykorzystujących takie obiekty jak: macierze orientacji definiowane za pomocą kątów Eulera, kwaterniony czy macierze przekształcenia jednorodnego. Te twory matematyczne mogą być wykorzystywane do modelowania łańcuchów kinematycznych. Te z kolei służą definiowaniu takich mechanizmów jak manipulatory, nogi maszyn kroczących lub platformy mobilne. Istniejące urządzenia obejmują zarówno sterowniki silników lub karty analogowych i cyfrowych wejść i wyjść, jak i kamery czy urządzenia RGB-D. Ponadto stworzono infrastrukturę do komunikacji wewnątrz jak i między warstwami.

2.9. URBI

URBI (Universal Real-time Behavior Interface) jest wieloplatformową strukturą ramową służącą do tworzenia programów sterujących dla złożonych systemów robotycznych [8, 7]. Oprogramowanie to zostało opracowane przez francuską firmę Gostai [1] i jest udostępniane na licencji AGPL. Bardzo wiele urządzeń wykorzystywanych do budowy systemów robotycznych ma swoje własne sterowniki. Istnieje również wiele użytecznych pakietów, które mogą być wykorzystane do stworzenia np. podsystemów percepcyjnych robotów. Jeżeli takie oprogramowanie chce się zintegrować w jeden spójny system realizujący złożone zadanie, to natrafia się na problem niekompatybilności części składowych. Zadaniem URBI jest rozwiązanie tego bardzo praktycznego problemu. W istocie URBI służy do tworzenia interfejsów do programów napisanych w innych językach, np. C++, Java, MATLAB i wykonywanych pod nadzorem różnych systemów operacyjnych (Linux, Windows, MacOSX).

Komponowanie systemu z dostępnych lub tworzonych modułów ułatwia język skryptowy urbiscript. URBI służy tworzeniu oprogramowania rozproszonego, o architekturze klient-serwer. Język urbiscript umożliwia programowanie zdarzeniowe i zawiera instrukcje służące do tworzenia programów równoległych. Dzięki temu oprogramowanie powstające na bazie URBI powstaje z równoległych i rozproszonych, wyzwalanych zdarzeniami komponentów.

Bibliotekę komponentów nazwano UObject – tak też nazywana jest architektura powstających systemów. Komponenty te można tworzyć samodzielnie, np. w języku C++. Większość tych komponentów służy uzyskaniu dostępu do sprzętu albo opakowuje istniejące w innych bibliotekach oprogramowanie. Poszczególne komponenty mogą pracować synchronicznie albo asynchronicznie – zależy to od konstruktora systemu. Program napisany w urbiscript określa interakcję między komponentami UObject. Stworzone komponenty mogą być wykorzystywane przez urbiscript, jako jego własne obiekty, albo mogą być traktowane jako zdalne obiekty działające pod nadzorem wyżej wspomnianych systemów operacyjnych. URBI zapewnia komunikację komponentów wewnątrz tworzonego systemu i dlatego klasyfikowana jest jako oprogramowanie pośredniczące (ang. *middleware*).

Istnieją wersje tego oprogramowania dla robotów Aibo, robota humanoidalnego HRP-2 oraz robotów mobilnych Pioneer [7]. URBI zostało również wykorzystane do stworzenia układu sterowania robota społecznego FLASH skonstruowanego na Politechnice Wrocławskiej [37, 36].

3. Podsumowanie

Mimo różnorodności opisanych programowych struktur ramowych można wyróżnić pewne ich cechy wspólne. Jedną z nich jest modularność, wynikająca ze standardowego podejścia do projektowania dużych systemów, polegającego na ich dekompozycji na mniejsze, współpracujące ze sobą elementy. Niezależnie, czy są to sterowniki w Player, moduły w CLARAty, procesy w ROS, czy komponenty w OROCOS, elementy tego typu umożliwiają powtórne użycie w różnych programach, a przez to również zwiększają niezawodność powstającego oprogramowania (elementy te są lepiej przetestowane, przy czym warte podkreślenia jest, iż mogą one być testowane niezależnie) oraz skraca czas projektowania i implementacji konstruowanego systemu. Podejście to, mimo wymienionych zalet, ma również wady, wśród których jedną z najwazniejszych są problemy z integracją. Przykładowo w ROS integracja podukładów sensorycznych (a w szczególności podsystemów wizyjnych) okazała się dla wielu użytkowników problemem, i to mimo elastyczności tej struktury ramowej oraz posiadanych różnorodnych mechanizmów komunikacji. Z tego powodu autorzy ROS opracowali strukturę ramową Ecto [60], gdzie uwaga została skupiona na organizacji obliczeń i przepływie danych w ramach jednego

procesu – zauważono bowiem, że programiści, aby zmniejszyć narzuty komunikacyjne, implementowali podsystemy percepcji (a szczególnie percepcji wizyjnej) w ramach jednego procesu.

Innym problemem jest konfiguracja systemu, a więc umożliwienie łatwej zmiany wartości parametrów niezbędnych do właściwej pracy poszczególnych elementów układu. W Player, OROCOS, ROS oraz MRROC++ istnieją mechanizmy wczytywania ustawień z plików konfiguracyjnych, jednak zarówno w przypadku układów wizyjnych jak i serwomechanizmów, podczas ich testowania z reguły potrzebne jest ich ręczne dostrajanie (zmiana parametrów regulatora, modyfikacja progu binaryzacji, ręczna selekcja wymaganych cech, etc.). Był to jeden z powodów wydzielenia z MRROC++ wyspecjalizowanych programowych struktur ramowych do implementacji podsystemów wizyjnych: wpierw projektu FraDIA (Framework for Digital Image Analysis) [39], która później wyewoluowała we wspomnianą wcześniej komponentową strukturę ramową DisCODe [40].

Dodatkowo bardzo ważnym aspektem, na który często nie jest zwracana uwaga, jest organizacja samego projektu, a więc sposób zarządzania modułami oraz właściwym jego trzonem. W przypadku tak dużych systemów, jakimi są układy sterowania robotów, odpowiednia organizacja jest niezbędna, aby umożliwić pracę większej liczbie użytkowników i deweloperów. Brak niezależności między trzonem a modułami oraz zadaniami powoduje znaczne problemy z wykorzystaniem danego narzędzia w innym laboratorium – każde laboratorium dysponuje z reguły unikalnym zestawem sprzętowym. Dekompozycja samego repozytorium, gdzie przechowywane są kody źródłowe, związana jest więc ze skalowalnością projektu. Przykładowo, trzon struktury ramowej Player przechowywany był we wspólnym repozytorium plików razem ze wszystkimi sterownikami. Dlatego też jego autorzy, rozpoczynając pracę nad system ROS, postanowili rozdzielić te repozytoria, przy czym poszli o krok dalej od twórców struktury ramowej OROCOS (gdzie wydzielono trzon, a komponenty przechowywane są w jednym repozytorium OCL, od OROCOS Component Library) i zaproponowali rozproszoną organizację repozytoriów, dzieląc je na stosy oraz pakiety oddzielne dla poszczególnych użytkowników.

Innym niedocenianym, a niezmiernie ważnym czynnikiem decydującym o sukcesie lub porażce danej struktury ramowej, mierzonym liczbą jej użytkowników, jest jakość dokumentacji dostarczanej wraz z oprogramowaniem. W położeniu nacisku na ten aspekt można szukać jednego z powodów sukcesu struktury ramowej ROS, która obecnie zaczyna być uznawana za standard przez coraz większy krąg robotyków.

Różnego rodzaju struktur ramowych powstało dziesiątki. Ten przegląd miał na celu przedstawienie tych bardziej powszechnie używanych oraz tych, których cechy mają szanse pojawić sie w nowych strukturach ramowych. Nie ulega wątpliwości, że struktury ramowe dla robotów będą podlegały dalszemu rozwojowi. Zapewne przyszłe rozwiązania w istotniejszy sposób będą koncentrowały się na dziedzinie, dla której zostały stworzone, a więc na robotyce.

Bibliografia

1. Strona projektu URBI. URBI [www.gostai.com].
2. Internet Communication Engine. [http://zeroc.com/ice.html], 2008.
3. CORBA Basics. [www.omg.org/gettingstarted/corbafaq.htm], 2014.
4. Alami R., Chatila R., Fleury S., Ghallab M.M., Ingrand F., *An architecture for autonomy. Int. J. of Robotics Research*, 17(4):315–337, 1998.
5. Ambler A.P., Corner D.F., RAPT1 user's manual. Department of Artificial Intelligence, University of Edinburgh, 1984.
6. Backes P., Hayati S., Hayward V., Tso K., *The kali multi-arm robot programming and control environment. NASA Conference on Space Telerobotics*, 89–7, 1989.
7. Baillie J.-C., *Design principles for a universal robotic software platform and application to URBI. IEEE ICRA 2007 Workshop on Software Development and Integration in Robotics (SDIR-II)*. IEEE Robotics and Automation Society, 2007.
8. Baillie J.-C., Nottale M., Pothier B., The URBI Tutorial v.1.5. [www.urbiforge.org/tutorial], 2007.
9. Blume C., Jakob W., *PASRO: Pascal for Robots*. Springer-Verlag, 1985.
10. Blume C., Jakob W., *Programming languages for industrial robots*. 1986.
11. Brooks A., Kaupp T., Makarenko A., Williams S., Orebäck A., Towards componentbased robotics. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, 163–168, August 2005.
12. Brooks A., Kaupp T., Makarenko A., Williams S., Orebäck A., *Orca: A component model and repository*. D. Brugali, (red.), *Software Engineering for Experimental Robotics*, wolumen 30 serii *Springer Tracts in Advanced Robotics*, 231–251. Springer, 2007.
13. Broten G., Monckton S., Giesbrecht J., Collier J., *Towards framework-based uxv software systems: An applied research perspective*. D. Brugali, (red.), *Software Engineering for Experimental Robotics*, 365–393. Springer-Verlag, 2007.
14. Brugali D., *Sidebar – middlewares for distributed computing*. D. Brugali, (red.), *Software Engineering for Experimental Robotics*, strony 395–398. Springer-Verlag, 2007.
15. Brugali D., *Stable analysis patterns for robot mobility*. Brugali D., (red.), *Software Engineering for Experimental Robotics*, 9–30. Springer-Verlag, 2007.
16. Brugali D., Agah A., MacDonald B., Nesnas I., Smart W., *Trends in robot software domain engineering*. D. Brugali (red.) *Software Engineering for Experimental Robotics*, 3–8. Springer-Verlag, 2007.
17. Brugali D., Brooks A., Cowley A., Côté C., Domínguez-Brito A.C., Létourneau D., Michaud F., Schlegel C., *Trends in component-based robotics*. D. Brugali, (red.), *Software Engineering for Experimental Robotics*, wolumen 30 serii *Springer Tracts in Advanced Robotics*, 135–142. Springer-Verlag, 2007.
18. Brugali D., Broten G.S., Cisternino A., Colombo D., Fritsch J., Gerkey B., Kraetzschmar G., Vaughan R., Utz H., *Trends in robotic software frameworks*. D. Brugali, redaktor, *Software Engineering for Experimental Robotics*, 259–266. Springer-Verlag, 2007.
19. Bruyninckx H., *Open robot control software: the orocos project. International Conference on Robotics and Automation (ICRA)*, wolumen 3, 2523–2528. IEEE, 2001.
20. H. Bruyninckx. OROCOS – Open Robot Control Software. [www.orocos.org], 2002.
21. H. Bruyninckx. *The real-time motion control core of the OROCOS project. Proceedings of the IEEE International Conference on Robotics and Automation*, 2766–2771. IEEE, September 2003.
22. Chrysanthakopoulos G., Singh S., An asynchronous messaging library for C#.
23. Cisternino A., Colombo D., Ambriola V., Combetto M., *Increasing decoupling in the robotics4.net framework*. D. Brugali, redaktor, *Software Engineering for Experimental Robotics*, 307–324. Springer-Verlag, 2007.
24. Collett T., MacDonald B., Gerkey B., *Player 2.0: Toward a practical robot programming framework. Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, December 2005.
25. Corke P., Kirkham R., *The ARCL robot programming system*. 484–493. 14–16 July 1993.
26. Corporation M., *Microsoft Robotics Studio*.

27. Czarnecki K., Helsen S., Classification of model transformation approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, wolumen 45, 1–17. Citeseer, 2003.
28. Fitzpatrick P., Metta G., Natale L., *YARP User Manual*, 2007.
29. Fitzpatrick P., Metta G., Natale L., Towards long-lived robot genes. *Robotics and Autonomous Systems*, 56(1):29–45, 2008.
30. Fleury S., Herrb M., Chatila R., GenoM: A tool for the specification and the implementation of operating modules in a distributed robot architecture. *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'97)*, 2:842–849, September 1997.
31. Gerkey B.P., Vaughan R. T., Støy K., Howard A., Sukhatme G. S., Mataric M. J., Most Valuable Player: A Robot Device Server for Distributed Control. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1226–1231, 2001.
32. Hayward V., Daneshmend L., Hayati S., An overview of KALI: A system to program and control cooperative manipulators. K. Waldron, redaktor, *Advanced Robotics*, 547–558. Springer–Verlag, Berlin, 1989.
33. Hayward V., Hayati S., *Kali: An environment for the programming and control of cooperative manipulators. 7th American Control Conference*, 473–478, 1988.
34. Hayward V., Paul R.P., *Robot manipulator control under unix RCCL: A robot control C library. International Journal of Robotics Research*, 5(4):94–111, Winter 1986.
35. Kaisler S., *Software Paradigms*. Wiley Interscience, 2005.
36. Kedzierski J., Janiak M., *Budowa robota społecznego FLASH*. Tchon K., Zielinski C., redaktorzy, *Postępy Robotyki*, wolumen 182 serii *Prace Naukowe – Elektronika*, 681–694. Oficyna Wydawnicza Politechniki Warszawskiej, 2012.
37. Kędzierski J., Małek Ł., Oleksy A., *Zastosowanie otwartego oprogramowania w systemie sterowania robotem społecznym*. Tchoń K., Zieliński C., redaktorzy, *Postępy Robotyki*, wolumen 182 serii *Prace Naukowe – Elektronika*, 671–680. Oficyna Wydawnicza Politechniki Warszawskiej, 2012.
38. Kleppe A., *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, 2009.
39. Kornuta T., *Application of the FraDIA vision framework for robotic purposes*. Bolc L., Tadeusiewicz R., Chmielewski L., Wojciechowski K., redaktorzy, *Proceedings of the International Conference on Computer Vision and Graphics, Part II*, wolumen 6375 serii *Lecture Notes in Computer Science*, strony 65–72. Springer Berlin/Heidelberg, 2010.
40. Kornuta T., Stefańczyk M., DisCODe: komponentowa struktura ramowa do przetwarzania danych sensorycznych. *Pomiary Automatyka Robotyka*, 16(7-8):76–85, 2012.
41. Kornuta T., Zieliński C., Robot control system design exemplified by multi-camera visual servoing. *Journal of Intelligent & Robotic Systems*, 1–25, 2013.
42. Lloyd J., Parker M., McClain R., Extending the RCCL Programming Environment to Multiple Robots and Processors. 465–469, 1988.
43. MacDonald B., Biggs G., Collett T., Software environments for robot programming. D. Brugali, redaktor, *Software Engineering for Experimental Robotics*, 107–124. Springer-Verlag, 2007.
44. Metta G., Fitzpatrick P., Natale L., YARP: Yet Another Robot Platform. *International Journal on Advanced Robotics Systems*, 3(1):43–48, 2006.
45. Mujtaba S., Goldman R., AL users' manual. Stanford Artificial Intelligence Laboratory, Sty. 1979.
46. Nesnas I., The CLARAty project: Coping with hardware and software heterogenity. Brugali D., redaktor, *Software Engineering for Experimental Robotics*, 9–30. Springer–Verlag, 2007.
47. Nesnas I., Simmons R., Gaines D., Kunz C., Diaz-Calderon A., Estlin T., Madison R., Guineau J., McHenry M., Shu I., Apfelbaum D., Claraty: Challenges and steps toward reusable robotic. *International Journal of Advanced Robotic Systems*, 3(1.):23–30, 2006.
48. Nesnas I. A. D., The CLARAty project: Coping with hardware and software heterogeneity. D. Brugali, redaktor, *Software Engineering for Experimental Robotics*, wolumen 30 serii *Springer Tracts in Advanced Robotics*, 31–70. Springer-Verlag, 2006.
49. Niederliński A., *Roboty przemysłowe*. Wydawnictwa Szkolne i Pedagogiczne, 1981.
50. Nilakantan A., Hayward V., The Synchronisation of Multiple Manipulators in Kali. *Robotics and Autonomous Systems*, 5(4):345–358, 1989.
51. Object Management Group. *The Common Object Request Broker: Architecture and Specification, Version 2.6.1*. Object Management Group, May 2002.
52. Paul R., WAVE: A model based language for manipulator control. *The Industrial Robot*, 10–17, March 1977.
53. Paul R., *Robot Manipulators: Mathematics, Programming, and Control*. The MIT Press, 1982.
54. Popplestone R. J., Ambler A.P., Bellos I., RAPT: A Language for Describing Assemblies. *Industrial Robot*, 5(3):131–137, September 1978.
55. Quigley M., Gerkey B., Conley K., Faust J., Foote T., Leibs J., Berger E., Wheeler R., Ng. ROS: an open-source Robot Operating System. *Proceedings of the Open-Source Software workshop at the International Conference on Robotics and Automation (ICRA)*, 2009.
56. Schmidt D. C., Gokhale A., Harrison T. H., Parulkar G., A high-performance endsystem architecture for real-time CORBA. *IEEE Communications Magazine*, 14(2), 1997.
57. Szyperski C., *Oprogramowanie komponentowe – obiekty to za mało*. WNT, 2001.
58. Vaughan R. T., Gerkey B. P., Reusable robot software and the Player/Stage project. D. Brugali, redaktor, *Software Engineering for Experimental Robotics*, wolumen 30 serii *Springer Tracts in Advanced Robotics*, 267–289. Springer, 2007.
59. Volpe R., Nesnas I., Estlin T., Mutz D., Petras R., Das H., The claraty architecture for robotic autonomy. Jet Propulsion Laboratory, 2001.
60. Willow Garage. Website of the Ecto framework for perception, [http://ecto.willowgarage.com], 2011.
61. Zieliński C., TORBOL – język programowania robotów przeznaczonych do wykonywania zadan transportowo-montażowych. *Archiwum Automatyki i Telemechaniki*, 3, 1989.
62. Zieliński C., TORBOL: An object level robot programming language. *Mechatronics*, 1(4):469–485, 1991.
63. Zieliński C., The MRROC++ system. *Proceedings of the First Workshop on Robot Motion and Control, RoMoCo'99*, 147–152, June 1999.
64. Zielinski C., Implementation of control systems for autonomous robots. *6th Int. Conf. on Control, Automation, Robotics and Vision, ICARCV'2000, 5–8 December 2000, Singapore (on CD-ROM)*, 2000.
65. Zieliński C., Formal approach to the design of robot programming frameworks: the behavioural control case. *Bulletin of the Polish Academy of Sciences – Technical Sciences*, 53(1):57–67, March 2005.
66. Zielinski C., Systematic approach to the design of robot programming frameworks. *Proceedings of the 11th IEEE International Conference on Methods and Models in Automation and Robotics (on CD)*, 639–646. Technical University of Szczecin, 29 August – 1 September 2005.
67. Zieliński C., Transition-function based approach to structuring robot control software. K. Kozłowski, (red.), *Robot Motion*

and Control, wolumen 335 serii *Lecture Notes in Control and Information Sciences*, 265–286. Springer-Verlag, 2006.
68. Zieliński C., *Inteligencja wokół nas. Współdziałanie agentów softwareowych, robotów, inteligentnych urządzeń*, wolumen 15, rozdział *Formalne podejście do programowania robotów – struktura układu sterującego*, 267–300. EXIT, 2010.
69. Zieliński C., Kasprzak W., Kornuta T., Szynkiewicz W., Trojanek P., Walecki M., Winiarski T., Zielinska T., Control and programming of a multi-robot-based reconfigurable fixture. *Industrial Robot: An International Journal*, 40(4):329–336, 2013.
70. Zieliński C., Kornuta T., Boryn M., Specification of robotic systems on an example of visual servoing. *10th International IFAC Symposium on Robot Control (SYROCO 2012)*, wolumen 10, 45–50, 2012.
71. Zieliński C., Kornuta T., Stefanczyk M., Szynkiewicz W., Trojanek P., Walęcki M., *Języki programowania robotów przemysłowych. Pomiary Automatyka Robotyka*, 16(11):10–19, 2012.
72. Zieliński C., Mianowski K., Nazarczuk K., Szynkiewicz W., *A Prototype Robot for Polishing and Milling Large Objects. Industrial Robot*, 30(1):67–76, January 2003.
73. Zieliński C., Szynkiewicz W., Mianowski K., Nazarczuk K., Mechatronic design of openstructure multi-robot controllers. *Mechatronics*, 11(8):987–1000, November 2001.
74. Zieliński C., Szynkiewicz W., Winiarski T., Staniak M., Czajewski W., Kornuta T., Rubik's cube as a benchmark validating MRROC++ as an implementation tool for service robot control systems. *Industrial Robot: An International Journal*, 34(5):368–375, 2007.
75. Zieliński C., Winiarski T., General specification of multi-robot control system structures. *Bulletin of the Polish Academy of Sciences – Technical Sciences*, 58(1):15–28, 2010.
76. Zieliński C., Winiarski T., Motion generation in the MRROC++ robot programming framework. *International Journal of Robotics Research*, 29(4):386–413, 2010.
77. Zieliński C., Winiarski T., Szynkiewicz W., Kornuta T., Trojanek P., *Inteligencja wokół nas. Współdziałanie agentów softwareowych, robotów, inteligentnych urządzeń*, wolumen 15, rozdział/l MRROC++ – programowa struktura ramowa do tworzenia sterowników systemów wielorobotowych, 317–384. EXIT, 2010.

Programming frameworks for development of robot controllers

Abstract: The constantly increasing diversity of types of robot and sensors opens new fields of applications. This affects the demand for tools facilitating their programming. This article focuses on the programming tools for development of high-level robotic controllers. It briefly discusses the evolution of methods of robot programming, starting from specialized languages, specialized libraries of functions for general purpose programming languages, ending up on robot programming frameworks. In particular, it presents a number of popular programming frameworks, enabling the creation of complex robot controllers.

Keywords: robot programming methods, robot programming languages, programming frameworks, robot controllers

dr inż. Tomasz Kornuta
tkornuta@ia.pw.edu.pl

Absolwent Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W 2003 roku uzyskał, tytuł, inżyniera, w 2005 tytuł, magistra inżyniera, a w 2013 stopień doktora nauk technicznych. Od 2008 roku pracuje w Instytucie Automatyki i Informatyki Stosowanej, a od 2009 roku pełni funkcję Kierownika Laboratorium Podstaw Robotyki. Jego zainteresowania naukowe obejmują metody programowania robotów oraz wykorzystanie informacji wizyjnej w robotyce, a w szczególności aktywną wizję oraz rozpoznawanie obrazów RGB-D. Autor/współautor ponad trzydziestu publikacji dotyczących powyższych tematów. Recenzent krajowych oraz międzynarodowych konferencji robotycznych (KKR, IEEE MMAR, IEEE ICAR, IFAC SYROCO) oraz czasopism (Pomiary Automatyka Robotyka, Sensor Review, International Journal of Advanced Robotic Systems). Członek IEEE RAS.

prof. dr hab. inż. Cezary Zieliński
c.zielinski@ia.pw.edu.pl

Od 2008 roku pracuje w Przemysłowym Instytucie Automatyki i Pomiarów PIAP. Ponadto jest profesorem na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W latach 2002–2005 sprawował, na tym wydziale funkcję prodziekana ds. nauki i współpracy międzynarodowej, 2005–2008 zastępcy dyrektora Instytutu Automatyki i Informatyki Stosowanej (IAiIS) ds. naukowych, a od 2008 pełni funkcję dyrektora tego instytutu. Od roku 1996 pełni rolę kierownika Zespołu Robotyki w IAiIS. Od 2007 roku jest członkiem Komitetu Automatyki i Robotyki Polskiej Akademii Nauk. Jego zainteresowania badawcze koncentrują się na zagadnieniach związanych z programowaniem i sterowaniem robotów.