Zezwala się na korzystanie z artykułu na warunkach licencji Creative Commons Uznanie autorstwa 3.0

Obsługa asynchronicznego przepływu danych w komponentowych podsystemach percepcji robotów

Tomasz Kornuta, Maciej Stefańczyk, Michał Laszkowski, Maksym Figat, Jan Figat, Cezary Zieliński

Instytut Automatyki i Informatyki Stosowanej, Politechnika Warszawska

Streszczenie: Obsługa asynchronicznego przepływu danych w złożonych potokach obliczeniowych, jakie z reguły tworzą podsystemy sensoryczne robotów, wymaga wytworzenia odpowiednich narzędzi wspomagających ich implementację. W artykule zaproponowano rozwiązanie umożliwiające warunkowe działanie poszczególnych bloków obliczeniowych w zależności od dostępnych danych. Rozważania teoretyczne doprowadziły do implementacji tych mechanizmów w strukturze ramowej DisCODe. Działanie rozwiązania przedstawiono na dwóch prostych przykładach.

Słowa kluczowe: robot, percepcja, komponent, struktura ramowa, DisCODe, asynchroniczny przepływ danych

DOI: 10.14313/PAR_207/127

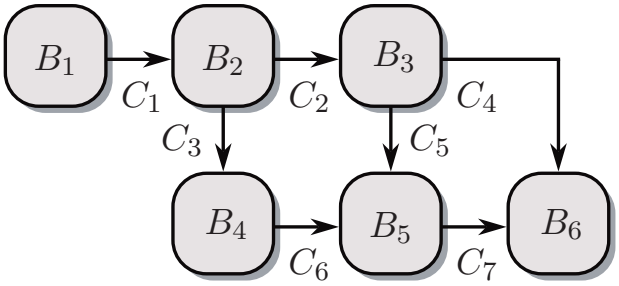
Aby sprawnie działać wśród ludzi, roboty usługowe muszą być wyposażone w różnorodne czujniki oraz układy przetwarzające odbierane dane sensoryczne. Z jednej strony dane te mogą nadchodzić z wielu czujników, pracujących z różnymi częstotliwościami. Przykładami tego typu podsystemów sensorycznych jest np. pas czujników ultradźwiękowych wykorzystywanych do nawigacji robota SCOUT [1, 2] czy też podsystem dokonujący fuzji danych odbieranych z kamery dookólnej oraz skanera laserowego, zamontowanych na robocie Elektron [3]. Z drugiej strony w przypadku kamer, a w szczególności czujników RGB-D typu Microsoft Kinect czy Asus Xtion, agregacja danych pomiarowych do postaci użytecznej w sterowaniu może wymagać wykonania szeregu operacji, tworzących w istocie złożone potoki obliczeniowe. Dodatkowo, w celu przyśpieszenia obliczeń, często wskazane jest ich zrównoleglenie za pomocą wielu wątków. Odpowiednie mechanizmy synchronizacji danych nadchodzących w sposób asynchroniczny mogą ułatwić i przyspieszyć implementację złożonych podsystemów sensorycznych.

1. Przetwarzanie danych odbieranych asynchronicznie

Agregacja danych pomiarowych wymaga przeprowadzenia szeregu, często złożonych obliczeń. Dlatego niezbędna jest dekompozycja procesu obliczeniowego, umożliwiająca wyróżnienie bloków prostych do zdefiniowania oraz implementacji. Poza wyróżnieniem szeregu bloków obliczeniowych niezbędne jest również określenie połączeń między nimi. Proces percepcji może być więc przedstawiony w postaci grafu, gdzie węzły odpowiadają blokom obliczeniowym, natomiast przepływ danych reprezentowany jest łukami. Oznaczając zbiór bloków jako B (od ang. *Computational Block*), a połączenia między nimi jako C (od ang. *Connection*), zadanie percepcji T można zdefiniować jako graf:

$$T = \langle B, C \rangle, \text{ gdzie } C \subset B \times B. \tag{1}$$

Przykładowy potok zaprezentowano na rys. 1. Konkretne bloki i połączenia rozróżniane są indeksami.

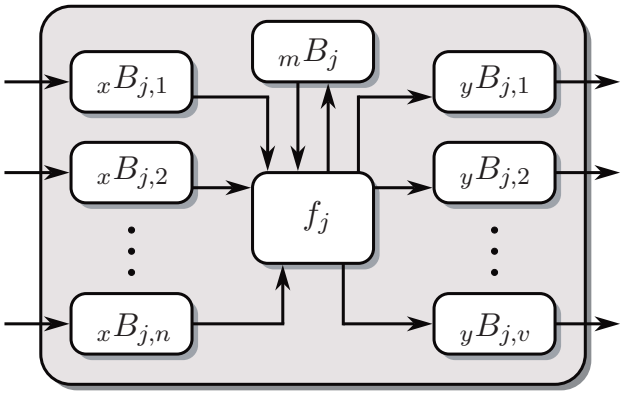


Rys. 1. Przykładowy potok obliczeniowy
Fig. 1. Exemplary computational flow

W każdym j-tym bloku obliczeniowym B_j można wyróżnić dwa rodzaje buforów: wejściowe ${}_xB_j$ oraz wyjściowe ${}_yB_j$. Dodatkowo blok ten może mieć również bufor wewnętrzny (pamięć) ${}_mB_j$, w którym mogą być przechowywane wartości zmiennych tymczasowych. Działanie bloku obliczeniowego w i-tym kroku można opisać za pomocą funkcji przejścia f_j, która na podstawie danych wejściowych oraz pamięci oblicza wartości buforów wyjściowych i dokonuje aktualizacji danych wewnętrznych (rys. 2):

$$\left[{}_mB_j^{i+1},\ {}_yB_j^{i+1}\right] := f_j\left({}_mB_j^{i},\ {}_xB_j^{i}\right). \tag{2}$$

Dodatkowo należy wziąć pod uwagę fakt, iż funkcja przejścia może działać na różne sposoby, zależnie od zawartości buforów wejściowych i buforu wewnętrznego. W szczególności, sposób jej działania może zmieniać się w zależności od dostępności danych w konkretnych buforach, co wynika bezpośrednio z faktu ich asynchronicznego nadchodzenia. Dekompozycja funkcji przejścia na zestaw funkcji cząstkowych może znacznie uprościć opis wariantów jej działania. Ze zbioru ${}_xB_j$ można wyróżnić podzbiór buforów ${}_xB_{j,\kappa} = [{}_xB_{j,k,1} \ldots, {}_xB_{j,k,l}]$, których zawartość niezbędna



Rys. 2. Ogólna struktura j-tego bloku obliczeniowego
Fig. 2. General structure of j-th computational block

jest do poprawnego przeprowadzenia obliczeń k-tej funkcji cząstkowej. $_yB_{j,\kappa}$ oznacza z kolei podzbiór buforów wyjściowych $_yB_j$, których wartości obliczone zostaną przez daną funkcję cząstkową. Prowadzi to do definicji k-tej cząstkowej funkcji przejścia (rys. 3):

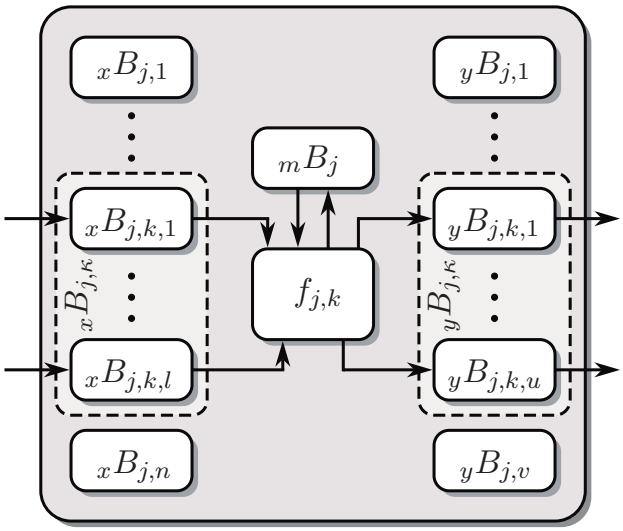
$$\left[{}_mB_j^{i+1},\ {}_yB_{j,\kappa}^{i+1}\right] :=\ f_{j,k}\left({}_mB_j^{i},\ {}_xB_{j,\kappa}^{i}\right), \qquad (3)$$

gdzie $k = 1, \ldots, n_{f_j}$ jest numerem funkcji cząstkowej, a n_{f_j} jest liczbą funkcji cząstkowych j-tego bloku obliczeniowego.

Po zdefiniowaniu kilku funkcji cząstkowych niezbędne jest określenie warunku ich aktywacji. Wychodząc z założenia, iż do przeprowadzenia obliczeń niezbędna jest znajomość danych w buforach wejściowych, można sformułować warunek aktywacji funkcji cząstkowej (3) w postaci predykatu:

$$P_{j,k}^{i}\left({}_xB_{j,\kappa}^{i}\right) =\ \exists\left({}_xB_{j,k,1}^{i}\right) \wedge \ldots \wedge \exists\left({}_xB_{j,k,n}^{i}\right), \qquad (4)$$

gdzie operator $\exists$ zwraca prawdę w przypadku obecności nowych (tzn. jeszcze nie wykorzystanych w obliczeniach



Rys. 3. Struktura j-tego bloku obliczeniowego z zaznaczeniem elementów związanych z k-tą funkcją cząstkową
Fig. 3. Structure of j-th computational block with k-th partial transition function elements highlighted

funkcji przejścia) danych w buforze. Powyższy warunek (4) jest niezbędny do przeprowadzenia obliczeń, natomiast nie określa kolejności wywołań funkcji, w przypadku gdy kilka warunków dla różnych funkcji jest spełnionych.

W dalszej części artykułu omówiono mechanizmy wspomagające programistę podczas implementacji bloków mogących działać warunkowo w wybranych robotycznych strukturach ramowych. W szczególności zaprezentowano strukturę ramową DisCODe oraz omówiono zaimplementowane w niej mechanizmy. Następnie uwagę skupiono na przykładowym komponencie tej struktury, realizującym przetwarzanie obrazów RGB-D do chmur punktów, działające warunkowo w zależności od obecności danych w poszczególnych buforach wejściowych. Zaprezentowano także dwa proste przykłady zadań percepcji wykorzystujące ten komponent.

2. Robotyczne struktury ramowe

Orocos W strukturze ramowej OROCOS (ang. *Open RObot COntrol Software*) [4] nie istnieją żadne wyspecjalizowane mechanizmy wspomagające wariantowe przetwarzanie danych, a zadanie zapewnienia odpowiedniego kolejkowania i wywoływania funkcji przejścia spoczywa na programiście. Stan portów wejściowych można sprawdzić za pomocą funkcji zwrotnych (ang. *Callback*) związanych z danymi portami, wywoływanymi w sytuacji, kiedy w porcie znajdują się nowe dane, lub za pomocą funkcji *updateHook()* wywoływanej w każdym przebiegu komponentu. Największym problemem, z którym należy się zmierzyć, jest rozpoznanie, czy dane w porcie są faktycznie "stare", zostały zmienione od czasu poprzedniego wywołania komponentu, czy też zostały odczytane przez poprzednie funkcje tego samego komponentu. Spowodowane jest to możliwością sprawdzenia jedynie tego, czy dane zostały już odczytane z portu czy nie, nie ma natomiast możliwości całkowitego wyczyszczenia zawartości portu.

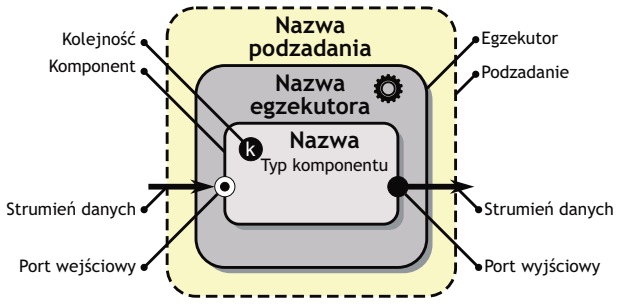
Ecto Podstawową jednostką obliczeniową struktury ramowej Ecto [5] jest komórka (ang. *Cell*), wyposażona w zestaw buforów wejściowych i wyjściowych (ang. *Tendrils*) oraz w pojedynczą funkcję przejścia, która przyjmuje jako argument wszystkie bufory wejściowe oraz generuje wyniki zapisywane na wszystkich wyjściach komórki. Funkcja ta uruchamia się tylko i wyłącznie wtedy, kiedy wszystkie dane wejściowe będą przygotowane. Z tego powodu w systemie tym problem wariantowego przetwarzania danych nie występuje na poziomie komponentu, a dopiero na poziomie całego zadania, definiując wariantowe potoki obliczeń. To jednak wymaga wytworzenia kilku wariantów tego samego komponentu, dla każdej kombinacji buforów umożliwiających przeprowadzenie obliczeń. W tym miejscu należy podkreślić zaletę struktury Ecto, która związana jest z innym przyjętym w Ecto założeniem, a mianowicie z acyklicznością grafów skierowanych (ang. *Plasm*) opisujących połączenia między komórkami. Umożliwia to topologiczne posortowanie grafu, co w połączeniu z założeniem produkcji wszystkich danych na wszystkich wyjściach komórek

zapewnia stałą i deterministyczną kolejność wywoływania funkcji w systemie.

ROS Struktura ramowa ROS (ang. *Robot Operating System*) [6] zawiera najbardziej rozbudowane mechanizmy synchronizacji wywoływania funkcji obsługi z danymi pojawiającymi się na wejściach kanałów komunikacyjnych zwanych tematami (ang. *Topic*) węzłów obliczeniowych (ang. *Node*). ROS oferuje mechanizm łączący 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 opcje: wywołanie funkcji w momencie, gdy wszystkie dane mają taką 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 uruchomionych (np. pokrywają się ich zestawy wejść) jest nieokreślona, nie ma również możliwości sprawdzenia, czy konkretne dane zostały już wykorzystane w ramach bieżącego cyklu pracy komponentu.

3. DisCODe

Trzon struktury ramowej DisCODe (ang. *Distributed Component Oriented Data Processing*) [7] został napisany w języku C++, a sama implementacja została oparta na trzech paradygmatach: programowania zorientowanego komponentowo [8], programowania refleksyjnego [9] oraz programowania generycznego [10]. Kombinacja tych paradygmatów dała podstawy do stworzenia mechanizmów pozwalających na dekompozycję dowolnego procesu percepcji na graf skierowany niezależnych, ale potrafiących ze sobą współpracować komponentów. Komponenty związane z różnymi etapami lub krokami percepcji przechowywane są w oddzielnych bibliotekach komponentów (zwanych DCL od ang. *DisCODe Component Library*), natomiast sama struktura dostarcza niezależnych od zadań mechanizmów ich wczytywania, uruchamiania, zarządzania ich konfiguracją, synchronizacją oraz wymianą danych między nimi.

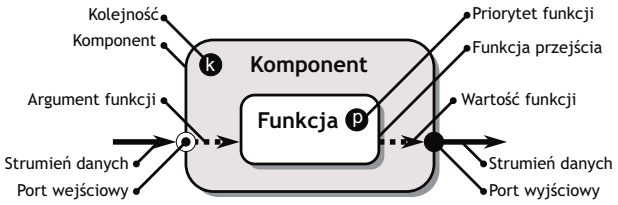


Rys. 4. Graficzna reprezentacja głównych elementów zadania DisCODe
Fig. 4. Graphical representation of the major elements of the DisCODe task

Podstawowymi elementami zadań percepcji są komponenty przekazujące sobie informacje za pomocą strumieni danych (rys. 4). Komponenty organizowane są w wątki zarządzające ich pracą zwane egzekutorami. Umożliwia to przyspieszenie działania całego procesu percepcji przez zrównoleglenie obliczeń. W DisCODe założone zostało, iż to użytkownik odgórnie (tzn. w pliku konfiguracyjnym danego zadania) decyduje o kolejności obsługi komponentów pracujących w ramach danego egzekutora. Podzadania stanowią następne narzędzie do organizacji i zarządzania komponentami, nadrzędne w stosunku do egzekutorów. Podział na podzadania jest istotny z punktu widzenia pracy systemu jako podsystemu sensorycznego układu sterowania robota – dzięki niemu można zaplanować wiele różnych potoków przetwarzania, a w danym momencie uruchamiać tylko te, które są potrzebne.

3.1. Aktywacja funkcji przejścia

Każdy komponent ma zestaw portów wejściowych, przez które otrzymuje dane, oraz portów wyjściowych, do których zapisuje wyniki swoich obliczeń (rys. 5). Za przeprowadzanie tych obliczeń odpowiedzialne są funkcje, których argumentami są wybrane porty wejściowe, natomiast produkowane przez nie wartości zapisywane są do wybranych portów wyjściowych. Aktywacja funkcji zależy od dostępności argumentów oraz priorytetów funkcji.



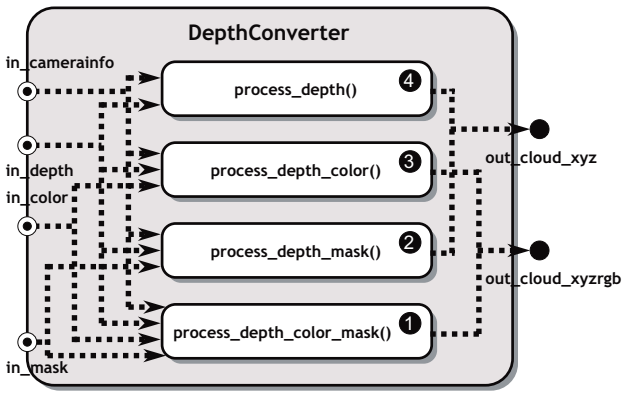
Rys. 5. Główne elementy komponentu DisCODe
Fig. 5. Major elements of the DisCODe component

Ponieważ może zaistnieć sytuacja, w której dane obecne w portach umożliwią aktywację kilku funkcji, wprowadzono priorytety, które ustalą kolejność ich wywołania. Przyjęto przy tym dwa założenia:

– nie jest możliwe zdefiniowanie dwóch funkcji zależnych od tego samego podzbioru portów wejściowych,

– priorytety związane są z licznością portów wejściowych, co oznacza, że wyższe priorytety zostają przyznane funkcjom zależącym od większej liczby portów.

Przyjęcie tych założeń umożliwia automatyczne przyznawanie priorytetów w większości przypadków oraz wymaga interwencji jedynie w przypadku, gdy różne funkcje są zależne od tej samej liczby portów. Rozwiązaniem tego problemu jest umożliwienie użytkownikowi ustalenia priorytetów względnych, uwzględnianych przez system podczas obliczania właściwych priorytetów wszystkich funkcji. Mając funkcje posortowane po priorytetach, działanie każdego komponentu można zobrazować w postaci automatu, gdzie sprawdzane są warunki kolejnych funkcji aż do aktywacji pierwszej z nich. W większości wypadków funkcja wykorzysta dane z portów, od których zależy, co spowoduje ich wyłączenie z dalszego procesu poszukiwania aktywnych funkcji. Istnieje jednak możliwość zdefiniowania funkcji w taki sposób, aby uruchamiała się po pojawieniu się danych w którymś z portów, jednocześnie z tych danych



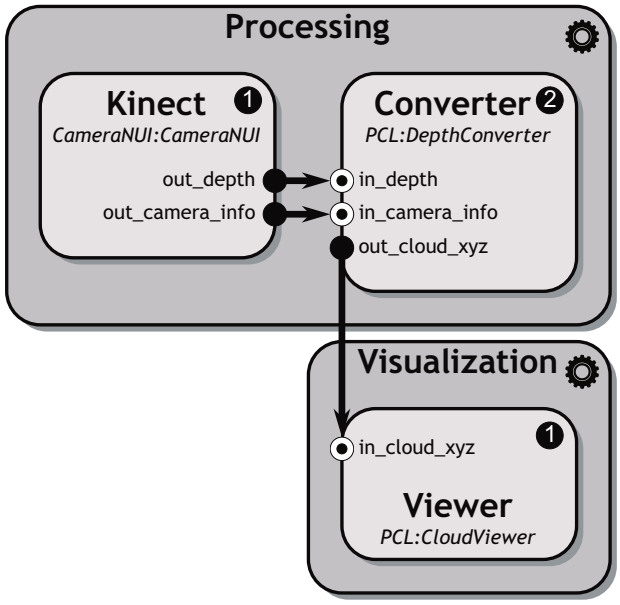
Rys. 6. Struktura komponentu DepthConverter
Fig. 6. Structure of the DepthConverter component

fizycznie nie korzystając. W takiej sytuacji dane pozostające w porcie wejściowym mogą być wykorzystane do aktywacji kolejnej funkcji (o ile zajdzie taka potrzeba).

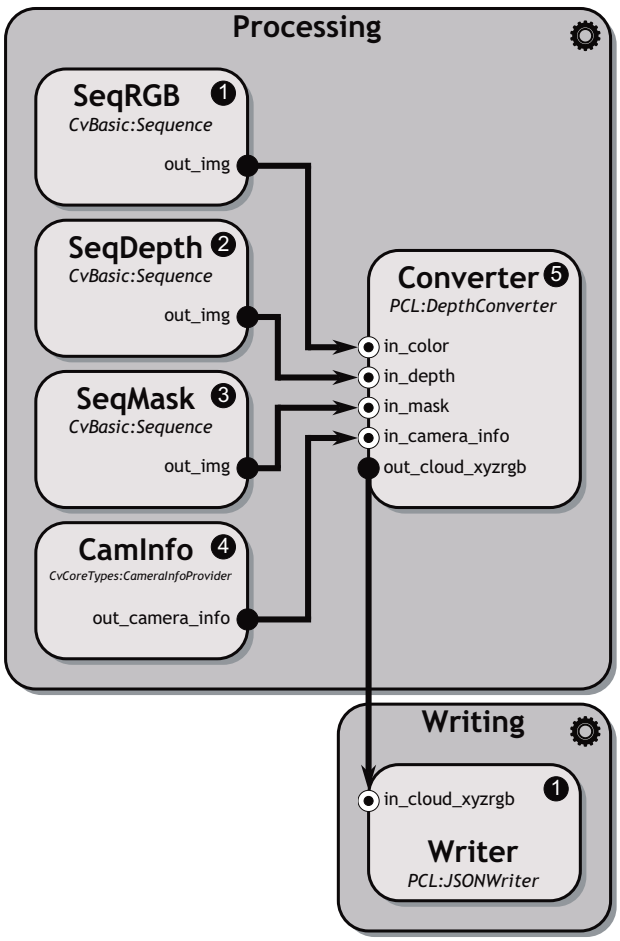
Należy rozpatrzyć jeszcze aspekt związany z faktem, iż tak opisane działanie każdego z komponentów abstrahuje od przydziału czasu procesora poszczególnym komponentom danego egzekutora. Aby nie doszło do znanego z teorii systemów operacyjnych zagłodzenia założono, iż w jednym cyklu obliczeniowym dla danego komponentu następuje próba pojedynczego uruchomienia każdej funkcji, w kolejności wynikającej z priorytetów, po czym system przechodzi do kolejnego komponentu. Do wyboru następnego komponentu wykorzystano ustalaną odgórnie przez użytkownika (w pliku konfiguracyjnym zadania) kolejność wywoływania.

3.2. Przykład: DepthConverter

Jako przykład obrazujący działanie omawianego mechanizmu obsługi danych wybrano komponent **DepthConverter**, realizujący transformację między dwoma głównymi reprezentacjami głębi: obrazem RGB-D a chmurą punktów. Obraz RGB-D jest to czterokanałowy obraz, z czego pierwsze trzy kanały zawierają informacje dotyczące koloru (RGB), a czwarty mapę głębi. Z kolei chmura punktów przechowuje położenie punktów w przestrzeni kartezjańskiej.



Rys. 7. Zadanie wyświetlające chmury punktów pobrane z czujnika Kinect
Fig. 7. Task responsible for display of point clouds acquired from the Kinect sensor

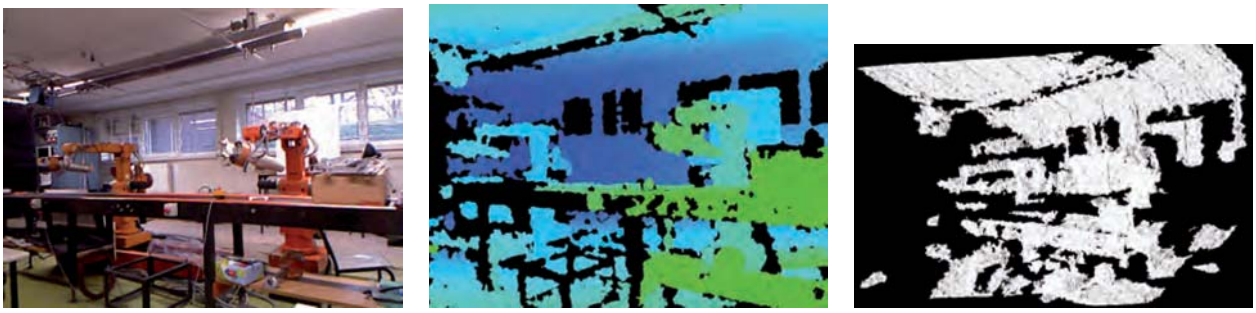


Rys. 8. Zadanie zapisujące chmurę punktów pojedynczego widoku obiektu do pliku
Fig. 8. Task responsible for saving of point clouds of a single view of an object to a file

Na rys. 6 pokazano strukturę komponentu **DepthConverter**. Komponent oferuje cztery podstawowe metody transformacji:

- generację chmury punktów obserwowanej sceny na podstawie samej mapy głębi: **process_depth**(),
- generację chmury kolorowych punktów obserwowanej sceny na podstawie mapy głębi oraz obrazu kolorowego: **process_depth_color**(),
- generację chmury punktów obiektu na podstawie mapy głębi oraz znanej maski obiektu: **process_depth_mask**(),
- generację chmury kolorowych punktów obiektu na podstawie mapy głębi, obrazu kolorowego oraz maski obiektu: **process_depth_mask_color**().

Wykorzystywana w ostatnich dwóch przypadkach maska wyznaczana jest w procesie zewnętrznej segmentacji. Maska ta, podobnie jak obrazy kolorowe oraz mapa głębi, stanowią trzy porty wejściowe, odpowiednio **in_mask**, **in_color** oraz **in_depth**. Do przechowywania powyższych trzech rodzajów danych wykorzystujemy strukturę **Mat** z biblioteki OpenCV



(a) (b) (c)

Rys. 9. Przykładowe działanie pierwszego zadania: (a) obraz RGB (b) mapa głębi (c) wynikowa chmura punktów
Fig. 9. Results of the work of the first task: (a) RGB image (b) depth map (c) the resulting poing cloud

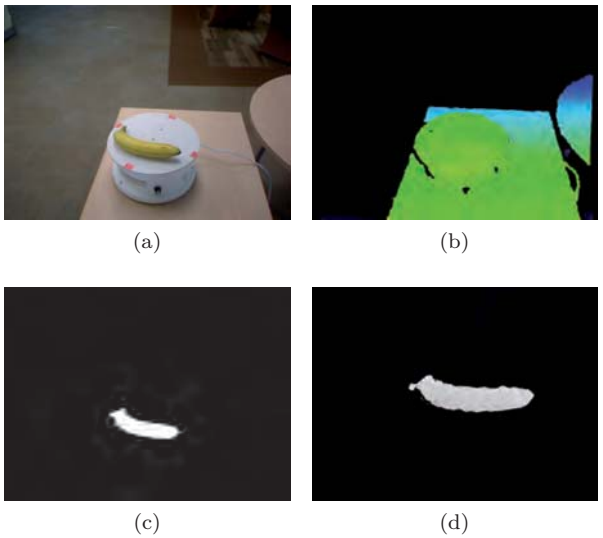
(ang. *Open Computer Vision Library*) [11]. Ponieważ do wyznaczenia położeń punktów wynikowej chmury w układzie kartezjańskim związanym z kamerą niezbędna jest również znajomość jej parametrów wewnętrznych (długość ogniskowej, przesunięcie matrycy etc.), dane te odbierane są przez dodatkowy port wejściowy **in_camera_info**.

Komponent ma dwa wyjścia: **out_cloud_xyz** oraz **out_cloud_xyzrgb**, w których zwracane są odpowiednio zwykłe oraz kolorowe chmury punktów. Chmury te są przechowywane w szablonowej strukturze **PointCloud** z biblioteki PCL (ang. *Point Cloud Library*) [12].

Priorytety związane z kolejnością sprawdzania możliwości wywoływania funkcji są wyznaczane przez liczbę portów wejściowych, od których funkcje te zależą. W pierwszej kolejności sprawdzana jest więc funkcja **process_depth_mask_color()**, dalej **process_depth_color()**, **process_depth_mask()**, a ostatnią jest **process_depth()**. Warto zwrócić uwagę na fakt, iż w przypadku funkcji **process_depth_color()** oraz **process_depth_mask()** priorytetyzacja względna nie gra żadnej roli, gdyż w przypadku obecności danych wymaganych przez zarówno jedną, jak i drugą funkcję, wywołana zostanie inna funkcja: **process_depth_mask_color()**.

Na rys. 7 pokazano strukturę realizującą pierwsze, przykładowe zadanie wykorzystujące powyżej omówiony komponent. W zadaniu tym obraz mapa głębi pochodząca z czujnika Kinect wraz z jego parametrami wewnętrznymi przekazywane są z komponentu typu **CameraNUI** do komponentu typu **DepthConverter**. Wygenerowana chmura punktów przekazywana jest do komponentu typu **CloudViewer** odpowiedzialnego za wyświetlenie wynikowej chmury na ekranie monitora. Na rys. 9 pokazano przykładowe obrazy otrzymane w wyniku działania powyższego zadania (punkt widzenia chmury pokazanej na rys. 9c jest inny od punktu widzenia czujnika Kinect podczas akwizycji).

Struktura drugiego przykładowego zadania wykorzystującego komponent **DepthConverter** pokazana została na rys. 8. Zadanie to odpowiedzialne jest za generację chmury punktów należących do obiektu i stanowi fragment prowadzonych prac w zakresie rozpoznawania obiektów trójwymiarowych w obrazach RGB-D. W pracach tych wykorzystywany jest zbiór widoków 300 obiektów codziennego użytku stworzony na Uniwersytecie w Waszyngtonie, tzw. *Washington RGB-D Object Dataset* [13]. W zbiorze tym na każdy widok obiektu składają się trzy pliki: plik zawie-



(a) (b) (c) (d)

Rys. 10. Przykładowe wyniki działania drugiego zadania: (a) obraz RGB (b) mapa głębi (c) maska obiektu (d) wynikowa chmura punktów
Fig. 10. Exemplary results of the work of the second task: (a) RGB image (b) depth map (c) object mask (d) the resulting poing cloud

rający obraz RGB, plik zawierający mapę głębi oraz plik zawierający maskę binarną – obraz, w którym piksele należące do obiektu zaznaczone są kolorem białym, a pozostałe (tło) kolorem czarnym. Obrazy te wczytywane są z plików za pomocą trzech komponentów typu **Sequence**. Dodatkowo wykorzystywany jest komponent typu **CameraInfoProvider**, który dostarcza parametrów czujnika Kinect. Dane te podawane są na wejście komponentu typu **DepthConverter**, który generuje kolorową chmurę punktów należących tylko i wyłącznie do obiektu. Chmura ta przesyłana jest do komponentu typu **JSONWriter**, która zapisuje ją do pliku. Przykładowe wyniki działania tego zadania pokazano na rys. 10.

4. Podsumowanie

W artykule skupiono uwagę na przetwarzaniu danych w złożonych procesach obliczeniowych, jakie typowo tworzą podsystemy percepcji robotów. Zaprezentowano metodę dekom-

pozycji tych potoków na bloki obliczeniowe oraz skupiono uwagę na problemie warunkowego przetwarzania asynchronicznie nadchodzących danych. Pokrótce omówiono mechanizmy realizowane w wybranych robotycznych programowych strukturach ramowych oraz zaprezentowano szczegóły rozwiązania zaimplementowanego w strukturze ramowej DisCODe. Działanie tego rozwiązania zaprezentowano na przykładach wykorzystujących komponent przetwarzający obrazy RGB-D do postaci chmur punktów. Należy wyjaśnić, iż przykładowy komponent oraz zadania zostały wybrane ze względu na ich prostotę, a w DisCODe dostępne są również inne komponenty wykorzystujące omówione mechanizmy, np. komponent realizujący wielomodalną segmentację obrazu [14].

Warto dodać, iż priorytetyzacja nie jest jedynym możliwym mechanizmem umożliwiającym selekcję funkcji ze zbioru funkcji w oparciu o obecność danych w buforach wejściowych. W szczególności rozważaliśmy rozwiązanie, w którym użytkownik musiałby zdefiniować pełen automat, określający jaka funkcja ma być wywoływana dla każdej kombinacji obecności/braku danych w buforach. Należy jednak zauważyć, że dla komponentu o pięciu wejściach należałoby ręcznie przyporządkować funkcje do każdej z $2^5 = 32$ kombinacji. W przypadku mechanizmu automatycznej priorytetyzacji użytkownik ręcznie musi rozstrzygnąć co najwyżej kilka niejednoznaczności (ta sama liczba buforów).

Podziękowania

Praca finansowana ze środków Dziekana Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej w ramach grantu nr 504M/1031/0016 oraz grantu nr 504M/1031/0044.

Bibliografia

1. Zieliński C., Kornuta T., Trojanek P., Winiarski T., *Metoda projektowania układów sterowania autonomicznych robotów mobilnych. Część 1. Wprowadzenie teoretyczne*, "Pomiary Automatyka Robotyka" 9/2011, 84–87.
2. Zieliński C., Kornuta T., Trojanek P., Winiarski T., *Metoda projektowania układów sterowania autonomicznych robotów mobilnych. Część 2. Przykład zastosowania*, "Pomiary Automatyka Robotyka" 10/2011, 84–91.
3. Olszewski M., Siemiątkowska B., Chojecki R., Marcinkiewicz P., Trojanek P., Majchrowski M., *Mobile robot localization using laser range scanner and omni-camera*, [in:] Zielińska T., Zieliński C. (eds.), CISM Courses and Lectures - 16th CISM–IFToMM Symposium on Robot Design, Dynamics and Control, RoManSy'06, Springer, Wien, New York, June 20–24, 2006, 229–236.
4. Bruyninckx H. (2003): *The Real-Time Motion Control Core of the OROCOS Project*, [in:] Proceedings of the IEEE International Conference on Robotics and Automation, 2766–2771, IEEE.
5. Willow Garage, *Website of the Ecto framework for perception, `http://ecto.willowgarage.com`*, 2011.
6. Quigley M., Gerkey B., Conley K., Faust J., Foote T., Leibs J., Berger E., Wheeler R., Ng A., *ROS: an open-source Robot Operating System*, [in:] Proceedings of the Open-Source Software workshop at the International Conference on Robotics and Automation (ICRA), 2009.
7. Kornuta T., Stefańczyk M., *DisCODe: komponentowa struktura ramowa do przetwarzania danych sensorycznych*, "Pomiary Automatyka Robotyka" 7-8/2012, 76–85.
8. Szyperski C., Gruntz D., Murer S. (2002): *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2nd edition.
9. Sobel J. M., Friedman D. P., *An Introduction to Reflection-Oriented Programming*, 1996.
10. Alexandrescu A. (2001): *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional.
11. Bradski G., Kaehler A. (2008): *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, 1st edition.
12. Rusu R. B., Cousins S., *3D is here: Point Cloud Library (PCL)*, [in:] International Conference on Robotics and Automation, Shanghai, China, 2011, 2011.
13. Lai K., Bo L., Ren X., Fox D., *A large-scale hierarchical multi-view rgb-d object dataset*, [in:] Robotics and Automation (ICRA), 2011 IEEE International Conference on, IEEE, 2011, 1817–1824.
14. Stefańczyk M., Kasprzak W., *Multimodal segmentation of dense depth maps and associated color information*, [in:] Bolc L., Tadeusiewicz R., Chmielewski L., Wojciechowski K. (eds.), Proceedings of the International Conference on Computer Vision and Graphics, Springer Berlin / Heidelberg, 2012, 626–632.

Artykuł recenzowany, nadesłany 05.12.2013 r., przyjęty do druku 09.04.2014 r.

Asynchronous data flow handling in component-based robot perception subsystems

Abstract: Handling of asynchronous data flows in complex computational systems such as robot sensor subsystems requires appropriate tools facilitating their implementation. The article proposes a solution to the aforementioned problem. The solution enables the activation of a conditional behaviour of the individual computational blocks, depending on the presence of data in their input buffers. Theoretical considerations led to the implementation of these mechanisms in a component-oriented framework for development of diverse robot perception subsystems: DisCODe. Operation of the solution is illustrated in two simple exemplary tasks.

Keywords: robot, perception, component, framework, DisCODe, asynchronous data flow

dr inż. Tomasz Kornuta

Absolwent Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W 2003 r. uzyskał tytuł inżyniera, w 2005 r. tytuł magistra inżyniera, a w 2013 r. stopień doktora nauk technicznych. Od 2008 r. pracuje w Instytucie Automatyki i Informatyki Stosowanej, a od 2009 r. 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 czterdziestu publikacji dotyczących powyższych tematów. Recenzent krajowych oraz międzynarodowych konferencji robotycznych (KKR, IEEE MMAR, IEEE ICAR, IFAC SYROCO) oraz czasopism (Sensor Review, International Journal of Advanced Robotic Systems). Członek IEEE RAS.

e-mail: tkornuta@ia.pw.edu.pl

mgr inż. Maciej Stefańczyk

Absolwent Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W 2010 r. uzyskał tytuł inżyniera, w 2011 r. tytuł magistra inżyniera, oba z wyróżnieniem. W 2011 r. rozpoczął pracę nad doktoratem dotyczącym zastosowania aktywnej wizji wraz z systemami opartymi na bazie wiedzy w systemie sterowania robotów. Główne zainteresowania naukowe obejmują zastosowanie informacji wizyjnej, zarówno w robotyce, jak i systemach rozrywki komputerowej.

e-mail: stefanczyk.maciek@gmail.com

inż. Michał Laszkowski

W lutym 2014 r. uzyskał tytuł inżyniera specjalności Informatyka na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W ramach pracy magisterskiej rozwija metody rozpoznawania trójwymiarowych obiektów w chmurach punktów z wykorzystaniem cech ekstrahowanych zarówno z informacji kolorowej, jak i z map głębi. Główny obszar jego zainteresowań naukowych stanowi percepcja robotów z wykorzystaniem czujników RGB-D.

e-mail: mlaszkow@gmail.com

mgr inż. Maksym Figat

Absolwent Wydziału Matematyki i Nauk Informacyjnych Politechniki Warszawskiej. Doktorant pierwszego roku na kierunku Automatyka i Robotyka na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W swoich badaniach skupia się na wykorzystaniu inżynierii oprogramowania w robotyce. W szczególności zajmuje się językami dziedzinowymi oraz inżynieria sterowana modelem w celu wytworzenia narzędzi ułatwiających tworzenie złożonych aplikacji robotycznych.

e-mail: maksym.figat44@gmail.com

mgr inż. Jan Figat

Doktorant pierwszego roku na kierunku Automatyka i Robotyka na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Główny obszar jego zainteresowań naukowych stanowi percepcja robotów z wykorzystaniem czujników RGB-D typu Microsoft Kinect. Obecnie pracuje nad metodami łączenia wielu chmur punktów oraz ekstrakcją ich cech.

e-mail: jan.figat@gmail.com

prof. dr hab. inż. Cezary Zieliński

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

e-mail: c.zielinski@ia.pw.edu.pl