Rafał BRZUCHACZ*

ZASTOSOWANIE JĘZYKA LOTOS DO MODELOWANIA WYBRANYCH STRUKTUR METODY HOOD

STRESZCZENIE

W pracy zaprezentowano możliwości użycia języka LOTOS [3, 7] w projektowaniu oprogramowania systemów czasu rzeczywistego metodą HOOD [6, 13]. Wybrane struktury HOOD zostają wyrażone w języku algebry procesów i abstrakcyjnych typów danych, co zapewnia możliwość formalnej analizy tworzonego projektu. We wprowadzeniu do pracy wyjaśniono cel i motywacje do tworzenia modelu formalnego projektu systemu czasu rzeczywistego. Następnie przedstawiono skrócony opis metodyki hierarchicznego projektowania HOOD, której notacja i proces projektowy stanowią bazę dla proponowanej metody formalizacji. Wyjaśniono znaczenie takich konstrukcji, jak moduł, interfejs, operacja, hierarchia użycia i zawierania. W kolejnym punkcie, po krótkim przedstawieniu języka LOTOS [3, 7], zaprezentowano technikę, w której konstrukcje metody HOOD zostają zinterpretowane w języku formalnym algebry procesów i abstrakcyjnych typów danych. Opis przeprowadzony jest dla prostego przykładu projektu w HOOD, celem wyjaśnienia istoty interpretacji. W części końcowej pracy podano wnioski wraz ze wskazaniem dalszych możliwych kierunków rozwoju i zastosowań prezentowanej metody.

***Słowa kluczowe**: systemy czasu rzeczywistego, weryfikacja oprogramowania, algebry procesów, abstrakcyjne typy danych, LOTOS, HOOD*

USE OF LOTOS LANGUAGE IN MODELING HOOD METHOD STRUCTURES

The article presents a possibility of using LOTOS [3, 7] formal language in the HOOD [6, 13] real-time system design. Process algebras and abstract data types are used to express some HOOD structures in order to allow formal analysis of the system. The introduction explains the motivations and the goal of the work. Then, there is a brief survey of the HOOD hierarchical design method in the next point. It states as the base for the formalization method explained in the article. The notions of module, interface, operation, use and include relations are briefly explained. After it, the simple LOTOS language constructs are also presented. Next point explains the formalization of the HOOD structures. This is the most essentials part of the work where the method is presented on the exemplary HOOD diagram. LOTOS code is produced relate to the HOOD informal semantic. It expresses the main concepts of the HOOD dynamic model. There are the conclusions and further works proposals at the end of the article.

***Keywords**: real-time systems, software verification, process algebras, abstract data types, LOTOS, HOOD*

1. WPROWADZENIE

1.1. Cel pracy

W pracy zaprezentowano możliwości użycia języka LOTOS (*Language of Temporal Ordering Specification*) do projektowania oprogramowania systemów czasu rzeczywistego metodą HOOD (*Hierarchical Object Oriented Design*). Wybrane struktury HOOD zostają wyrażone w języku algebry procesów i abstrakcyjnych typów danych, co zapewnia możliwość formalnej analizy tworzonego projektu. Zaproponowana technika buduje w tym celu model formalny struktur metody, które definiują dynamiczny model oprogramowania systemu czasu rzeczywistego (SCR).

1.2. Motywacje

Projektowanie oprogramowania systemów czasu rzeczywistego

Projektowanie oprogramowania SCR nie jest zadaniem łatwym ze względu na szereg specyficznych własności, które musi posiadać tworzona aplikacja. Są to przede wszystkim [14]:

- ciągłość działania,
- zależność od otoczenia,
- współbieżność,
- przewidywalność,
- punktualność.

Cechy te decydują o szczególnej trudności w budowaniu oprogramowania SCR, w którym testowanie powstających modułów programu napotyka poważne trudności lub jest wręcz niemożliwe. Ponadto, ze względu na zastosowania aplikacji SCR np. w lotnictwie lub medycynie błędy w oprogramowaniu są niedopuszczalne. Konieczne staje się więc zastosowanie bardziej zaawansowanych środków projektowych, które zapewnią wykrycie błędów w fazie poprzedzającej implementację systemu.

Metody formalne projektowania SCR

Naturalnym podejściem do stosowania metod formalnych jest bezpośrednie projektowanie oprogramowania SCR

* Doktorant, Akademia Górniczo-Hutnicza, Wydział EAIiE

w odpowiednim języku formalnym. Stworzony zostaje wówczas matematyczny model aplikacji, który można następnie poddać automatycznej weryfikacji przy użyciu wyspecjalizowanego narzędzia. Niewątpliwą zaletą takiego podejścia jest możliwość wykrycia błędów w projekcie jeszcze przed przejściem do fazy implementacji systemu. Wadą jest często duże skomplikowanie tworzonego modelu formalnego już dla niedużych systemów. W obrębie metod używanych do specyfikowania i projektowania oprogramowania SCR można wymienić:

– sieci Petriego [8],
– algebry procesów CCS (*Calculus of Communicating Systems*) [11] i CSP (*Communicating Sequential Processes*) [5],
– logiki temporalne [9].

Wsparcie formalne dla istniejących metod projektowych

Konstrukcja formalnego modelu poprzez translację artefaktów utworzonych z zastosowaniem istniejących metod i języków stanowi odrębne podejście do projektowania SCR. Metody obiektowe HOOD [6], COMET (*Concurrent Object Modeling and architectural design mEThod*) [4] lub strukturalne SADT (*Structural Analysis and Design Technique*), SA/SD (*Structural Analysis/Structural Design*) [14] wykorzystywane w projektowaniu SCR dysponują bogatymi środkami opisu systemu (różnego rodzaju diagramy), jednak utworzone modele nie mogą być poddane formalnej analizie. Proponowanym w niniejszym artykule rozwiązaniem jest stworzenie formalnej interpretacji struktur wybranej metody projektowej, które mogą zostać następnie wykorzystane do weryfikacji powstającego modelu oprogramowania. W pracy wykorzystano metodę HOOD [13, 6], która należy do grupy technik obiektowych z dobrze zdefiniowanym procesem projektowania aplikacji.

Zastosowanie formalnego modelowania w HOOD

HOOD wykorzystuje metodę kolejnych uściśleń w budowanym projekcie. Począwszy od modułu stanowiącego specyfikację całego systemu, projekt polega na dekomponowaniu każdego modułu do zbioru modułów potomnych. Im niższy poziom w drzewie dekompozycji, tym więcej szczegółów projektowych. Szczególnie ważne jest tutaj zachowanie spójności między kolejnymi warstwami tego drzewa. Implementacja modułu w postaci zbioru jego potomków powinna zachować funkcjonalność modułu rodzica. Weryfikacja tej właściwości w sferze modeli nieformalnych jest bardzo trudna, często praktycznie niemożliwa. Pojawia się potrzeba wyrażenia modułów i relacji je wiążących w notacji, która pozwoli rozstrzygnąć, czy niższe poziomy drzewa projektowego zachowują funkcjonalność poziomów wyższych. Zbadanie tej własności jest w projekcie HOOD kluczowe i stanowi podstawową przesłankę do budowy modelu formalnego w języku LOTOS.

Z technicznego punktu widzenia formalizacja pozwala w pełni zautomatyzować proces weryfikacji modelu. Co więcej, sam model może też być generowany automatycznie na podstawie diagramów wykorzystywanej metody projektowej.

2. WYBÓR METODY FORMALNEJ

Wykorzystanie HOOD pociąga za sobą decyzje odnośnie do wyboru metody formalnej. HOOD, jak to zostanie niżej pokazane, określa hierarchię modułów pozwalających na ukrycie nieistotnych w danym etapie szczegółów projektu. Własność taka przysługuje także formalnym językom algebr procesów takim, jak CCS [11] i CSP [5] czy też LOTOS [7]. System specyfikowany w algebrze procesów jest procesem (agentem w CCS) realizującym komunikację z otoczeniem (innymi procesami) przez porty i ukrywającym wewnętrzną strukturę. Struktura ta jest zbiorem podprocesów realizujących funkcjonalność procesu wyższego szczebla. Podproces jest w sensie ogólnym także pewnym procesem, więc definiowana jest tutaj wielopoziomowa struktura realizująca funkcjonalność procesu najbardziej zewnętrznej warstwy. Taka organizacja systemu dobrze odpowiada hierarchii modułów w HOOD. W artykule wykorzystano język LOTOS z uwagi na następujące jego cechy. Po pierwsze stanowi on połączenie CCS i CSP, co zapewnia stosowną ekspresywność języka. Po drugie zawiera konstrukcje abstrakcyjnych typów danych, co dobrze nadaje się do bezpośredniego modelowania struktur HOOD takich jak operacja ograniczona czy też maszyna skończenie stanowa modułu.

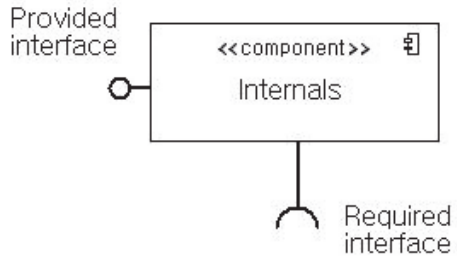
3. HOOD

HOOD [13, 6, 14] należy do klasy metod obiektowych. Obejmuje dość szeroki zakres w cyklu życia oprogramowania, począwszy od projektowania wstępnego systemu, a kończąc na jego implementacji (rys. 1). Niniejszy punkt stanowi przegląd podstawowych konstrukcji metody. Zastosowano tutaj notację UML zgodną ze specyfikacją 2.0 tego języka [12]. Konstrukcje niedające się wyrazić w UML (*Universal Modeling Language*), są definiowane tylko w języku naturalnym.

Rys. 1. Zakres HOOD w cyklu życia oprogramowania

3.1. Moduł (komponent)

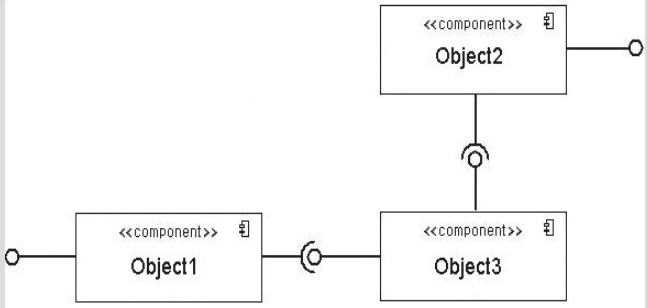
Pojęcie modułu (rys. 2) jest podstawowe dla metody. Całość projektu oprogramowania w HOOD jest siecią współpracujących modułów, z których każdy definiuje funkcjonalność poprzez udostępniany interfejs. Moduł interpretowany jest w UML 2.0 jako komponent i nazwa ta będzie dalej stosowana. W skład udostępnianego interfejsu wchodzą operacje, które mogą być wywoływane przez inne komponenty. *Required interface* (rys. 2) to zbiór operacji wymagany przez dany komponent do realizacji swojej wewnętrznej funkcjonalności.



Rys. 2. Komponent HOOD

3.2. Relacja użycia

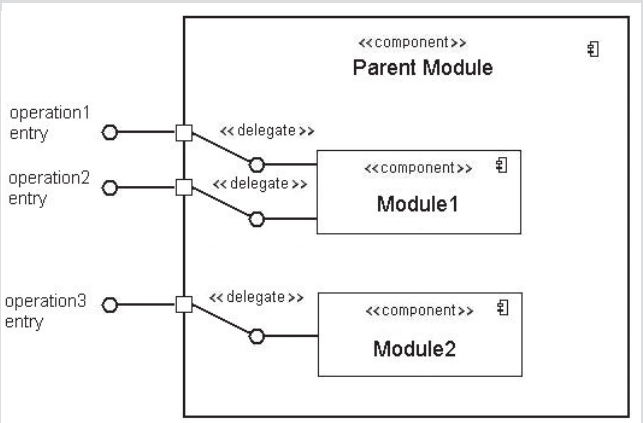
Jeśli dany komponent wywołuje operację z interfejsów innych komponentów, to jest z nimi w relacji użycia (*use relation*) (rys. 3). Relacja ta decyduje o kierunku przepływu sterowania pomiędzy komponentami.



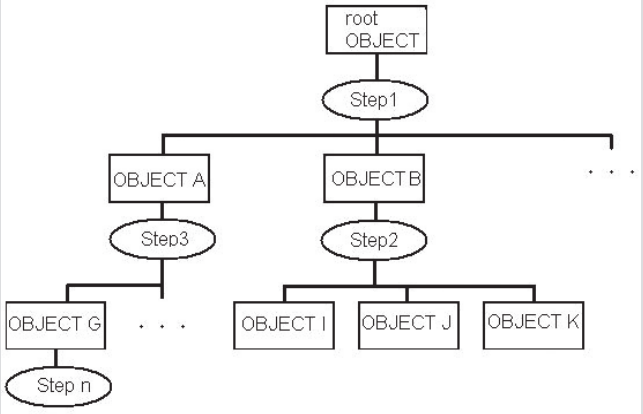
Rys. 3. Relacja użycia

3.3. Relacja zawierania – proces projektowy

Drugą podstawową zasadą w metodzie HOOD jest stosowanie dekompozycji komponentów. W projekcie istnieje hierarchia rodzic – potomek (rys. 4), która definiuje drzewo projektowe HDT (*HOOD Design Tree*) (rys. 5). Projekt rozpoczyna się od zdefiniowania komponentu – korzenia tego drzewa przy wykorzystaniu artefaktów wcześniejszego etapu specyfikacji wymagań. Powstaje tzw. konfiguracja systemu. Następnie w wyniku dekompozycji korzenia powstają jego komponenty potomne, które dalej mogą być także dekomponowane. W ten sposób powstaje struktura drzewa HDT, gdzie węzłami są grupy komponentów tej samej warstwy abstrakcji. Projekt kończy się z chwilą zdefiniowania komponentów terminalnych (najniższej warstwy), które będą implementowane, grupowane i alokowane w określonej architekturze fizycznej.



Rys. 4. Relacja zawierania



Rys. 5. Drzewo projektowe HOOD

3.4. Klasy komponentów

HOOD wyróżnia dwie klasy komponentów:

1) komponent aktywny,
2) komponent pasywny.

Komponent aktywny to taki, który posiada co najmniej jedną operację ograniczaną. Ograniczenie nakładane na operacje określa sposób, w jaki inne komponenty mogą korzystać z udostępnianego interfejsu danego modułu. Komponent pasywny to taki, którego operacje udostępniane nie są ograniczone. Klasy te mogą być także zdefiniowane w terminach wątków wykonania. Wątek w HOOD jest ciągiem wywołań operacji pomiędzy komponentami. Jedynie komponenty aktywne mogą inicjować taki ciąg. Moduły pasywne tworzą kod, który może być tylko wywołany przez inny moduł.

3.5. Model dynamiczny komponentu

HOOD stosuje ścisły rozdział opisu ciała operacji danego komponentu od części sterującej przepływem sterowania między komponentami w projekcie.

OPCS – ciało operacji

Dla każdej operacji udostępnianej w danym komponencie wprowadzona zostaje osobna struktura OPCS (*Operation Control Structure*). OPCS jest sekwencyjnym kodem realizującym określony algorytm i być może korzystającym z OPCS zewnętrznych komponentów.

OBCS – kontrola przepływu sterownia

Kontrola przepływu sterownia odbywa się za pośrednictwem struktury OBCS (*Object Control Structure*). Opisuje ona ograniczenia nakładane na operacje udostępniane przez dany komponent aktywny. Gdy klient żąda wykonania operacji ograniczanej, OBCS serwera narzuca odpowiednie warunki wykonania OPCS. OBCS istnieje tylko dla komponentu aktywnego, co wynika wprost z jego definicji.

3.6. Ograniczenia operacji

Ograniczenia przypisane operacji definiują sposób, w jaki klient synchronizuje się serwerem. Zbiór ograniczeń w projekcie określa tzw. model dynamiczny (behawioralny) HOOD i składa się ze wszystkich struktur OBCS przypisanym komponentom aktywnym w projekcie.

Ograniczenia stanem

Ten typ ograniczeń opisywany jest w HOOD diagramami maszyn skończenie stanowych FSM (*Finit State Machine*). Dla danego komponentu budowana jest maszyna stanowa, w której przejścia pomiędzy stanami etykietowane są operacjami udostępnianymi w interfejsie. Operacje te tworzą grupę usług komponentu ograniczanych stanem. Ich wywołanie przez zewnętrznego klienta powoduje zmianę stanu w FSM serwera.

Ograniczenia protokołem

Poniżej opisano trzy protokoły stosowane do ograniczeń operacji.

1) Protokół ASER (*Asynchronous Exectution Request*).
   Klient żądający wywołania operacji nie jest blokowany przez serwer. Żądanie jest natomiast kolejkowane i wykonywane w późniejszym czasie. Dyscyplina kolejki zależy od konkretnej implementacji komponentu na poziomie terminalnym.
2) Protokół LSER (*Loosly Synchronous Execution Request*).
   Klient żądający wywołania operacji oczekuje na potwierdzenie rozpoczęcia jej realizacji przez serwer. Po otrzymaniu tego potwierdzenia klient nie jest dalej blokowany.
3) Protokół HSER (*Highly Synchronous Execution Request*).
   Klient żądający wywołania operacji jest blokowany aż do momentu, w którym serwer potwierdzi zakończenie jej wykonania.

Istnieją także ograniczenia związane z przekroczeniem oczekiwania klienta na potwierdzenie lub zakończenie wykonania operacji w przypadku LSER i HSER. Jednak ograniczenia te nie będą rozważane w dalszej części pracy.

Ograniczenia współbieżnością

Poniżej opisano krótko trzy ograniczenia (MTEX, RWER, ROER) związane ze współbieżnym wywoływaniem operacji.

1) MTEX (*Mutual Exclusion Execution Request*).
   Jest to blokada nałożona na operacje gwarantująca, że podczas jej wykonywania żadne inne żądania jej wykonania nie będą realizowane.
2) RWER (*Read-Write Execution Request*).
   Ograniczenie RWER stosuje się dla operacji zmieniających wewnętrzny stan komponentu serwera. Oznacza to, że podczas wykonywania ograniczonej operacji żadne inne żądanie wykonania operacji rodzaju RWER lub ROER nie będzie realizowane. Ograniczenie dotyczy wszystkich operacji wymienionych typów w komponencie serwera.
3) ROER (*Read Only Execution Request*).
   ROER oznacza, że operacja może być wywoływana przez kilka komponentów równocześnie pod warunkiem, że w tym samym czasie nie jest wykonywana operacja typu RWER.

4. JĘZYK LOTOS

LOTOS jest językiem specyfikacji architektury i funkcjonowania systemów rozproszonych [2, 3, 7]. Język ten ma obecnie wiele różnych zastosowań [1], i podobnie jak ESTEREL i SDL (*Specification and Description Language*) może być stosowany do opisu skomplikowanych systemów informatycznych, w tym także SCR. Ograniczony rozmiar pracy nie pozwala na prezentację składni i semantyki LOTOS. Poniżej zaprezentowano jedynie prosty przykład programu w LOTOS wraz ze stosownym komentarzem. Pełny opis języka można znaleźć w dokumencie [7] zawierającym jego formalną definicję. Praktyczne wprowadzenie do podstawowych konstrukcji podano także w [3].

4.1. Przykładowa specyfikacja systemu w LOTOS

Program LOTOS skład się z dwóch części:

1) części procesowej,
2) części definiującej abstrakcyjne typ danych wykorzystywane w procesach.

Na najwyższym poziomie języka definicje te ujęte są w formie jednostki zwanej specyfikacją.

specification SpecExampl[*lista-portów*] (*lista-zmiennych*) :
noexit
(definicje typów danych *)*
  behaviour
(definicje procesów *)*
  endspec

Listy-portów deklarują dostępne porty wyrażenia procesowego znajdującego się po słowie ‘behaviour’. Odpowiednio lista-zmiennych jest deklaracją podająca nazwy wykorzystywanych zmiennych wraz z ich typami. Poniżej podano przykładową definicję procesu CommExample wraz z procesami składowymi: sender i reciver.

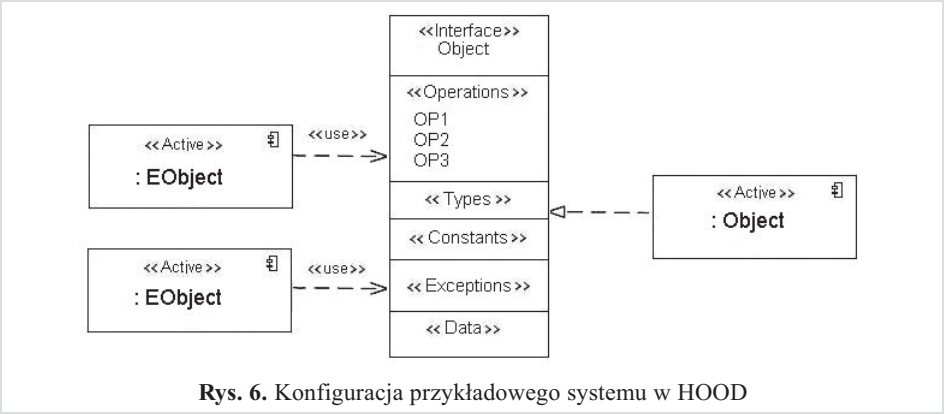
```
process CommExample [S, A] (St: State): noexit :=  (* Definicja procesu o portach S, A i zmiennej St typu State *)
        sender[S, A] (S0) |[S,A]|                  (* Wyrażenie procesowe: proces sender synchronizuje się  *)
        ( recivier[S,A] (S0)  |||  reciver[S,A](S0) ) (* na portach S, A z dwoma działającymi równolegle   *)
                                                   (* procesami reciver. *)
where
        process sender [S,A] (St:State) : noexit :=  (* Początek definicji procesu sender.*)
                S ! M1 ;                           (* Przesłanie wartości M1 przez port S. *)
                A ? Ack: Message ;                 (* Oczekiwanie na potwierdzenie na porcie A. Otrzymywana *)
                                                   (* wartość musi być typu Messie. *)
                sender [S,A] (S1)                  (* Rekurencyjne wywołanie procesu  z nową wartością *)
                                                   (* zmiennej St. *)
        endproc                                    (* Koniec definicji procesu sender*)

        process reciver[S,A] (St:State) : noexit :=
                S? M : Message [eqv(M,M1)];        (* Port  S oczekuje  na przejęcie wartości typu Message. *)
                                                   (* Do synchronizacji dojdzie gdy operacja eqv zwróci wart. *)
                                                   (* true * )
                A ! M1Ack;
                reciver[S,A](S1)
        endproc
endproc
```

Definicja typu danych wykorzystywanych przez procesy określa elementy pewnego zbioru i dozwolone operacje na jego elementach. Semantyka tych operacji określana jest równościowo tak jak to mam miejsce np. w arytmetyce liczb naturalnych. Więcej informacji na temat abstrakcyjnych typów danych i teorii specyfikacji algebraicznej, z której korzysta język LOTOS, można znaleźć w [10].

5. INTERPRETACJA STRUKTUR HOOD W LOTOS

Przedstawiona została interpretacja wybranych konstrukcji HOOD w języku LOTOS. Interpretacja jest przeprowadzona dla przykładowego widoku konfiguracji systemu w HOOD.

```
type MessageType is                                (* Początek definicji typu *)
   BOOLEAN                                         (* Wykorzystanie zdefiniowanego wcześniej typu BOOLEAN *)
   sorts Message                                   (* Sort  elementów  MessageType*)
   opns                                            (* Definicje operacji *)
      M1          : -> Message                     (* Operacja bezargumentowa (stała)*)
      M!Ackw      : -> Message
      eqv         : Message ,Message -> BOOL       (* Operacja dwuargumentowa *)
   eqns                                            (* Semantyka równościowa operacji *)
   forall x,y: Message
   ofsort BOOL
      eqv(x,x) = TRUE;
      eqv(x,y) = FALSE;
   endtype                                         (* Koniec definicji typu* )
```



Rys. 6. Konfiguracja przykładowego systemu w HOOD

5.1. Przykład projektu w HOOD

Rysunek 6 przedstawia konfigurację systemu w HOOD. Jest to pierwszy poziom drzewa HDT określający system do zaprojektowania wraz z jego otoczeniem.

Komponent Object stanowi korzeń HDT. Jego interfejs składa się z trzech operacji: OP1, OP2, OP3 wywoływanych przez środowisko, które stanowią dwa komponenty EObject. Ich interfejsy zostały pominięte.

5.2. Formalizacja

W tablicy 1 przedstawiono diagramy i kod LOTOS odpowiadający modelowi z rysunku 6. Konstrukcje językowe nieistotne lub oczywiste na danym poziomie opisu zostały pominięte i zastąpione kropkami lub komentarzem. Nazwę modelowanej struktury HOOD, a także znaczenie poszczególnych konstrukcji LOTOS zawarto w komentarzu.

Model w tablicy 1 wykorzystuje dane, których typy należy także zdefiniować. Najistotniejszy jest tutaj typ o sygnaturze StateType, który określa stany komponentu i dopuszczalne między nimi przejścia, a więc definiuje FSM komponentu Object. Definicja ta została przedstawiona w tablicy 2.

6. WNIOSKI

W pracy zaprezentowano możliwość użycia języka LOTOS w projektowaniu oprogramowania systemów czasu rzeczywistego metodą HOOD. Podstawowym zagadnieniem w hierarchicznym modelowaniu (np. w HOOD) jest zachowywanie spójności między kolejnymi warstwami drzewa komponentów (HDT). Implementacja modułu w postaci zbioru jego potomków musi spełniać funkcjonalność modułu rodzica. Zbadanie tego spełniania nie jest możliwe przy wykorzystaniu środków dostarczanych jedynie przez HOOD. Pojawia się potrzeba wyrażenia modułów i relacji je wiążących w notacji formalnej, umożliwiającej analizę projektu, a w szczególności pozwalającej rozstrzygnąć, czy niższe poziomy drzewa projektowego zachowują funkcjonalność poziomów wyższych. Proponowana w pracy metoda umożliwia realizację tego zadania.

6.1. Walidacja semantyki w LOTOS

Analiza modelu formalnego HOOD może dostarczyć sensownych wyników i ocen projektu w tej metodzie tylko wówczas, gdy zaproponowany model zachowuje semantykę poszczególnych struktur HOOD. Kwestia ta nie jest w pracy dyskutowana. Przy budowie modelu w LOTOS oparto się na opisie słownym i diagramach zawartych w [6] i [13]. Informacje te wydają się wystarczające do celów formalizacji przedstawionej w pracy. Przedstawiony model formalny jest pewną propozycją, która nie wyklucza innych możliwych interpretacji w LOTOS.

6.2. Modelowanie struktur HOOD

W budowie modelu formalnego wykorzystano zarówno część procesową, jak i część definiującą abstrakcyjne typy danych w LOTOS. Procesy zostały użyte do określenia podjednostek modelu komponentu, odpowiedzialnych za przepływ sterowania. Typy danych wykorzystano między innymi do zamodelowania maszyny skończenie stanowej. Takie wykorzystanie typów umożliwia łatwe wyrażenie stanu komponentu i funkcji przejścia między stanami w postaci zbioru równań algebraicznych. Alternatywą jest tutaj tworzenie specjalnego procesu, którego sekwencje synchronizacji w wewnętrznych portach odpowiadałyby zmianom stanu. Podejście to jednak napotyka szereg trudności. Przykładowo należałoby wypisać *explicite* wszystkie możliwe sekwencje stanów, co przy dużej ich liczbie nie jest łatwą czynnością.

6.3. Dalsze prace

Główne kierunki dalszych prac dotyczą bardziej kompletnego określenia procesu projektowego, jak również automatyzacji procesu translacji.

Proces projektowy

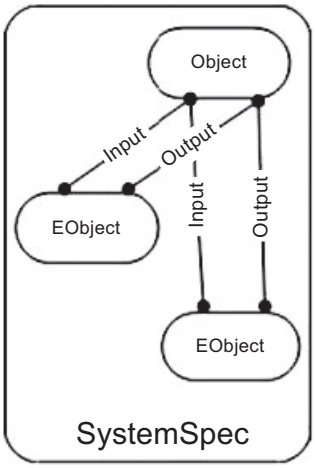
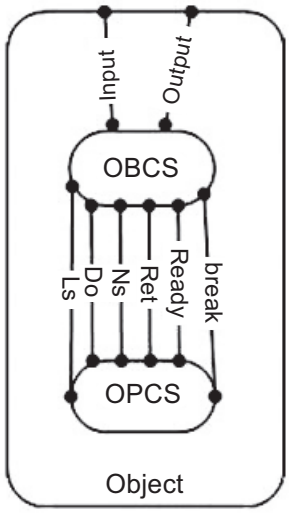
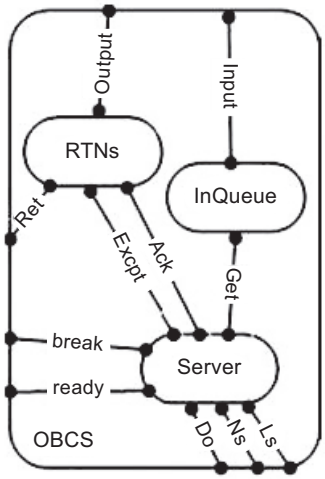
Zaproponowany model formalny przedstawiono w pracy w oderwaniu od właściwego procesu projektowego HOOD. Możliwe jest tutaj osadzenie procedury opierającej się na modelu LOTOS, która umożliwi weryfikację artefaktów powstałych na danym poziomie drzewa projektowego. Powstała w ten sposób metoda będzie wykorzystywać informacje zwrotną generowaną automatycznie przez narzędzie weryfikujące kod LOTOS w celu poprawy modelu komponentów w HOOD.

Automatyzacja

Model LOTOS może być generowany automatycznie na podstawie diagramów i dokumentacji HOOD. Na dalszym etapie możliwe jest zintegrowanie narzędzi automatycznie weryfikujących kod LOTOS. Przykładem może być tutaj wyspecjalizowany pakiet CADP (*Construction and Analysis of Distributed Processes*) [1].

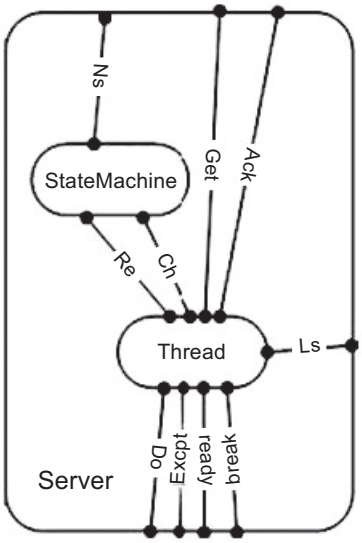
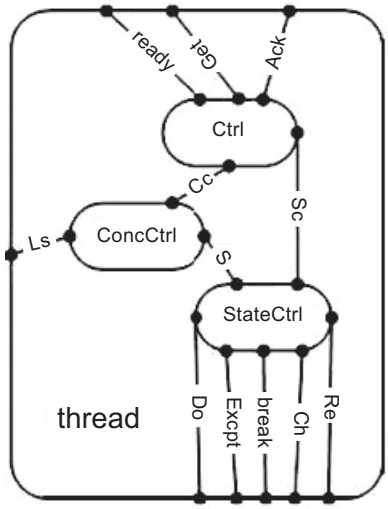
Tablica 1

Formalizacja komponentu

Diagram LOTOS	**Kod LOTOS**	**Komentarz**
	specification **SystemSpec** [...] (...) : noexit library TYPES endlib behaviour **Object**[Input,Output] (…) |[input,output]| (**EObject** [Input,Output] (…) ||| **EObject** [Input,Output] (…)) where ... endproc	Modelowana struktura: Konfiguracja Systemu Zachowanie: Proces Object synchronizuje się z dwoma instancjami procesu EObject. Port Input służy do zgłaszania żądań wykonania operacji, port Outpout do przekazania wyników jej wykonania lub potwierdzenia rozpoczęcia wykonania. Obiekty EObject wykonują się równolegle bez synchronizacji między sobą
	process **Object**[...](...): noexit := ... **OPCS** [LS,Do,Ns,Ret,Ready,break] (...) |[Do,Ls,Ns,Ret,Ready,break]| **OBCS** [Input, Output,Do, Ns, Ls, Ret, Ready,break] (...) where ... endproc	Modelowana struktura HOOD: korzeń systemu Zachowanie: Proces OBCS steruje przyjmowaniem żądań i zwracaniem wyników lub potwierdzeń na zewnątrz. Synchronizuje się z procesem OPCS, który modeluje wykonie właściwego ciała danej operacji. Realizacja narzuconych na operacje ograniczeń realizowana jest wewnątrz OBCS
	process **OBCS** [...] (...) **InQueue** [Input, Get] (...) |[Get]| **Server** [Get, Ack, Excpt,...] (...) |[Excpt,Ack]| **RTNs**[Excpt,Ack, …] (...) where ... endproc	Modelowana struktura HOOD: OBCS komponentu Zachowanie: Proces InQueuee kolejkuje przychodzące zgłoszenia. Proces Server pobiera je (gdy wolny jest port Get) i wysyła do realizacji na zewnątrz przez port Do. Do procesu RTNs wysyła natomiast informacje o wyjątku przez port Excpt bądź potwierdzenie rozpoczęcia wykonywania operacji przez port Ack. Proces RTNs, po wykonaniu danej operacji, otrzymuje jej wyniki na porcie Ret, a następnie udostępnia je na porcie Output

Tablica 1 cd.

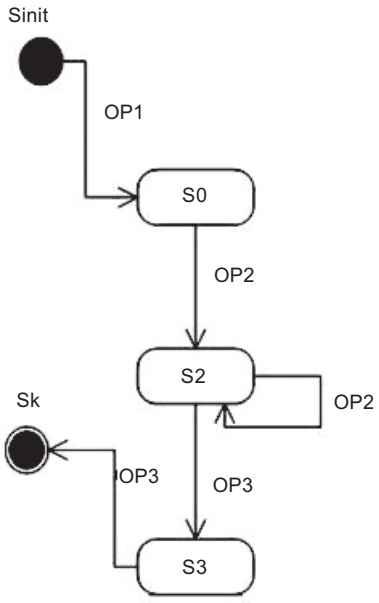
Formalizacja komponentu

Diagram LOTOS	**Kod LOTOS**	**Komentarz**
	process **Server** [...] (...) : noexit := **StateMachine** [Ch, Re, …] (…) |[Ch, Re]| (**thread** [Ch, Re,…]) where ... endproc	Modelowana struktura HOOD: ograniczenia operacji, maszyna skończenie stanowa komponentu Zachowanie: Proces thread steruje sprawdzaniem wszystkich trzech typów ograniczeń. Ograniczenia protokołu i są sprawdzane wewnątrz thread. Ograniczenia stanu są sprawdzane w procesie StateMachine
	process **thread** […] : noexit := **Ctrl** [Get,Ack, Cc, Sc, Ready] |[Cc,Sc]| (**concCtrl** [Cc, Ls, S] |[S]| **stateCtrl** [Sc, S, …]) where process **Ctrl** [Cc,Sc,…] : noexit := … (* sprawdzenie ograniczen *) (* protokolu *) endproc process **concCtrl** [Cc, S] : noexit := … (*sprawdzenie ograniczeń *) (* współbieżności *) endproc process **stateCtrl** […] : noexit := … (*sprawdzenie ogr. stanu*) endproc endproc	Modelowana struktura HOOD: jak wyżej Zachowanie: Ograniczenia są sprawdzane w kolejności: protokół, współbieżność, stan

Tablica 2

Definicja typu StateType

Diagram FSM	**Kod LOTOS**	**Komentarz**

	type **StateType** is
	BOOLEAN,OperationType
	sorts State
	opns
	check:
	Operation, State -> State
	fire : Operation, State -> State
	S0, S2, S3, Sinit, Sk : -> State
	stateOk : -> State
	eqv : State, State -> BOOL eqns
	forall x : Operation,
	y,z :State
	
	ofsort State
	
	check (OP1, Sinit) = stateOK;
	check (OP2, S0) = stateOK;
	check (OP2, S2) = stateOK;
	check (OP3, S2) = stateOK;
	check (OP3, S3) = steteOK;
	
	fire (OP1, Sinit) = S0;
	fire(OP2, S0) = S2;
	fire (OP2, S2) = S2;
	fire (OP3, S2) = S3;
	fire (OP3, S3) = Sk;
	ofsort Bool
	eqv(y,y) = true;
	eqv(y,z) = false;
	
	endtype
		Modelowana struktura HOOD:
		stan, funkcja przejścia
		
		Opis:
		Typ definiuje sort elementów o nazwie Sate. Operacja check jest wykorzystywana do sprawdzenia ograniczeń stanu i zwraca element stateOK, gdy przejście jest dopuszczalne dla danego stanu i operacji. Operacja fire realizuje funkcje przejścia między stanami. Operacja eqv służy w części procesowej do sprawdzania równości pomiędzy stanami.
		Przykładowo:
		eqv(S, StateOK) = false
		oznacza, ze S jest różny od StateOK

Literatura

[1] CADP Home Page. http://www.inrialpes.fr/vasy/cadp/

[2] Garavel H.: *Compilation et verification de programmes LOTOS.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), 1989

[3] Garavel H.: *Presentation du Language LOTOS.* Annexes A et B de [2], 1993

[4] Gomaa H.: *Design Concurrent, Distributed, and Real-Time Application with UML.* George Mason University, Addison Weseley, 2000

[5] Hoare C.A.R.: *Communicating Sequential Processes.* Prentice Hall, 1985

[6] Hood Technical Group, Hood Reference Manual, Release 4

[7] ISO. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneve, 1988.

[8] Jensen K.: *Coloured Petri nets, Basic Concepts, Analysis Methods and Practical Use.* Vol. 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 1997

[9] Klimek R.: *Wprowadzenie do logiki temporalnej.* Kraków, UWND AGH 1999

[10] Mañas J.A.: *A Tutorial on ADT Semantics for LOTOS Users – Part I and II: Fundamental Concepts.* Technical Report, Dpt. Ingenieria Telematica, E.T.S.I. Telecommunicion, Madrid, Spain, 1988

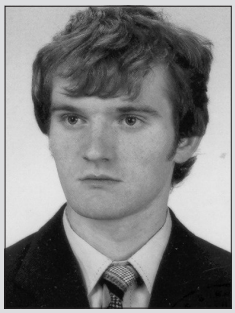
[11] Milner R.: *Communication and Concurrency.* Prentice Hall International, 1989

[12] OMG. Unified Modelling Language: Superstructure, version 2.0, 2005

[13] Rosen J-P.: *HOOD An Industrial approach for software design.* HOOD Technical Group, 1997

[14] Szmuc T.: *Modele i Metody Inżynierii Oprogramowania Systemów Czasu Rzeczywistego.* Kraków, UWND AGH 2001

Wpłynęło: 5.04.2006



Rafał BRZUCHACZ

W latach 1998–2003 studiowałem automatykę na Wydziale EAIiE Akademii Górniczo-Hutniczej w Krakowie. Obrana przeze mnie specjalność to informatyka w sterowaniu i zarządzaniu. W 2003 roku obroniłem pracę dyplomową pt. *Formalne modele obliczeniowe w informatyce*. W 2003 roku rozpocząłem studia doktoranckie z automatyki na Wydziale EAIiE AGH. Moje zainteresowania naukowe to informatyka czasu rzeczywistego, metody weryfikacji oprogramowania, języki algebraicznej specyfikacji programów.

e-mail: rafalb@nova.ia.agh.edu.pl