AUTOMATYKA • 2011 • Tom 15 • Zeszyt 3

Leszek Kotulski*, Adam Sędziwy*

Algorytm scalania diagramów stanu w modelowaniu systemów wbudowanych

1. Wprowadzenie

Systemy komputerowe i ich oprogramowanie stają się coraz bardziej skomplikowane w szczególności z uwagi na równoległą pracę ich komponentów i konieczność synchronizacji pracy. Z tego względu inżynieria programowania wprowadza szereg metod wspomagających rozwój oprogramowania [1]. Wspomagają one precyzyjną specyfikację systemu (często z wykorzystaniem narzędzi formalnych) oraz określają jak, bazując na tej specyfikacji, zaimplementować oprogramowanie. W większości przypadków równolegle z opracowaniem kodu źródłowego systemu zalecane jest opracowanie testów badających poprawność tego systemu. Spowodowane jest to faktem, iż nie ma bezpośredniego powiązania pomiędzy modelem formalnym systemu a oprogramowaniem reprezentującym go – czynnik ludzki występujący w procesie translacji tych opisów niestety często prowadzi do błędów. Kolejnym problemem jest też fakt, że model formalny wyrażony w terminologii algebr procesów jest złożony i mało intuicyjny z punktu widzenia informatyka, gdyż wymaga od niego dużej wiedzy matematycznej.

W projekcie Alvis [2, 3] zdefiniowano język programowania, którego konstrukcje równolegle z generacją kodu pozwalają na opracowanie modelu formalnego w postaci tzw. diagramów LTS (*Labbeled Transition System*) dla każdego agenta. W niniejszej pracy omówimy algorytm pozwalający wygenerować globalny LTS dla systemu składającego się z kilku agentów. W szczególności omówione zostaną kwestie związane z efektywnością obliczeniową i pamięciową tego podejścia.

Struktura pracy jest następująca. W rozdziale 2 zawarto krótkie wprowadzenie do języka Alvis, a także omówiono jego podstawowe pojęcia. W rozdziale 3 przedstawiono problem generowania złożonych stanów LTS i algorytm ich generacji. Ostatni rozdział zawiera podsumowanie pracy.

* AGH Akademia Górniczo-Hutnicza, Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki, Katedra Automatyki, al. A. Mickiewicza 30, 30-059 Kraków

2. Język modelowania Alvis

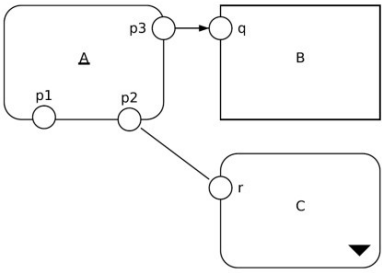
Językiem opisu systemów wbudowanych, który stanowi podstawę formalną dalszych rozważań jest Alvis [2]. Łączy on w sobie możliwości graficznego modelowania systemów i ich opisu za pomocą wysokopoziomowego języka programowania. Model zbudowany za pomocą Alvisa jest modelem trójwarstwowym. W warstwie graficznej definiowane są dane oraz interakcje zachodzące między agentami rozumianymi jako fragmenty rozważanego systemu. Warstwa kodu służy opisowi zachowania konkretnego agenta. Jest ona zrealizowana w języku opartym na składni języka Haskell [5] oraz języka ADA [4]. Za pomocą języka Haskell definiuje się typy danych dla poszczególnych parametrów oraz funkcje, do obsługi komunikacji międzyagentowej zastosowano wymianę komunikatów z mechanizmem rendezvous. W trzeciej, systemowej warstwie modelu stworzonego w języku Alvis mamy do czynienia z opisem wszystkich agentów obecnych w systemie i ich stanów. Ważną cechą warstwy trzeciej jest możliwość generacji tzw. grafów LTS opisujących zmiany stanów dla poszczególnych agentów wchodzących w skład rozważanego systemu. Do analizy zachowania tego systemu, będącego wynikiem współpracy między poszczególnymi agentami, a także jego formalnej weryfikacji, konieczne jest uzyskanie *złożonego* grafu LTS, reprezentującego stany kilku (wszystkich) agentów jednocześnie. Trudność generacji grafu złożonego wynika z faktu, iż nawet dla niewielkiej liczby pojedynczych grafów LTS, cechujących się niewielką liczbą stanów, graf złożony ma zwykle duży rozmiar, wykluczający jego „ręczne" wyprowadzenie. Z uwagi na to niezbędne stało się opracowanie metody scalania, która pozwoliłaby na automatyzację procesu łączenia takich pojedynczych diagramów LTS. Celem niniejszej pracy jest prezentacja takiego algorytmu z uwzględnieniem szczegółów implementacyjnych wpływających na efektywność metody.

2.1. Diagramy komunikacji

Diagramy komunikacji stanowią wizualny składnik modelu zrealizowanego w języku Alvis i opisującego rozważany system. Ich celem jest zobrazowanie kanałów komunikacji między agentami. Diagram komunikacji jest grafem hierarchicznym, którego węzły reprezentują zarówno agentów, jak i części modelu niższego poziomu. Za pomocą diagramów możliwe jest agregowanie zbioru agentów do postaci pojedynczego modułu, będącego także agentem, określanym jako hierarchiczny Diagramy komunikacji zawierające agenty hierarchiczne są określane mianem diagramów hierarchicznych. Diagramy o strukturze płaskiej są diagramami niehierarchicznymi. Niniejsza praca koncentruje się na takim właśnie przypadku. Należy pamiętać, że do pełnej specyfikacji modelu, oprócz diagramów komunikacji, niezbędna jest warstwa kodu określająca zachowanie agentów.

Przyjęto konwencję, według której agenty będące węzłami diagramu komunikacji reprezentowane są za pomocą prostokątów (*pasywne*), prostokątów z zaokrąglonymi narożnikami (*aktywne*) oraz prostokątów z zaokrąglonymi narożnikami i trójkątem w prawym dolnym

rogu (*hierarchiczne*). Dodatkowo, każdy z agentów ma porty służące komunikacji, oznaczone za pomocą okręgów umieszczonych na krawędziach figury danego prostokąta. Kierunek przesyłu informacji w danym kanale komunikacji określa strzałka. Jeśli porty połączone są zwykłą linią, oznacza to, że komunikacja odbywa się w obu kierunkach.



Rys. 1. Hierarchiczny diagram komunikacji złożony z agenta aktywnego (A), pasywnego (B) oraz hierarchicznego (C)

Na rysunku 1 przedstawiono przykładowy diagram komunikacji reprezentujący wszystkie, omawiane wyżej obiekty.

2.2. Warstwa kodu

Warstwa kodu oparta jest na języku Haskell, Ada, a także zawiera oryginalne polecenia języka Alvis. Zachowanie każdego nie hierarchicznego agenta musi być opisane w warstwie kodu. Tabela 1 ilustruje zbiór najczęściej używanych poleceń warstwy kodu języka Alvis [2]. Dla uproszczenia przyjęto, że **p** oznacza nazwę portu, **x** jest nazwą parametru, **g1**, **g2**, ... oznaczają predykaty (warunki logiczne), **e** jest wyrażeniem, natomiast **ms** to milisekundy (dotyczy długości interwałów czasowych dla niektórych instrukcji występujących w kodzie).

Tabela 1
Alfabetyczna lista najczęstszych poleceń języka Alvis

Polecenie	Opis
`cli`	Wyłącza obsługę przerwań
`critical {...}`	Definiuje sekcję krytyczną
`delay ms`	Wstrzymuje wykonanie kodu na określoną liczbę `ms`
`exec x = e`	Oblicza wyrażenie, a wynik wstawia do zmiennej `x` (słowo kluczowe `exec` może być pominięte)
`exit`	Zamyka agenta wykonującego to polecenie

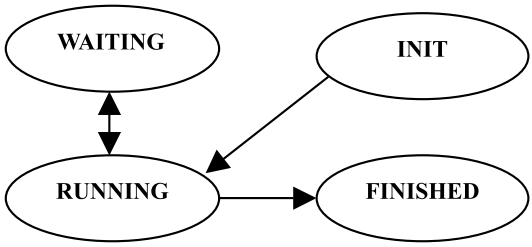
`if (g1) {...}`
`elseif(g2)`
`{...}`
`...`
`else {...}`
	Blok instrukcji warunkowych

Tabela 1 cd.

Polecenie	Opis
`in p`	Odbiera sygnał przez port **p**
`in p x`	Odbiera sygnał przez port **p**, a otrzymaną wartość przypisuje parametrowi **x**
`jump label`	Przekazuje kontrolę do linii kodu oznaczonej etykietą **label**
`jump far A`	Przekazuje kontrolę do agenta **A**
`loop (g) {...}`	Wykonuje pętlę, dopóki warunek **g** jest spełniony
`loop (every ms) {...}`	Wykonuje pętlę co określoną liczbę **ms**
`loop {...}`	Pętla nieskończona
`null`	Instrukcja pusta
`out p`	Wysłanie sygnału przez port **p**
`out p x`	Wysłanie wartości parametru **x** przez port **p**
`proc (g) p {...}`	Dot. agenta pasywnego. Definiuje procedurę skojarzoną z portem **p**. Warunek **g** jest opcjonalny
`select { alt (g1) {...} alt (g2) {...} ... }`	Wybiera jeden z warunków
`start`	Uruchamia agenta **A**, jeśli jest on w stanie INIT, w przeciwnym razie nie robi nic
`sti`	Włącza obsługę przerwań

2.3. Stan agenta i przejścia między stanami

W przypadku systemu złożonego jedynie z agentów aktywnych każdy z nich może przebywać w jednym z następujących trybów pracy: *init*, *finished*, *running*, *waiting* (rys. 2).



Rys. 2. Możliwe tryby pracy agtenta i przejścia między nimi

Definicja 1. Stanem agenta X nazywamy czwórkę postaci

$$S = (am(X),\ pc(X),\ ci(X),\ pv(X)),$$

gdzie *am*(*X*) oznacza tryb pracy agenta, *pc*(*X*) jest licznikiem kroków programu opisującego aktywność agenta, *ci*(*X*) to lista zawierająca informację kontekstową dla danego stanu *X*, zaś *pv*(*X*) opisuje wartości parametrów w danym stanie.

Stan agenta może ulec zmianie na skutek wykonania kroku w jednym z poleceń znajdujących się kodzie. Z uwagi na fakt, iż część poleceń języka to polecenia jednokrokowe (np. **in**, **out**, **exec**, **exit**, **jump**, **null**, **start**), część zaś złożona jest z więcej niż jednego kroku (np. **if**, **loop**, **select**), konieczne jest przyjęcie zasady numerowania poleceń w kodzie języka Alvis. Numeracja ta znajduje swoje odbicie w wartości pola *pc*(*X*), opisującego stan agenta *X*. Za przykład posłużyć może kod pewnego agenta A, zamieszczony poniżej (rys. 3).

```
agent A {
i :: Int = 0 ;
x :: Int = 1 ;
loop ( x <> 0 ) {                                          → 1
 select {                                                  → 2
  alt ( i == 0 && ready[in(p)]) { in p x  ; i = 1 ; }      → 3 , 4
  alt ( i == 1 && ready[out(q)] ) { out q x ; i = 0 ; }    → 5 , 6
 }
 if ( i == 1 ) { out p ; }                                 → 7 , 8
 else { null ; }                                           → 9
}
exit ;                                                     → 10
}
```

Rys. 3. Przykładowy kod agenta wraz z numeracją kroków

W przypadku poleceń wielokrokowych, numeracja kroków przebiega w sposób rekurencyjny, tzn. jako pierwszy numerowany jest krok otwierający blok polecenia, następnie numeruje się kroki zawarte w bloku. Rysunek 3 ilustruje powyższą zasadę.

Aby przeprowadzić analizę zmian stanu agenta, wprowadza się następującą, uproszczoną notację. Stan agenta X oznacza się jako $S = (am, pc, ci, pv)$. Oznaczenie agenta, ujęte jako dolny indeks danej zmiennej, będzie figurowało tylko w przypadku, gdy będzie tego wymagał kontekst. Nowa wartość licznika kroku, wynikająca z wykonania *i*-tego przejścia, od stanu *S* do $S' = (am', pc', ci', pv')$, będzie oznaczona jako *nextpc*. Typ instrukcji wykonywanej w tym kroku jest zwracana przez funkcję *instr*(*i*). Dodatkowo ustala się, że p^* jest portem skojarzonym z portem *p*. Komunikacja między agentami odbywa się w trybie synchronicznym.

Wykonanie *i*-tego przejścia, $S \xrightarrow{i} S'$, przez agenta znajdującego się w trybie *running*, powoduje następującą zmianę stanu.

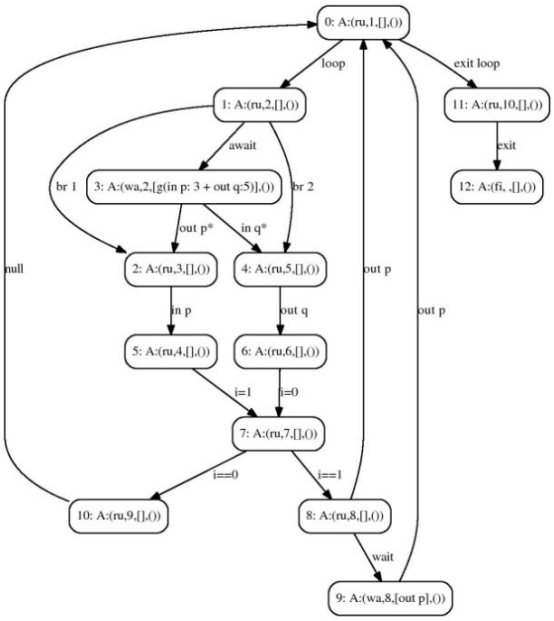
1) Jeśli *instr*(*i*) równe jest **`in p`**, wówczas może zajść jeden z poniższych przypadków:
   – Istnieje agent *Y* oczekujący na porcie *p**, aby nadać sygnał na port *p* agenta *X* (tj. *Y* zablokowany jest na instrukcji **`out p`***). W takiej sytuacji *X* odbiera sygnał z *p*, *pc*' = *nextpc*, *am*' = *running*. Warto zwrócić uwagę, że agent *Y* przechodzi do kolejnego kroku programu, a jego tryb zmnienia się z *waiting* na *running*.
   – Istnieje agent *Y*, oczekujący na pewnym warunku **g** instrukcji **`select`**, na pojawienie się agenta odbierającego na porcie *p* (czyli $ci_Y = [\mathbf{g}(...;out\ p^*{:}nextpc_Y;...])$. Zgodnie z semantyką języka Alvis, pojawienie się agenta *X* gotowego do odbioru na porcie *p*, spowoduje przejście *Y* do linii kodu $pc_Y' = nextpc_Y$, określonej w informacji kontekstowej, dodatkowo $am_Y' = running$ i $ci_Y' = []$. Tymczasem dla agenta *X*: $am_X' = waiting$ oraz $ci_X' = [in(p)]$.
   – Żaden agent nie czeka na nadanie sygnału do portu *p**. Wówczas *ci*' = [*in*(*p*)], *am*' = *waiting*.
2) Jeśli *instr*(*i*) równe jest **`out p`**, wówczas postępuje się tak samo jak w poprzednim punkcie, zamieniając ze sobą jedynie operacje *in* i *out*.
3) Jeśli *instr*(*i*) jest warunkiem **g** w instrukcji **`if g`**, **`elseif g`**, **`loop g`**, wówczas *pc'* przyjmuje wartość zależną od **g**.
4) Jeśli *instr*(*i*) jest warunkiem **g** w jednej z gałęzi instrukcji **`select`**.
   – Gdy zachodzi warunek **g**, wówczas *pc*' = *nextpc*. Należy zauważyć, że spełnienie warunku **g** zawierającego **`in p`** (**`out p`**) oznacza, że istnieje pewien agent *Y* oczekujący na nadawanie (odbieranie) do (z) portu *p**.
   – Gdy aktualnie nie zachodzi żaden z warunków instrukcji **`select`**, ale w wyniku aktywności innych agentów pewna ich ilość, związana z gałęziami b_1, ..., b_k może zajść w przyszłości, po spełnieniu warunków c_1, c_2, ..., c_k (związanych z operacjami **`in`**, **`out`**), wówczas w tanie *S'* będzie występować *am*' = *waiting* oraz $ci' = [g(c_1 : nextpc_1 + c_2 : nextpc_2 + ... + c_k : nextpc_k)]$. Suma występująca w okrągłym nawiasie oznacza alternatywę poszczególnych warunków, po zajściu których, wartość licznika kroku ustawiana jest stosownie do numeru spełnionego warunku.

2.4. Graf LTS

Definicja 2. Niech *S* i *S'* będą stanami pewnego agenta opisanego w modelu Alvis. *S'* jest **bezpośrednio osiągalny** z *S* wtedy i tylko wtedy, gdy istnieje przejście $t \in T$, takie że $S \xrightarrow{t} S'$, gdzie *T* jest zbiorem dopuszczalnych przejść dla rozważanego agenta. *S'* jest **osiągalny** z *S* wtedy i tylko wtedy, gdy istnieje ciąg stanów $S_1, S_2, ..., S_{k+1}$ oraz ciąg przejść $t_1, t_2, ..., t_k \in T$, takich że $S = S_1 \xrightarrow{t_1} S_2 \xrightarrow{t_2} ... \xrightarrow{t_k} S_{k+1} = S'$. Zbiór wszystkich stanów osiągalnych z pewnego stanu początkowego S_0 oznaczamy jako $R(S_0)$.

Definicja 3. Grafem LTS nazywamy graf *LTS* = (*V*, *E*, *L*), gdzie $V = R(S_0)$, $E = \{(S, t, S') : S \xrightarrow{t} S', t \in L \wedge S, S' \in R(S_0)\}$ oraz $L = T$.

LTS reprezentuje wszystkie stany osiągalne z pewnego stanu początkowego, oraz przejścia między nimi. Na rysunku 4 znajduje się graf LTS dla agenta A, którego kod przedstawiono na rysunku 3.



Rys. 4. LTS agenta A

3. Generacja złożonych grafów LTS

W niniejszym rozdziale przedstawione zostanie zagadnienie łączenia pojedynczych grafów LTS w jeden, złożony graf LTS, opisujący interakcje między agentami tworzącymi dany system.

Definicja 4. Niech będzie dany ciąg agentów $(A_1, A_2, ..., A_N)$ i ciąg odpowiadających im stanów $\vec{S} = (S_1, S_2, ..., S_N)$. $\vec{S}$ nazywamy **stanem złożonym**. Dodatkowo, przyjmuje się, że $(T_1, T_2, ..., AT_N)$ jest ciągiem zbiorów przejść dla poszczególnych agentów. Mówimy, że $\vec{S}' = (S_1', S_2, ..., S_N')$ jest **bezpośrednio osiągalny** z $\vec{S}$ jeśli zachodzą poniższe dwa warunki:

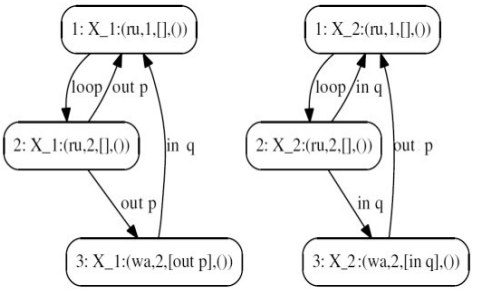
1) $\vec{S} \neq \vec{S}'$,

2) $\exists t \in \bigcup_{i=1}^{N} T_i \forall i \in \{1,...N\} : S_i \xrightarrow{t} S_i' \vee S_i = S_i'$.

Powyższą własność oznaczamy $\vec{S} \xrightarrow{t} \vec{S}'$. Stan $\vec{S}'$ jest **osiągalny** z $\vec{S}$, gdy istnieje ciąg stanów $\vec{S} = \vec{S}_1, \vec{S}_2, ..., \vec{S}_{k+1} = \vec{S}'$ oraz przejść $t_1, t_2, ..., t_k \in \bigcup_{i=1}^{N} T_i$, takich, że $\vec{S}_i \xrightarrow{t_i} \vec{S}_{i+1}$ dla i = 1, 2, ... N. Zbiór stanów osiągalnych z pewnego złożonego stanu początkowego oznaczamy jako $R(\vec{S}_0)$.

Definicja 5. Złożonym grafem LTS nazywamy graf $LTS_{comp} = (V, E, L)$, gdzie $V = R(\vec{S}_0), L = \bigcup_{i=1}^{N} T_i, E = \{(\vec{S}, t, \vec{S}') : \vec{S}, \vec{S}' \in V, t \in L\}$.

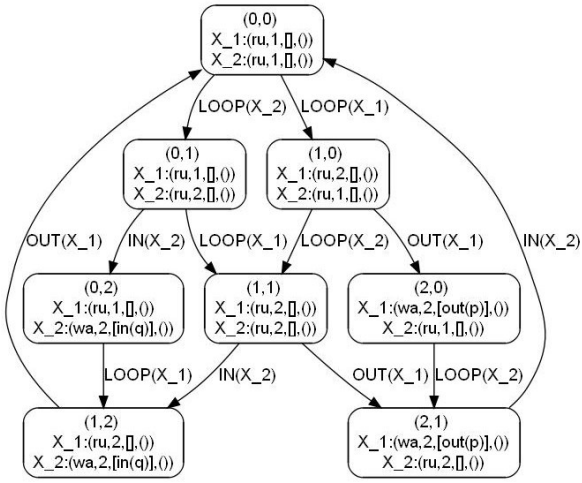
Na rysunku 5 przedstawiono grafy LTS agentów pisarza (X_1) i czytelnika (X_2) oraz odpowiadające im kody. Przyjmując, że początkowy stan złożony to $\boldsymbol{S_0}$ = (X_1:(ru,1,[],()), X_2:(ru,1,[],())) otrzymujemy złożony graf LTS pokazany na rysunku 6.




```
Agent X_1 {
  loop {     --1
     out p; --2
  }
}
```

```
Agent X_2 {
  loop {      --1
     in p*;  --2
  }
}
```

Rys. 5. Grafy LTS agentów oraz ich kody



Rys. 6. Złożony graf LTS dla modelu pisarz-czytelnik

3.1. Algorytm scalania

W niniejszych rozważaniach zakłada się, że w systemie występują jedynie agenty aktywne. Podstawą algorytmu generacji złożonego grafu LTS jest obserwacja, że przejście z danego stanu złożonego może wzbudzić jedynie agent znajdujący się w stanie *running*. Dla każdego takiego agenta *A* sprawdza się, czy jego przejście do następnego stanu powo-

duje przejścia pozostałych agentów. Jeśli tak, to wykonywane są one jednocześnie z przejściem *A*. Przedstawiony poniżej pseudokod, którego główną procedurą jest **Generuj** (rys. 7) realizuje taki właśnie scenariusz. Procedura **NastępneStany** (rys. 8) ma za zadanie określenie stanów bezpośrednio osiągalnych z danego stanu złożonego. Dla uproszczenia pseudokodu przyjęto, że $s \prec x, y$ oznacza zastąpienie odpowiednich stanów pojedynczych w *s*, stanami *x* i *y*.

```
Generuj(x_1^(0), x^(0)_2, ... x_N^(0))
   wejście: s^0 = (x_1^(0), x_2^(0), ... x_N^(0)) – ciąg stanów początkowych dla agentów
            X_1, X_2, ... X_N
   wyjście: G = (V, E) – złożony graf LTS dla X_1, X_2, ... X_N
 1 begin
 2     Q ← s^0;
 3     Oznacz wszytskie stany złożone jako nieodwiedzone;
 4     while Q jest niepusty do
 5         s ← Q.dequeue();
 6         Oznacz stan s jako odwiedzony;
 7         Jeśli s ∉ V dodaj s do V;
 8         S ← ∅;
 9         foreach stan running x, wchodzący w skład s do
10             S ← S∪NastępneStany(x, s);
11         foreach s' ∈ S do
12             Kolejkuj s' w Q jeśli s' jest nieodwiedzony;
13             if s' ∉ V then
14                 V ← V ∪ {s'};
15             E ← E ∪ {(s, s')};
16     G = (V, E);
17     return G;
```

Rys. 7. Procedura **Generuj**

Złożoność obliczeniowa powyższego algorytmu generacji wynosi $O(MN\prod_{i=1}^{N}|R(S_{0,i})|)$, gdzie *N* jest liczbą scalanych elementarnych grafów LTS, natomiast *M* to maksymalna liczba stanów bezpośrednio osiągalnych z jakiegokolwiek stanu jakiegokolwiek grafu elementarnego. Iloczyn wielkości zbiorów stanów osiągalnych wynika z linii 4 procedury **NastępneStany.**

Górnym ograniczeniem na ilość pamięci koniecznej na przechowanie złożonego grafu LTS jest $O(\prod_{i=1}^{N}|R(S_{0,i})|)$, przy czym w praktyce jest ona znacznie mniejsza, z uwagi na fakt, iż nie wszystkie punkty przestrzeni stanów są dozwolone, jak również nie wszystkie

przejścia między stanami dopuszczalnymi są możliwe. Zakładając, że średni stopień (wchodzący i wychodzący) węzła w k-wierzchołkowym złożonym grafie LTS wynosi d, otrzymujemy liczbę krawędzi $|E| = kd/2$. Zatem będzie on grafem rzadkim. Przykładowo, gęstość złożonego diagramu LTS o 100 stanach i średnim stopniu węzła 5 wynosi $D = d/(k-1) = 5/99 \approx 5\%$.

NastępneStany(x, s)

wejście: x – stan pojedynczego agenta, s – stan złożony
wyjście: S – zbiór stanów bezpośrednio osiągalnych z s

```
1  begin
2      X ← agent opisany stanem x;
3      if bieżąca instrukcja nie zawiera in/out then
4          foreach stan x' running, osiągalny bezpośrednio z x do
5              s' ← s ≺ x';
6              S ← S ∪ {s'};
7      else if bieżąca instrukcja zawiera in/out i pewien agent and Y oczekuje na
       X then
8          y ← bieżący stan Y;
9          x' ← stan running osiągalny bezpośrednio z x;
10         y' ← stan running osiągalny bezpośrednio z y;
11         s' ← s ≺ x', y';
12         S ← {s'};
13     else if bieżąca instrukcja zawiera in/out i żaden agent nie oczekuje na X
       then
14         x' ← stan waiting osiągalny bezpośrednio z x;
15         s' ← s ≺ x';
16         S ← {s'};
17     return S
```

Rys. 8. Procedura **`NastępneStany`**

3.2. Struktury danych

Algorytm generacji przestawiony powyżej został zaimplementowany w języku JAVA. Stan pojedynczego agenta reprezentuje klasa **`AtomicLTSState`**, natomiast stan złożony klasa **`CompositeLTSState`** będąca rozszerzeniem klasy **`ArrayList<AtomicLTSState>`**. Złożony graf LTS jest obiektem klasy **`DirectedSparseGraph<CompositeLTSState,Transition>`** stanowiącej parametryzację klasy generycznej, zdefiniowanej w grafowej bibliotece JUNG [6].

Z punktu widzenia strukturalnej złożoności kodu najistotniejszym jest odpowiednie zaprojektowanie struktur danych, które w maksymalnym stopniu uproszczą sprawdzanie warunków z linii 7 oraz 13 z kodu znajdującego się na rysunku 8. W tym celu utworzono klasę **Mappings** zawierającą trzy kluczowe pola:

1) `HashMap<PortType,PortType>` **port2port** – struktura odpowiedzialna za identyfikowanie portu skojarzonego z danym portem.
2) `HashMap<PortType,LTS>` **port2agent** – struktura mapująca zadany port na agenta-właściciela tego portu.
3) `HashMap<LTS,PortType []>` **agent2ports** – struktura zwracająca wszystkie porty będące w posiadaniu danego agenta.

Posiadając wyżej zdefiniowaną klasę, warunki z linii 7 i 13 procedury **NastępneStany**, oblicza się następująco. Niech bieżąca linia kodu zawiera instrukcję **in p**. Dodatkowo, niech Y oznacza agenta, który jest potencjalnym nadawcą sygnału do portu **p**. Wówczas $Y =$ **port2agent(port2port(p))**. Tryb pracy agenta Y (czyli am_Y) znaleźć można przez odczytanie z tablicy stanu złożonego, stanu odpowiadającego temu agentowi. Tablica haszująca została dodana do klasy w celu obsługi agentów pasywnych: dla szybkiego dostępu do portów danego agenta.

Należy zauważyć, że klasa **Mappings** pozwala w czasie stałym uzyskać informacje na temat portów i skojarzeń typu port-agent.

4. Podsumowanie

W niniejszej pracy przedstawiono algorytm automatycznej generacji złożonych grafów LTS, stanowiących składową modelu Alvis, opisującego działanie i własności kooperujących systemów wbudowanych. Szczególny nacisk położono na redukcję złożoności obliczeniowej, co zostało osiągnięte dzięki wprowadzeniu struktur o charakterze asocjacyjnym, pozwalających w czasie stałym uzyskiwać informacje kluczowe dla egzekucji algorytmu generacji. Struktury uwzględniają także możliwość rozbudowy algorytmu o agentów pasywnych.

W pracy, oprócz złożoności czasowej algorytmu, przedstawiono również oszacowanie złożoności pamięciowej związanej z przechowywaniem wygenerowanego grafu złożonego.

Literatura

[1] Sommerville I., *Software Engineering*. International Computer Science, Pearson Education Limited, 7th edition, 2004.
[2] Szpyrka M., Matyasik P., Mrówka R., Kotulski L. Balicki K., *Formal introduction to Alvis modelling language*. International Journal of Applied Mathematics and Computer Science, praca złożona do druku, 2011.

[3] Szpyrka M., Matyasik P., Mrówka R., *Alvis – modelling language for concurrent systems, Intelligent Decision Systems in Large-Scale Distributed Environments*. Studies in Computational Intelligence, Bouvry P., Gonzalez-Velez H., Kołodziej J. (Eds.), vol. 362, 315–342, Springer-Verlag, 2011.
[4] Ada – http://www.ada-auth.org
[5] Haskell – http://www.haskell.org.
[6] JUNG – http://jung.sourceforge.net.