Wykorzystanie hierarchicznego modelu współbieżnego automatu w projektowaniu sterowników cyfrowych

Wydział Elektrotechniki, Informatyki i Telekomunikacji
Uniwersytet Zielonogórski

Prace Naukowe z Automatyki i Informatyki
Tom 6

Grzegorz Łabiak

Wykorzystanie hierarchicznego modelu współbieżnego automatu w projektowaniu sterowników cyfrowych

Oficyna Wydawnicza
Uniwersytetu Zielonogórskiego
2005

Grzegorz Łabiak
Instytut Informatyki i Elektroniki
Uniwersytet Zielonogórski
ul. Podgórna 50
65-246 Zielona Góra, Polska
e-mail: G.Labiak@iie.uz.zgora.pl

Recenzenci:

- Tadeusz ŁUBA, Politechnika Warszawska
- Bolesław POCHOPIEŃ, Politechnika Śląska

Tekst monografii został przygotowany na podstawie pracy doktorskiej autora

Monografia została wydana w ramach projektu badawczego
Komitetu Badań Nukowych nr 4 T11C 006 24

ISBN 83-89712-42-3

Skład i łamanie w systemie LaTeX2$_\varepsilon$: Grzegorz Łabiak
Druk i oprawa: Oficyna Wydawnicza Uniwersytetu Zielonogórskiego

Oficyna Wydawnicza Uniwersytetu Zielonogórskiego, Zielona Góra 2005
65-246 Zielona Góra; ul. Podgórna 50
tel./fax (068) 328 78 64; OficynaWydawnicza@adm.uz.zgora.pl

SPIS TREŚCI

SPIS WAŻNIEJSZYCH SYMBOLI I OZNACZEŃ

FSM	maszyna o skończonej liczbie stanów
S	zbiór stanów
s	stan lub wyjście przerzutnika
I	zbiór wejść lub sygnałów
i	wejście lub sygnał
O	zbiór wyjść
o	wyjście
f	funkcja stanu następnego
h	funkcja wyjścia
PN	sieć Petriego
P	zbiór miejsc
p	miejsce
T	zbiór tranzycji
t	tranzycja lub warunek realizacji tranzycji (*encond*)
M	funkcja znakowania
HPN	hierarchiczna sieć Petriego
$f_x, f_{\overline{x}}$	pozytywne i negatywne dopełnienie algebraiczne
ρ, ρ^*	relacja, domknięcie relacji
hrc, hrc^*	funkcje hierarchii
$root$	najwyższy stan nadrzędny
$parent$	bezpośredni stan nadrzędny
$endst$	stan końcowy
lca	najniższy wspólny przodek
$type$	funkcja typu stanu
$default$	funkcja stanu początkowego
$history$	funkcja historii
E	zbiór zdarzeń
e	zdarzenie
out	funkcja źródła tranzycji
in	funkcja celu tranzycji
$tlabel$	funkcja etykiety tranzycji
$trigger$	wzbudzenie tranzycji
$taction$	akcja tranzycji
$saction$	funkcja etykiety stanu
$entry$	akcja wejściowa
do	akcja statyczna
$exit$	akcja wyjściowa
Z	statechart
C, C_0	konfiguracja, konfiguracja początkowa
$path$	ścieżka
μC	mikrokonfiguracja

$downclos$	domknięcie ku dołowi
$intcond$	wewnętrzny warunek wywłaszczenia sterowania ze stanu
$\mu\Sigma$	mikrokrok
Σ	krok
G, G_0	stan globalny, globalny stan początkowy
X	zbiór zdarzeń wejściowych
Y	zbiór zdarzeń wyjściowych
$input$	odwzorowanie zdarzeń wejściowych
I_x	zbiór sygnałów wejściowych
$output$	odwzorowanie zdarzeń wyjściowych
I_y	zbiór sygnałów wyjściowych
${}^\bullet s$	zbiór bezpośrednich tranzycji wejściowych do stanu
$s^\bullet$	zbiór bezpośrednich tranzycji wyjściowych ze stanu
$activecond$	warunek aktywności stanu
$encond$	warunek realizacji tranzycji
$stinacttr$	zbiór tranzycji wywłaszczających sterowanie ze stanu
$sigactst$	zbiór stanów ustawiających sygnał
$sigenactst$	zbiór stanów ustawiających sygnał związany ze zdarzeniem wejściowym do stanu
$sigexactst$	zbiór stanów ustawiających sygnał związany ze zdarzeniem wyjściowym ze stanu
$sigacttr$	zbiór tranzycji ustawiających sygnał
δ	funkcja wzbudzenia
$activate$	składowa zapalająca
$inactivate$	czynnik podtrzymujący
$setup$	składnik ustawiający stan początkowy
λ	funkcja sygnału
x	sygnał wejściowy
X_A	funkcja charakterystyczna
$[G_0\rangle$	zbiór wszystkich stanów globalnych
$[C_0\rangle$	zbiór wszystkich konfiguracji
$X_{[G_0\rangle}$	funkcja charakterystyczna zbioru wszystkich stanów globalnych
$X_{[C_0\rangle}$	funkcja charakterystyczna zbioru wszystkich konfiguracji

Rozdział 1

WSTĘP

1.1. Motywacje podjęcia tematu

Postęp technologiczny ostatnich lat spowodował zasadnicze zmian w projektowaniu układów cyfrowych. Do końca lat osiemdziesiątych głównie stosowane były standardowe elementy wielkiej skali integracji w połączeniu z elementami małej i średniej skali integracji. Obecnie dominującą pozycje zdobyły układy *ASIC* (*ang.* Application Specific Integrated Circuits), czyli specjalizowane układy scalone (zarówno analogowe, jak i cyfrowe), projektowane z myślą o konkretnym zastosowaniu. Pojawienie się nowej technologii układów specjalizowanych również wywarło wielki wpływ na rozwój nowych metod projektowych i systemów *CAD*. Z kolei powstawanie nowych tańszych technologiii metod projektowych stale powiększa obszar zastosowań układów elektronicznych, sprawiając, że systemy cyfrowe coraz bardziej, a zwłaszcza pod postacią systemów osadzonych, przenikają otoczenie człowieka, stając się niezbędnymi przedmiotami codziennego użytku. Równocześnie zmianom ostatnich lat towarzyszy stały i równomierny wzrost gęstości upakowania tranzystorów w układzie scalonym, który zgodnie z obserwacją Gordona Moore'a (jednego ze współzałożycieli firmy *Intel*), poczynioną w roku 1973, sprawia, że co 18 miesięcy gęstość upakowania się podwaja (Buttazzo, 2001; Gajski i in., 2001).

Aktualnie ma miejsce taka oto sytuacja, że istnieje silne zapotrzebowanie na układy cyfrowe wszelakich zastosowań, dostawcy technologii oferują projektantom układy programowalne zawierające nawet 10 milionów bramek, lecz projektanci niestety nie zawsze są wyposażeni w odpowiednie techniki projektowe zdolne do zaspokojenia potrzeb i wykorzystania dostępnych zasobów sprzętowych. Wiele tradycyjnie stosowanych metod projektowych osiągnęło kres swoich możliwości i nie przystaje do dzisiejszych wyzwań. Coraz większego znaczenia nabiera modelowanie poziomu systemu, modelowanie najbardziej abstrakcyjne, które będąc pierwszym etapem projektowania, poprzez swe ograniczenia, rzutuje na wszystkie pozostałe etapy i na ostateczną funkcjonalność projektowanego urządzenia. Opracowanie dobrych metod modelowania poziomu systemu sprawi, że projektowane urządzenia będą charakteryzować się większymi i bardziej rozbudowanymi możliwościami zastosowań. Pewną ceną abstrakcyjnego modelowania może być stosunkowo duże zużycie zasobów podstawowych elementów elektronicznych, lecz wydaje się, że w sytuacji istnienia stabilnego wzrostu upakowania tranzystorów (ocenianego przez firmy produkujące układy scalone jeszcze na około 15 do 20

lat (Buttazzo, 2001)) jest zasadnym opracowywanie metod zwiększających możliwości modelowania funkcjonalności, kosztem powiększonej konsumpcji elementów podstawowych. Tym bardziej jest to zasadne, gdyż opisywanemu wzrostowi towarzyszy zmniejszenie kosztów elementów jednostkowych, a opracowywane nowe techniki wydaje się, że zawsze mogą być doskonalone.

Autor w swej pracy skupił się na modelowaniu zachowania diagramami stanów, zwanych również diagramami statechart (Harel, 1987), które poprzez silne odwołanie do wizualizacji i jawne wykorzystywanie hierarchii behawioralnej, uważane są obecnie za jeden z najlepszych sposobów opisu złożonego zachowania (de Micheli, 1998). Zachowanie wyrażone za pomocą diagramów może zostać wykorzystane między innymi do projektowania układów cyfrowych następującego rodzaju: systemów reaktywnych w ogólności i układów sterowania w szczególności. Innym zastosowaniem może być specyfikacja sterowania w układach cyfrowych, realizowanych według tradycyjnego modelu typu jednostka sterująca ze ścieżką przetwarzania danych. W wielu przypadkach, zwłaszcza dotyczących złożonego zachowania, tradycyjne metody projektowania sterowania, takie jak automat o skończonej liczbie stanów czy sieć Petriego, okazują się niewystarczające. Brak wsparcia dla posługiwania się pojęciami ogólnymi w modelowaniu sprawia, że projektant zmuszony jest operować wyjątkowo dużą liczbą szczegółów. W tym względzie, diagramy statechart oferują środki hierarchicznej specyfikacji.

Dotychczas powstało bardzo niewiele systemów *CAD*, które umożliwiałyby zastosowanie diagramów do projektowania układów cyfrowych i jednoczesną ich implementację w strukturach programowalnych. Jedynym znanym autorowi systemem tego rodzaju jest program *Statemate MAGNUM* firmy *I-Logix* (STA, 2004), który produkuje równoważny opis behawioralny w językach *HDL* (I-L, 2000*b*). Inne tradycyjne przemysłowe systemy *CAD* projektowania układów cyfrowych oferują jedynie wsparcie dla hierarchii strukturalnej, co w modelowaniu nie znajduje oczekiwanego zastosowania.

Rosnące zapotrzebowanie na coraz bardziej złożone układy cyfrowe, obniżka kosztów elementów jednostkowych oraz pewne niedostatki istniejących narzędzi *CAD*, zrodziły potrzebę podjęcia tematu opracowania lepszej metodyki projektowania sterowników cyfrowych dla celów implementacji w strukturach programowalnych.

1.2. Teza i cele pracy

Kierując się motywacjami przedstawionymi w poprzednim podrozdziale, autor podjął się przeprowadzenia badań, których teza została sformułowana następująco:

> **Cyfrowe systemy sterujące mogą być efektywnie modelowane z wykorzystaniem hierarchicznego modelu automatu współbieżnego oraz bezpośrednio implementowane w strukturach programowalnych.**

Przez efektywne modelowanie autor rozumie opracowanie i realizację systemu *CAD*, działającego na wzór już istniejących innych systemów, który by oferował

możliwość translacji zachowania specyfikowanego diagramami statechart, będącymi hierarchicznym modelem automatu współbieżnego, na postać możliwą do analizy i implementacji w strukturach programowalnych, np. *FPGA*. Dodatkowo założono, że opracowany system ma być zgodny ze zdobywającą coraz to nowe obszary zastosowań technologią *UML*. Z tak przyjętej tezy głównej wynikają następujące cele o charakterze teoretycznym:

- zbadanie przydatności technologii *UML* w projektowaniu układów cyfrowych,
- opracowanie składni i semantyki diagramów dla potrzeb specyfikacji zachowania cyfrowych układów sterowania,
- opracowanie modelu matematycznego, który w sposób formalny definiuje składnię i zachowanie diagramów,
- opracowanie założeń dynamiki oraz zasad realizacji sprzętowej – opis równaniami logicznymi,
- opracowanie algorytmu generowanie grafu osiągalności dla modelowanego zachowania.

Realizacja celów teoretycznych stanowi podstawę pod budowę autorskiego systemu *CAD*, zwanego dalej systemem *HiCoS* (jest to skrótowiec utworzony od angielskich słów ***Hi****erarchical* ***Co****ncurrent* ***S****ystem*), implementującego opracowane metody. W ramach pracy nad systemem przyjęto następujące cele praktyczne:

- opracowanie gramatyki języka *SSF* stanowiącego format danych wejściowych,
- opracowanie wewnętrznej struktury danych i realizacja programu kompilatora dokonującego przejścia na postać w języku *VHDL* na poziomie *RTL*,
- programowa implementacja algorytmu generowania grafu osiągalności.

Ponadto założono, że prawdziwość opracowanych zasad realizacji sprzętowej zostanie potwierdzona na drodze eksperymentalnej na dwa sposoby:

- poprzez porównanie specyfikowanego zachowania z wynikami symulacji dla modeli w języku *VHDL*, wygenerowanymi z systemu,
- poprzez porównanie, dla niedużych przykładów, grafów osiągalności stworzonych ręcznie, z grafami wygenerowanymi przez system.

Można przyjąć, że zgodność otrzymanych wyników otrzymanych z systemu z wynikami oczekiwanymi, z dużym prawdopodobieństwem oznacza poprawność opracowanych i zaimplementowanych koncepcji. Ostatnim celem praktycznym jest przeprowadzanie implementacji przykładowych modeli w strukturach programowalnych – jako docelowe wybrano popularne i dostępne autorowi układy programowalne *FPGA* firmy *Xilinx*.

1.3. Struktura pracy

Praca została podzielona na dziewięć rozdziałów i dodatki. Rozdział pierwszy wprowadza w tematykę badań. Rozdziały drugi, trzeci, czwarty i piąty mają charakter teoretyczno-przeglądowy. Rozdziały szósty, siódmy i ósmy przedstawiają uzyskane wyniki. Rozdział dziewiąty podsumowuje badania, a w dodatkach zamieszczono pewne informacje szczegółowe i przykłady testowe.

Rozdział pierwszy charakteryzuje przebieg prowadzonych prac. Przedstawia motywacje podjęcia tematu, opisuje metodykę prowadzenia badań, definiuje tezę główną oraz omawia wynikające z niej cząstkowe cele teoretyczne i praktyczne.

W rozdziale drugim scharakteryzowano cyfrowy układ sterowania binarnego jako system reaktywny. Podano przykładowe zastosowanie takiego układu. Przedstawiono tradycyjne metody projektowania. Ponadto omówiono te techniki i technologie, które wiążą się z pracami prowadzonymi przez autora. Do omawianych zagadnień należą: języki opisu sprzętu, binarne diagramy decyzyjne, funkcja charakterystyczna, układy programowalne, a w szczególności układy *FPGA*.

Rozdział trzeci w sposób nieformalny charakteryzuje diagramy statechart. Jako pierwsza została opisana technologia *UML* (najpopularniejsza technologia wykorzystująca diagramy) oraz jej zastosowanie w projektowaniu układów sterowania. Następnie omówiono język *SpecChart* – jako kolejną technikę projektową, stosującą diagramy. Zdefiniowano postać graficzną zgodną z normą *UML* oraz określono związki diagramów z sieciami Petriego.

W rozdziale czwartym dokładnie scharakteryzowano wybrane kwestie natury semantycznej oraz dokonano porównania propozycji autora z dokonaniami innych naukowców.

Rozdział piąty opisuje wybrane komercyjne systemy *CAD* wykorzystujące diagramy. Najwięcej miejsca poświęcono dominującemu pakietowy *Statemate MAGNUM* firmy *I-Logix*.

Dla inżyniera projektanta układów cyfrowych najistotniejszą cechą diagramów jest ich postać graficzna, lecz w pracach nad zastosowaniem diagramów statechart niezbędne jest posługiwanie się precyzyjnie zdefiniowanym aparatem matematycznym. Stąd w rozdziale szóstym został zaprezentowany własny matematyczny model wraz z pojęciami pomocniczymi, niezbędnymi do formalnego zdefiniowania diagramów i opisu ich zachowania.

Rozdział siódmy stanowi najistotniejszą część pracy i jest opisem autorskiej koncepcji układowej realizacji modelu zachowania specyfikowanego diagramami statechart. Główną cechą charakterystyczną proponowanego podejścia jest przedstawienie docelowego układu na poziomie rejestrów i przesłań pomiędzy nimi.

Rozdział ósmy, obok rozdziału siódmego, również zawiera prezentacje autorskich metod oraz przedstawia osiągnięte rezultaty praktyczne. Głównym praktycznym osiągnięciem jest system *HiCoS*. Stąd kolejno są omówione budowa i działanie systemu, wejściowy język opisu (język *SSF*) wraz z jego wizualizacją, postać danych wyjściowych z systemu. Następnie omawiany jest algorytm generowania grafu osiągalność, polegający na redukcji hierarchicznego modelu automatu współbieżnego do automatu sekwencyjnego. Jako ostatnie zostały przedstawione i ocenione

wyniki implementacji wybranych modeli testowych.

W rozdziale dziewiątym ustosunkowano się do tezy badań, wymieniono elementy nowatorskie i autorskie oraz wskazano pewne niedostatki opracowanej metodyki i określono związane z tym kierunki dalszych poszukiwań.

W dodatkach zawarto formalny opis gramatyki języka *SSF* oraz podano jego możliwe dalsze rozszerzenia. Przedstawiono tam również przykłady sterowników o charakterze praktycznym, podając nie tylko ich opis słowny wraz z diagramem stanów, ale również zamieszczono specyfikację w formacie *SSF*.

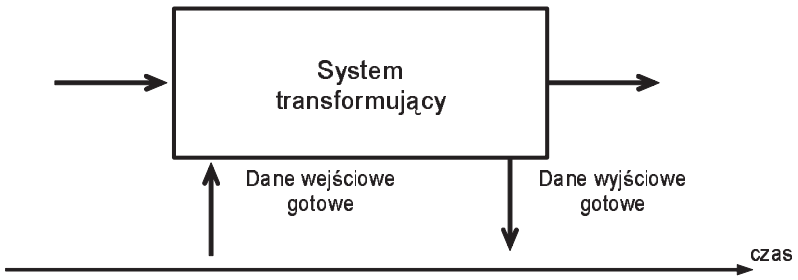
Rozdział 2

WYBRANE METODY, TECHNIKI, I TECHNOLOGIE REALIZACJI STEROWNIKÓW CYFROWYCH

2.1. Wprowadzenie

Projektowane systemy, ze względu na zachowanie, można podzielić na systemy transformujące oraz na systemy reaktywne (Drusinsky, 1997; Kyeyune, 2000).

Systemy transformujące (rys. 2.1) są to systemy sterowane danymi, co oznacza, że zachowanie takiego systemu zależy od przepływu danych pomiędzy jego składowymi. Systemy te charakteryzują się tym, że każdy z jego elementów składowych (a w szczególności cały system) czeka na pojawienie się gotowych danych na jego wejściu, co jest odpowiednio sygnalizowane, po czym następuje przetwarzanie tych danych i dane te są przekazywane na wyjście, co również jest sygnalizowane. Po takim przetwarzaniu system transformujący (lub jego składowe) przechodzą w stan oczekiwania na pojawienie się kolejnego kompletu danych na wejściu. Takie systemy można opisywać diagramami przepływu danych.

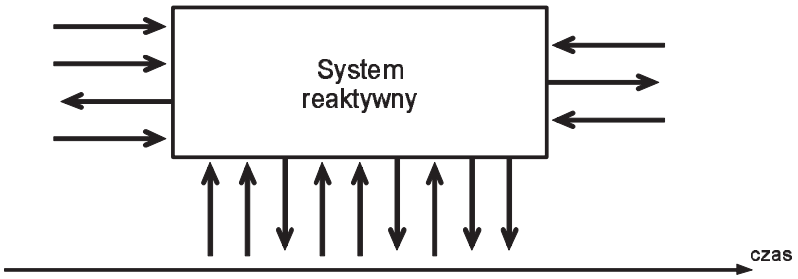


Rys. 2.1. Model systemu transformującego (Kyeyune, 2000)

Systemy typu transformującego można modelować metodą od ogółu do szczegółu (*ang.* top-down). Cechą typową tych systemów jest to, że złożone zależności między danymi wejściowymi a danymi wyjściowymi, mogą zostać funkcjonalnie dekomponowane na prostsze zależności składowe.

Systemy reaktywne (rys. 2.2) nie funkcjonują w tak przewidywalnym scenariuszu czasowym jak systemy transformujące. Ich główną cechą charakterystyczną

jest to, że dane wejściowe takiego systemu mogą pojawiać się zupełnie dowolnie, (co na rysunku oznaczone jest poprzez strzałki umieszczone przy systemie), tzn. że nie przyjmuje się żadnych założeń, co do tego, na których wejściach i kiedy pojawią się dane, a w przypadku systemów czasu rzeczywistego oczekuje się, że odpowiedź układu będzie natychmiastowa.



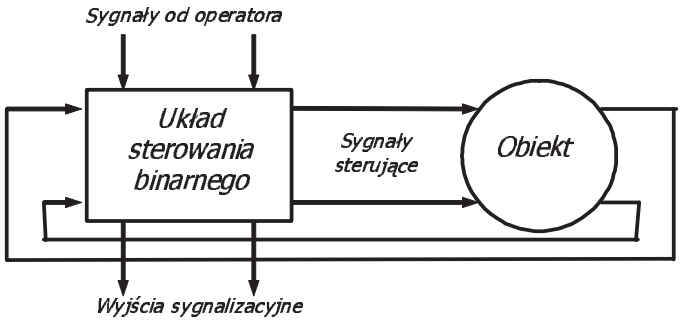
Rys. 2.2. Model systemu reaktywnego (Kyeyune, 2000)

Ponadto, o systemach reaktywnych można powiedzieć, że (Harel i Politi, 1998):

- są sterowane zdarzeniami,
- prowadzą stałą interakcję ze swoim otoczeniem, używając do tego sygnałów wejściowych i wyjściowych, które mogą być zarówno ciągłe, jak i dyskretne, systemy te powinny odpowiadać na przerwania, tzn. reagować na zdarzenia o odpowiednio wysokim priorytecie, nawet jeżeli są zajęte innymi obliczeniami,
- ich działanie i reakcje na sygnały wejściowe często muszą odpowiadać surowym wymaganiom czasowym,
- zmieniają swój stan w zależności od bieżącego trybu działania i wartości danych oraz od przeszłego zachowania,
- nie muszą być współbieżne, aczkolwiek najczęściej ich zachowanie jest przyrównane do komunikujących się procesów działających współbieżnie.

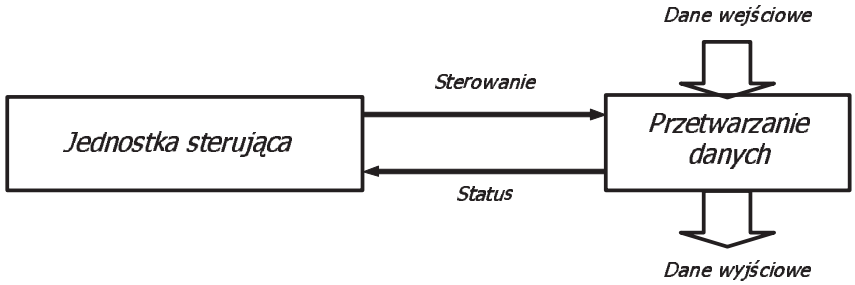
Przykładami systemów reaktywnych są system rezerwacji miejsc w samolocie, systemy osadzone takie jak układy awioniki czy zaawansowane elektronicznie artykuły gospodarstwa domowego. Do systemów reaktywnych również należą układy sterujące, a w szczególności cyfrowe układy sterowania binarnego (rys. 2.3), czyli takie, które na swoim wejściu i wyjściu operują wartościami binarnymi.

Innym możliwym zastosowaniem reaktywnego systemu sterującego jest jednostka sterująca w systemie przetwarzającym dane (*ang.* Finite-State Machine with Datapath). Rysunek 2.4 przedstawia model takiej architektury (Gajski i in., 1994), gdzie wejściami do jednostki sterującej są sygnały statusu przetwarzania



Rys. 2.3. Układ sterowania binarnego (Misiurewicz, 1987)

danych, a wyjściami sygnały kierujące obliczeniami i przygotowujące dane wyjściowe. Taki model układu znajduje zastosowanie w cyfrowym przetwarzaniu sygnałów (*ang.* Digital Signal Processing) oraz stanowi podstawową architekturę mikroprocesorów ogólnego przeznaczenia.



Rys. 2.4. Jednostka sterująca ze ścieżką przetwarzania danych (Misiurewicz, 1987)

Do opisów systemów reaktywnych mogą stosować się diagramy przepływu danych oraz dekompozycja funkcjonalna, lecz te metody, skuteczne w przypadku systemów transformujących, są niewystarczające do opisu ich pełnego działania. Do kompletnego przedstawienia zachowania można by użyć automatu skończonego. Wówczas utworzenie modelu, polegałoby na wyodrębnieniu stanów lub trybów działania, określeniu zdarzeń i warunków, które powodują przejścia między stanami oraz na wyspecyfikowaniu akcji wykonywanych w poszczególnych stanach i przy rozpatrywanych przejściach. Taki model jednak posiada pewne wady. Brak hierarchii powoduje, że projektant nie może specyfikować projektu takiego systemu na odpowiednim dla swoich potrzeb poziomie szczegółowości, a jako że jest to model „płaski" zmuszony jest on operować najdrobniejszymi szczegółami specyfikacji. Z kolei brak wsparcia dla współbieżności często jest przyczyną wykładniczej eksplozji stanów globalnych, co powoduje, że taki model przestaje być zupełnie zrozumiały. Zatem, właściwym modelem do opisu zachowania systemów reaktywnych powinien być model wspierający trzy wymienione cechy, czyli: operować pojęciem

stanu, wspierać hierarchię, opisywać współbieżność. Takim modelem są diagramy statechart zaproponowane przez Dawida Harela (Harel, 1987), czyli hierarchiczny model automatu współbieżnego (Gajski i in., 1994).

W polskiej literaturze przedmiotu spotkać można inne nazwy diagramów, jak np.: mapy stanów (Harel, 2001), wykresy stanów (Subieta, 1999*b*), diagramy Harela (Subieta, 1999*b*) czy diagramy stanów (Subieta, 1999*b*). Wszystkie te propozycje zawierają odwołanie do najistotniejszej cechy diagramów, mianowicie do ich postaci graficznej. Propozycją najczęściej spotykaną są diagramy stanów (Subieta, 1999*b*), lecz jak to przedstawia w swoich innych materiałach (np. Subieta, 1999*c*) autor słownika terminów z zakresu obiektowości (Subieta, 1999*b*), w którym ta propozycja jest zamieszczona, nazwa ta nie oddaje istoty rzeczy, ponieważ powoduje fałszywe asocjacje z innymi znaczeniami. Ponadto diagramy statechart to nie zawsze postać graficzna, gdyż można je postrzegać jako abstrakcyjny model matematyczny, tak jak to na przykład jest przedstawione w rozdziale 6. Wówczas określenia mapa, wykres czy diagram tracą sens i z tego powodu autor proponuje nie tłumaczyć angielskiej nazwy własnej „statechart", a zamiast tego stosować polską nazwę własną „statechart", będącą kalką językową (albo może „stanogram" lub „stagram"?).

2.2. Maszyny o skończonej liczbie stanów

Maszyna o skończonej liczbie stanów (*ang.* Finite-State Machine lub w skrócie *FSM*), zwana również maszyną stanową lub automatem cyfrowym (Traczyk, 1986; Majewski i in., 1992; Adamski, 1990; Gajski i in., 1994; Majewski, 1998; Kalisz, 2002; Kamionka-Mikuła i in., 2004), jest najbardziej popularnym modelem opisującym zachowanie systemów sterowania, w którym chwilowe działanie systemu jest w sposób naturalny reprezentowane w formie stanów i przejść między nimi. Zasadniczo *FSM* składa się ze zbioru stanów (Gajski i in., 1994), zbioru przejść między stanami oraz zbioru akcji przypisanych stanom lub przejściom (def. 2.1). Ze względu na powiązanie akcji wyjściowych z przejściami lub stanami wyróżnia się dwa rodzaje automatów: Mealy'ego i Moore'a.

Definicja 2.1. Maszyną o skończonej liczbie stanów *nazywana jest szóstka* $\langle S, I, O, f, h, s_0 \rangle$*, w której:*
$S = \{s_0, s_1, \ldots, s_j\}$ *jest skończonym zbiorem stanów,*
$I = \{i_0, i_1, \ldots, i_k\}$ *jest skończonym zbiorem wejść,*
$O = \{o_0, o_1, \ldots, o_l\}$ *jest skończonym zbiorem wyjść,*
$f : S \times I \to S$ *jest funkcją stanu następnego,*
$h : S \times I \to O$ *jest funkcją wyjścia,*
$s_0 \in S$ *jest pewnym wyróżnionym stanem początkowym, od którego automat zaczyna działanie.*

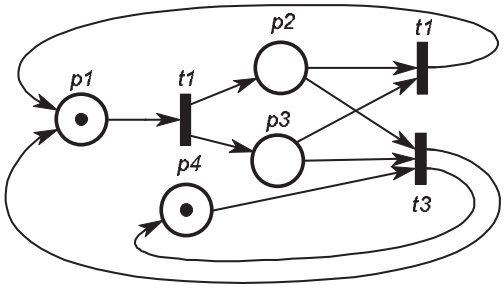
Realizacja sprzętowa tak zdefiniowanej maszyny *FSM*, w najprostszej postaci, polega na skojarzeniu z każdym stanem jednego przerzutnika, np. typu D. W takiej realizacji w danej chwili aktywnym jest tylko jeden przerzutnik, co jest równoważne aktywności skojarzonego z nim stanu. Ze względu na tę właściwość metoda taka

jest nazywana *one-hot*. Istnieją również inne metody projektowania i syntezy *FSM* (m.in. opisane w Traczyk, 1986; Majewski, 1998; Kalisz, 2002; Kamionka-Mikuła i in., 2004). Niektóre z nich opierają się na bardziej wymyślnym kodowaniu stanów (Traczyk, 1986), gdzie w zależności od potrzeb, kryterium może być liczba przerzutników wykorzystanych do kodowania stanu lub złożoność logiki obliczającej kod stanu następnego, czy też szybkość działania takiego automatu.

Dla inżyniera projektanta modelowanie zachowania realizowanego jako *FSM* odbywa się głównie poprzez narysowanie równoważnego mu grafu stanu lub, co już jest mniej czytelne, poprzez tabelaryczne zdefiniowanie funkcji przejść. Wadą *FSM* są trudności modelowania zachowania składającego się z dużej liczby stanów, a tą cechą charakteryzuje się wiele systemów sterujących. Rozwiązaniem może być inne spojrzenie na zagadnienie sterowania polegające na opisie zachowania w sposób współbieżny lub też operowanie pojęciami uogólnionymi (abstrakcyjnymi), co już nie jest wspierane przez metodykę *FSM*.

2.3. Sieci Petriego

Sieć Petriego (w skrócie oznaczana *PN*), postrzegana jako model matematyczny (Peterson, 1981; Reisig, 1988; Murata, 1989; Adamski, 1990; 1998; Banaszak i in., 1993; Jensen, 1997; Girault i Valk, 2003), jest dwudzielnym skierowanym grafem, w którym występują dwa rodzaje węzłów, zwane miejscami i tranzycjami. W graficznej postaci (rys. 2.5) miejsca reprezentowane są przez kółka a tranzycje przez prostokąty połączone łukami skierowanymi. Każdemu miejscu w sieci może być przyporządkowany co najwyżej jeden żeton (marker, znacznik).



Rys. 2.5. Przykład sieci Petriego

Definicja 2.2. Siecią Petriego *nazywana jest następująca czwórka* $\langle P, T, F, M_0 \rangle$*, w której:*
$P = \{p_0, p_1, \ldots, p_k\}$ *jest skończonym niepustym zbiorem miejsc,*
$T = \{t_0, t_1, \ldots, t_l\}$ *jest skończonym niepustym zbiorem tranzycji,*
$F \subseteq (P \times T) \cup (T \times P)$*jest skończonym niepustym zbiorem łuków,*
$M_0 : P \rightarrow \{1, 0\}$ *jest funkcją znakowania początkowego, przypisującą żetony wyróżnionym miejscom startowym, oraz dodatkowo przyjmuje się, że zbiory miejsc i tranzycji są rozłączne* $P \cap T = \emptyset$*.*

Działanie sieci Petriego określone jest następującymi regułami:

- tranzycja jest gotowa do realizacji, jeżeli wszystkie jej miejsca wejściowe posiadają żeton,
- realizacja tranzycji pociąga za sobą usunięcie żetonów (markerów) ze wszystkich jej miejsc wejściowych i umieszczenie żetonów we wszystkich jej miejscach wyjściowych.

W oparciu na tak przyjętym modelu matematycznym sieci Petriego można zrealizować współbieżny automaty cyfrowy (*multiautomat cyfrowy*), realizujący funkcje systemu reaktywnego (Kozłowski, 1993; Kozłowski i in., 1995; Biliński, 1996; Adamski, 1998; Wolański, 1998*a*; 1998*b*; Puczyńska i in., 2000; Węgrzyn i Węgrzyn, 2000; Andrzejewski, 2001; Cortadella i in., 2002; Karatkevich i Andrzejewski, 2002). W tym celu należy wprowadzić pojęcie interpretowanej sieci Petriego, tzn. takiej sieci, która reaguje na sygnały wejściowe, generując sygnały do otoczenia. W interpretowanej sieci Petriego generowanie sygnałów wyjściowych można związać z miejscem (sieć typu Moore'a) lub z tranzycją (sieć typu Mealy'ego). Sygnały wejściowe mogą występować w wyrażeniach logicznych nałożonych na tranzycje, zwanych predykatami lub etykietami, których spełnienie jest koniecznym warunkiem ich realizacji. Dodatkowo, w celu rozszerzenia możliwości opisu, stosuje się łuki zabraniające i zezwalające (Adamski, 1990; Kozłowski, 1993; Kozłowski i in., 1995; Biliński, 1996; Wolański, 1998*a*; Andrzejewski, 2002*b*).

Interpretowane sieci Petriego doczekały się wielu metod syntezy. Oto najważniejsze z nich:

- bezpośrednia implementacja sieci typu *one-hot* – wywodzi się ze stosowanej dla automatów sekwencyjnych metody *one-hot*, a polega na przyporządkowaniu każdemu miejscu przerzutnika typu D i następnie na podstawie topografii sieci i reguł odpaleń na określeniu jego funkcji wzbudzeń (Adamski, 1991; Kozłowski, 1993; Kozłowski i in., 1995; Biliński, 1996),
- bezpośrednia implementacja sieci z zakodowanymi miejscami – podobna do metody kodowania stanów automatów sekwencyjnych; w metodzie tej dąży się do tego, aby jak najwięcej miejsc z sieci współdzieliło jeden przerzutnik (Kozłowski, 1993; Kozłowski i in., 1995; Biliński, 1996),
- metoda przekształcenia sieci na równoważny automat sekwencyjny – polega na przekształceniu sieci Petriego do pojedynczego automatu sekwencyjnego; realizowane jest to poprzez wykorzystanie grafu osiągalności sieci (Adamski, 1991; Biliński, 1996) i potraktowanie jego wierzchołków jako stanów wewnętrznych klasycznego cyfrowego automatu sekwencyjnego,
- dekompozycja sieci i implementacja jej składowych automatowych – sprowadza się do rozkładu sieci na składowe automaty sekwencyjne, które tworzone są poprzez usunięcie w sieci wszystkich łuków dochodzących do łączących i odchodzących od rozwidlających tranzycji (Kozłowski, 1993; Kozłowski i in., 1995; Biliński, 1996; Adamski, 1998),

- synteza wysokiego poziomu – polega na opisaniu sieci w sposób behawioralny lub strukturalny w syntezowalnym podzbiorze wybranego języka *HDL*, np. *C*, *VHDL* czy *Verilog* (Wolański, 1998*a*; 1998*b*; Puczyńska i in., 2000; Skowroński, 2000; Andrzejewski, 2001).

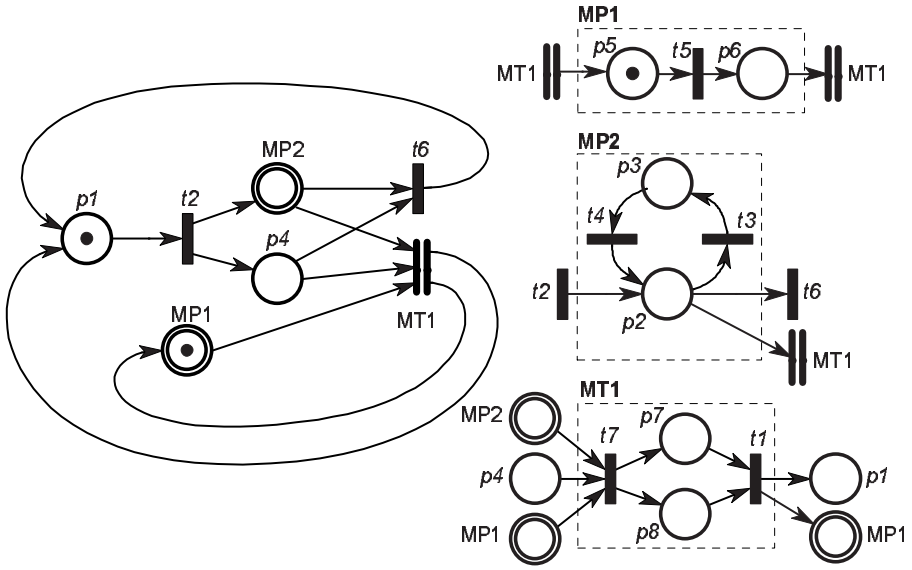
Główną zaletą stosowania sieci Petriego, w porównaniu z tradycyjnym modelem automatu sekwencyjnego, jest zapobieżenie wykładniczemu wzrostowi liczby stanów modelowanego zachowania. Wynika to z możliwości opisu uwzględniającego współbieżność.

2.4. Hierarchiczne sieci Petriego

Hierarchiczną siecią Petriego (w skrócie oznaczaną *HPN*), czasami zwaną makrosiecią (Fernandes i in., 1997; Węgrzyn, 1998*a*; 1998*b*), jest sieć, w której występują makroelementy typu makromiejsca i makrotranzycje. Rysunek 2.6 przedstawia przykładową sieć hierarchiczną (zaczerpniętą z pracy (Węgrzyn, 1998*a*)), gdzie makromiejsca zaznaczone są podwójnymi kółkami, makrotranzycje podwójnymi belkami, a obok na rysunku znajdują się ich rozwinięcia. Sieć hierarchiczna jest siecią powstałą z tradycyjnej „płaskiej" sieci Petriego, w której z wybranych jej fragmentów wyodrębniono podsieci. Takie podsieci, w zależności od elementów, poprzez które są dołączane do sieci, zaznaczane są jako makromiejsca (elementami łączącymi są miejsca) lub makrotranzycje (elementami łączącymi są tranzycje). W rozwinięciach makroelementów można wyodrębniać kolejne podsieci, tworząc z nich makroelementy niższych poziomów hierarchii. Uzyskany w ten sposób model charakteryzuje się hierarchią strukturalną.

Modelowanie zachowania przy użyciu hierarchicznych sieci Petriego zasadniczo polega na operowaniu pojęciami najniższego poziomu szczegółowości (podejście od szczegółu do ogółu, *ang.* bottom-up). Dopiero mając zachowanie opisane tradycyjną siecią Petriego w sposób naturalny można w niej wyodrębnić makroelementy. Główna zaleta tak otrzymanego modelu ujawnia się, gdy jest on wykorzystywany do dokumentowania zachowania. Przy postępowaniu odwrotnym – od ogółu do szczegółu (*ang.* top-down) – projektant, co prawda może operować abstrakcyjnymi pojęciami makroelementów (abstrakcyjnymi w sensie oderwania od szczegółów najniższego poziomu), jednakże w przypadku łuków dochodzących do makroelementów zawsze trzeba pamiętać o końcowym (lub początkowym) miejscu lub tranzycji, będących elementami najniższego poziomu szczegółowości, co jest wbrew zasadzie abstrahowania.

Przedstawiony model hierarchicznej sieci Petriego nie jest jedynym rozszerzeniem tradycyjnej *PN* uwzględniającym hierarchię. Inne propozycje to: Hierarchiczne Kolorowalne *PN* (*HCPN*) (Jensen, 1997; Węgrzyn, 1998*a*; 1998*b*), Zorientowane Obiektowo *PN* (*OOPN*) (Esser, 1996; Vojnar, 1999), Hierarchiczne Zorientowane Obiektowo *PN* (*HOOPN*) (Hong i Bae, 1998). Na szczególną uwagę zasługuje propozycja o nazwie *PetriCharts* (Holvoet i Verbaeten, 1995), która charakteryzuje się elementami hierarchii behawioralnej oraz silnym podobieństwem do diagramów statechart. Bardzo podobnym modelem, docelowo wykorzystywanym w syntezie



Rys. 2.6. Przykład hierarchicznej sieci Petriego (Węgrzyn, 1998*a*)

programowej, jest model przedstawiony w pracach (Andrzejewski, 2001; 2002*a*). Innym przykładem wykorzystania hierarchii behawioralnej w połączeniu z sieciami Petriego jest graficzny język *Grafchart* (Årzén, 1996).

2.5. Języki opisu sprzętu

Głównym zadaniem języków opisu sprzętu (*ang.* Hardware Description Languages, *HDL*) jest umożliwienie wymiany informacji między projektantami a systemami *CAD*. Cechą języków *HDL*, w przeciwieństwie do opisu słownego, są precyzyjnie zdefiniowane semantyka i składnia, dzięki czemu możliwe jest przekazywanie informacji o projektowanym układzie w sposób, który pozwala na jednoznaczną jej interpretację. Ze względu na swoją zwięzłość, języki te wykazują większą przydatność w procesie modelowania aniżeli tradycyjne metody, takie jak sieci działań czy diagramy stanów, aczkolwiek w literaturze (de Micheli, 1998) zawarto pogląd, że diagramy statechart zapewniają lepszą wizualizację zachowania, niemożliwą do uzyskania przy stosowaniu tradycyjnych języków *HDL*.

Języki *HDL* opisywane są za pomocą składni, semantyki i pragmatyki (Aho i in., 1990; Hopcroft i Ullman, 2003). Składnia w sposób formalny decyduje o strukturze zdań przynależnych do języka. Semantyka ustala i interpretuje znaczenia treści przekazywanych przez zdania należące do języka, zaś pragmatyka określa kiedy, jak i do czego należy stosować przyjęte konstrukcje językowe, tak aby uzyskać najbardziej pożądany efekt.

Języki *HDL* można podzielić na kilka sposobów. Jednym z podziałów jest podział na języki deklaracyjne i proceduralne. W językach deklaracyjnych rozwiązy-

wany problem opisuje się poprzez stosowanie deklaracji, nie podając szczegółowej metody rozwiązywanego problemu, przy czym istotną cechą charakterystyczną jest to, że kolejność poszczególnych deklaracji nie ma znaczenia. Przykładem takiego języka może być autorski język *SSF* (Łabiak, 2000*a*), gdzie główną konstrukcją opisu zachowania jest deklaracja tranzycji (opis języka znajduje się w podrozdziale 8.2 i w dodatku A). W przypadku języków proceduralnych model rozwiązywanego problemu zapisywany jest w postaci sekwencji kroków, których kolejność ma znaczenie zasadnicze. Przykładem może być instrukcja *process* z języka *VHDL*. Inny spotykany podział języków *HDL* to podział na języki strukturalne i języki behawioralne. Opis strukturalny sprowadza się do wyspecyfikowania połączeń między komponentami (elementami) układu, a modele tworzone w ten sposób są odpowiednikami schematów układów. W opisie behawioralnym istotnym jest operowanie zachowaniem, np. zmiana stanu modelowanego układu uwzględniona w opisie ma charakter behawioralny. Język *SSF*, operujący przejściami, jest przykładem języka behawioralnego.

Oprócz specyfikowania zadań, istotną cechą języków jest ich zastosowanie do dokumentowania i symulacji, co w procesie projektowania sprzętu ma niebagatelne znaczenie. Kolejnym zastosowaniem języków jest ich wykorzystanie do syntezy. W przypadku obecnie najpopularniejszych języków *HDL*: *VHDL* i *Verilog*, ze względu na to, że nie wszystkie konstrukcje dają się łatwo zrealizować układowo, wyodrębnione zostały ich podzbiory, które określone są mianem syntezowalnych. Wykorzystanie tych języków do syntezy okazało się na tyle efektywne, iż wielu producentów oprogramowania *CAD* stosuje je jako format wejściowy dla narzędzi syntezy. Natomiast w przypadku języka *SSF*, w rozdziale 7, przedstawiono sposób syntezy modeli opisanych w tym języku, co stanowi podstawę systemu *HiCoS* (podrozdział 7.2 i rozdział 8).

2.6. Techniki symboliczne

Techniki symboliczne, tzn. takie techniki, gdzie zbiory stanów są symbolicznie reprezentowane przez funkcje charakterystyczne, znalazły szerokie zastosowanie w syntezie, testowaniu oraz weryfikacji systemów o skończonej liczbie stanów. Coudert, Berthet oraz Madre (Coudert i in., 1989) byli pierwszymi, którzy zastosowali binarne diagramy decyzyjne do symbolicznego reprezentowania zbioru stanów. Wynikiem prac tych badaczy było sformułowanie algorytmu trawersu przestrzeni stanów na zasadzie przeszukiwania wszerz, gdzie w kolejnych iteracjach algorytm wyznacza ze zbioru stanów aktualnych zbiór wszystkich możliwych stanów następnych. W zaproponowanej metodzie zbiory stanów są reprezentowane przez funkcje charakterystyczne. Kluczową operacją algorytmu jest obliczanie obrazu zbioru stanów bieżących, reprezentowanych przez funkcję charakterystyczną, wyznaczonego przez funkcję stanów następnych (Biliński, 1996; Rasiowa, 1998). W przypadku *FSM* i sieci Petriego złożoność obliczeniowa technik symbolicznych, realizowanych z użyciem diagramów *BDD*, nie zależy w sposób istotny od ilości modelowanych stanów i tranzycji, lecz znacząco zależy od sposobu implementacji drzew *BDD* w pamięci komputera oraz od kolejności zmiennych w diagramie.

2.6.1. Binarne diagramy decyzyjne

Binarne Diagramy Decyzyjne (*ang.* Binary Decision Diagrams, *BDD*) (Ghosh i in., 1992; Biliński, 1996; Minato, 1996; de Micheli, 1998; Somenzi, 2004) są acyklicznym zorientowanym grafem z jednym wyróżnionym węzłem zwanym korzeniem, do którego nie dochodzą żadne krawędzie oraz z dwoma węzłami końcowymi zawierającymi wartości zero i jeden, odpowiadającymi boolowskim wartościom **0** i **1**. Każdy węzeł nieterminalny posiada numer, który służy do skojarzenia węzła ze zmienną logiczną z wyrażenia logicznego, opisującego funkcję reprezentowaną przez drzewo oraz od każdego takiego węzła odchodzą dwie krawędzie zaetykietowane zerem i jedynką.

Uporządkowanym *BDD* (*ang.* Ordered *BDD*) jest diagram, w którym zmienne boolowskie we wszystkich ścieżkach poprowadzonych od korzenia do węzła końcowego, występują w tym samym porządku i żadna zmienna w ścieżce nie pojawia się więcej niż jeden raz.

Każdą funkcję logiczną można rozwinąć w następujący szereg, zwany rozwinięciem Shannona (Biliński, 1996; de Micheli, 1998):

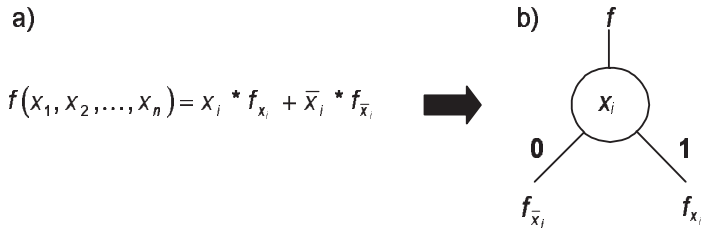
$$f(x_1, x_2, \ldots, x_n) = x_i * f_{x_i} + \bar{x}_i * f_{\bar{x}_i} \tag{2.1}$$

gdzie:

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n),$$

$$f_{\bar{x}_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

są zwane odpowiednio pozytywnym oraz negatywnym dopełnieniem algebraicznym funkcji f ze względu na zmienną x_i. Równanie 2.1 może zostać przedstawione w postaci *BDD* w sposób pokazany na rysunku 2.7:



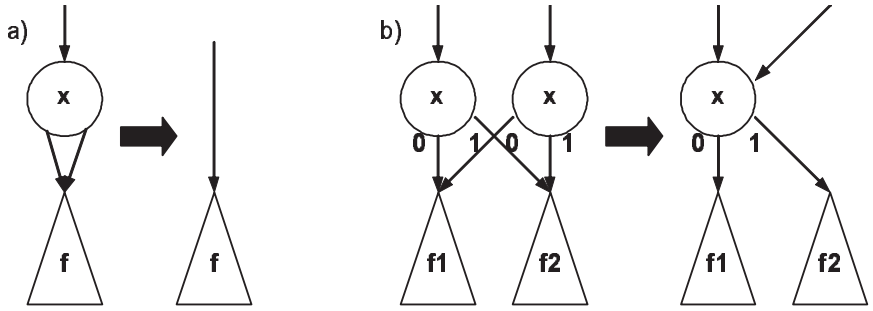
Rys. 2.7. Idea *BDD*: a) rozwinięcie Shannona, b) węzeł *BDD*

Stosując rekursywnie rozwinięcie Shannona, można stworzyć diagram *BDD* reprezentujący dowolną funkcję logiczną. Zredukowanym uporządkowanym binarnym diagramem decyzyjnym (*ang.* Reduced *OBDD*) jest diagram *OBDD*, z którego usunięte zostały węzły nadmiarowe. Do zamiany diagramu *OBDD* na diagram *ROBDD* stosuje się następujące reguły redukcji przedstawione na rysunku 2.8:

- węzły których krawędzie wychodzące wskazują na ten sam węzeł potomny – są usuwane,

- podgrafy izomorficzne – są współdzielone.

Gdy porządek zmiennych jest ustalony, diagram *ROBDD* dowolnej funkcji logicznej jest formą kanoniczną. Oznacza to, że jeżeli dwa diagramy *ROBDD* są identyczne, to ich funkcje logiczne są równoważne. Ten fakt ma istotne znaczenie dla implementacji programowej, stąd chcąc na przykład porównać dwie funkcje boolowskie, wystarczy porównać ich diagramy *ROBDD*, lub chcąc zrobić kopię diagramu, wystarczające jest skopiowanie jego wskaźnika, poprzez co oszczędza się na czasie oraz pamięci.



Rys. 2.8. Reguły redukcji węzłów dla *BDD*: a) eliminacja węzłów, b) współdzielenie węzłów

Mając funkcję logiczną przedstawioną w postaci *ROBDD*, możliwe jest przekształcenie diagramu na równania logiczne zapisane w językach *HDL*. Przeglądając drzewo w sposób prefiksowy (tzn. najpierw wykonując operację związaną z węzłem, np. zapis do pliku zmiennej z węzła, a następnie schodząc w dół drzewa), można otrzymać wyrażenie niezawierające ani negacji sumy ani negacji iloczynu. W pewnym sensie, jest to postępowanie odwrotne do rozwinięcia Shannona.

2.6.2. Funkcja charakterystyczna

Główną cechą metod symbolicznych jest operowanie pojęciem zbioru stanów, symbolicznie reprezentowanym przez funkcję charakterystyczną. Samo pojęcie funkcji charakterystycznej jest dobrze znane w teorii algebry (Rasiowa, 1998) i może być zastosowane do reprezentowania zbioru stanów (Ghosh i in., 1992; Biliński, 1996).

Definicja 2.3. Funkcją charakterystyczną *zbioru elementów, jest funkcja boolowska* $X_A : U \rightarrow \{0,1\}$*, określona w następujący sposób:*

$$X_A(x) = \begin{cases} 1 \Leftrightarrow x \in A, \\ 0 \Leftrightarrow x \notin A. \end{cases} \tag{2.2}$$

Jak widać to z definicji 2.3, funkcja charakterystyczna może być wprost obliczana jako suma wszystkich elementów jej zbioru. Operacje na zbiorach są w bez-

pośrednim powiązaniu z działaniami logicznymi na ich funkcjach charakterystycznych. Związek ten jest następujący:

$$X_{(A\cup B)} = X_A + X_B; \qquad X_{(A\cap B)} = X_A * X_B; \qquad X_{(\overline{A})} = \overline{X_A} \tag{2.3}$$

W programach komputerowych typu *CAD* zbiory stanów mogą być reprezentowane jako funkcje charakterystyczne i w sposób efektywny implementowane z użyciem diagramów *BDD* (Coudert i in., 1989; Burch i in., 1990; Ghosh i in., 1992; Biliński, 1996; Rausch i Krogh, 1998; Minato, 1996; Łabiak, 2001*c*; Miczulski, 2002*a*).

2.7. Układy programowalne i układy FPGA

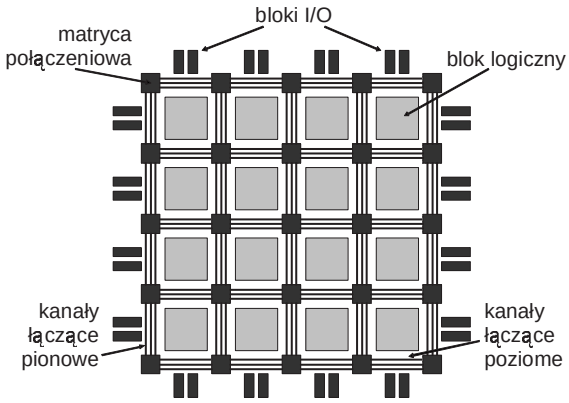
Specjalizowane układy cyfrowe (Majewski i in., 1992), według (Łuba i Zbierzchowski, 2000), można podzielić następująco:

- układy zamawiane przez użytkownika (*ang.* full custom),
- układy projektowane przez użytkownika (*ang.* semi custom),
- układy programowane przez użytkownika (*ang.* Field Programmable Logic Devices – *FPLD*).

Największą atrakcyjnością wśród inżynierów elektroników, ze względu na koszty i wygodę projektowania, cieszą się układy *FPLD*. Na rynku dostępnych jest bardzo wiele propozycji różniących się pod względem technologii, złożoności, ceny oraz architektury. Najważniejszym kryterium porządkującym jest podział układów uwzględniający ich strukturę wewnętrzną, czyli rozmieszczenie elementów logicznych i połączeń między nimi. Przyjmując takie kryterium układy *FPLD* dzieli się następująco (Łuba i Zbierzchowski, 2000):

- *PLD* (*ang.* Programmable Logic Devices) — głównym elementem jest matryca *AND-OR* (typu *PAL* lub *PLA*) oraz zespoły elementów wyjściowych,
- *CPLD* (*ang.* Complex *PLD*) — cechą charakterystyczną jest matryca połączeń *PIA* (*ang.* Programmable Interconnect Array), łącząca makrokomórki logiczne,
- *FPGA* (*ang.* Field Programmable Gate Array) — wewnętrzną strukturę stanowi macierz elementów logicznych między którymi poprowadzone są kanały połączeniowe (rys. 2.9).

Największymi producentami układów *FPGA* są firmy *Atmel*, *Quick Logic* oraz *Xilinx*, która jako wiodący dostawca układów w tej technologii w roku 1999 posiadała 35% udziału w rynku układów programowalnych (Pasierbiński i Zbysiński, 2001). Głównymi układami *FPGA* oferowanymi przez firmę *Xilinx* są rodziny układów *Spartan*, *Spartan II*, *Virtex* i *Virtex II*.



Rys. 2.9. Architektura typowego układu *FPGA* (Pasierbiński i Zbysiński, 2001)

Architektura układów *Spartan* przypomina wcześniejszą, już nie produkowaną, rodzinę układów serii 4000. Komórka logiczna tych układów, dzięki zastosowaniu trzech tablic *LUT* (*ang.* Look-Up Table – tablica pełniąca rolę tabeli prawdy funkcji logicznej), pozwala na realizację trzech dowolnych funkcji logicznych (dwie 4-argumentowe i jedna 3-argumentowa), jednej funkcji 9-argumentowej, dwóch funkcji 4- i 6-argumentowych lub jednej 5-argumentowej. Na wyjściach bloku zastosowano dwa przerzutniki. Układy rodziny Virtex zaprojektowano wzorując się na architekturze układów rodziny *Spartan II*, co tłumaczy podobieństwo w budowie tych układów. Blok logiczny tych układów składa się z dwóch 4-wejściowych tablic *LUT* i również jest wyposażony w dwa przerzutniki na wyjściach. Z kolei układy z serii *Virtex II*, stanowią najnowszą propozycję firmy *Xilinx* i jednocześnie są układami największymi, oferującymi nawet do 10 milionów bramek. Budowa pojedynczego bloku logicznego, całkowicie odmienna od układów wcześniejszych, składa się z czterech komórek logicznych, oficjalnie przez producenta nazywanych *slice*, z których każda zbudowana jest z dwóch 4-wejściowych tablic *LUT*, generatora szybkich przeniesień, bramek pomocniczych i dwóch przerzutników na wyjściach.

Niektóre z dostępnych układów *FPGA* oferują możliwość rekonfigurowania ich funkcji w czasie pracy, co jest nowym wyzwaniem dla metod projektowania układów cyfrowych (Łuba i Zbierzchowski, 2000).

2.8. Podsumowanie

W rozdziale scharakteryzowano układ sterowania oraz jego zastosowanie, przedstawiono te sposoby projektowania układów, które wobec diagramów statechart są metodami tradycyjnymi i wobec których hierarchiczny model automatu współbieżnego stanowi naturalne ich rozwinięcie. Tak więc kolejno omówiono automat sekwencyjny i automat współbieżny. Przedstawiono również hierarchiczną sieć Petriego jako konkurencyjną metodykę opisu. Ponadto w rozdziale przedstawiono te

techniki i technologie realizacji sterowników cyfrowych, które znalazły praktyczne zastosowanie w badaniach nad bezpośrednią implementacją proponowanego modelu automatu w strukturach programowalnych. Wymiernym wynikiem prowadzonych badań jest realizacja komputerowego systemu automatycznego projektowania (system *HiCoS*, rozdział 8), w którym wejściem jest własny język opisu sprzętu, a w wyjściem model w języku *VHDL* przeznaczony do implementacji w strukturach programowalnych (np. *FPGA*). Stąd w rozdziale zamieszczono krótką charakterystyka języków opisu sprzętu. Dodatkowo w systemie zaimplementowano, jako poboczną ścieżkę projektową, algorytm generowania symbolicznej przestrzeni stanów, wykorzystując do tego celu binarne diagramy decyzyjne i pojęcie funkcji charakterystycznej.

Rozdział 3

DIAGRAMY STATECHART W MODELOWANIU ZACHOWANIA

3.1. Język UML

Jednym z głównych zadań programowania jest, szeroko rozumiane, komputerowe modelowanie świata rzeczywistego, a technologia *UML* ma za zadanie modelować to, co będzie programem, reprezentującym wybrany fragment rzeczywistości. Skoro możliwości modelowania języka *UML* nie są ograniczone do żadnego partykularnego wycinka świata, to można tę technologię wykorzystać do modelowania tych aspektów, które dotyczą układów cyfrowych, tym bardziej, że obserwuje się silne podobieństwa między procesem projektowania sprzętu i oprogramowania. W kontekście niniejszej pracy należy zwrócić szczególną uwagę na fakt, że zarówno układ cyfrowy jak i obiekt programowy (czy też w ogólności komponent) mogą cechować się działaniem reaktywnym. Układy cyfrowe nie są jedynym nowym obszarem zastosowań języka *UML*. W pracy (Mrozek, 2001) autor promuje *UML* jako język projektowania systemów mechatronicznych, zwłaszcza w aspekcie pracy zespołów interdyscyplinarnych, operujących różnymi technologiami. Co więcej, jedną z postulowanych technologii przedstawionych w publikacji (Mrozek, 2001) są właśnie układy programowalne, lecz w artykule brak jest jednak jakichkolwiek informacji o szczegółach realizacyjnych. Zdaniem autora, cechy *UML* (w szczególności pojęcie stereotypu (Subieta, 1999*b*; Booch i in., 2001; UML, 2003)) oraz jego wzrastająca popularność (np. nowe zastosowania przy projektowaniu systemów czasu rzeczywistego (Licht i Fengler, 2003; Lu i in., 2003)), uzasadniają potrzebę zbadania przydatności tej technologii w projektowaniu układów cyfrowych.

3.1.1. Krótka charakterystyka języka UML

Język *UML* (*ang.* Unified Modelling Language) jest wynikiem połączonych wysiłków trzech znanych metodologów: Grady Booch'a, Ivara Jacobsona, James'a Rumbaugh'a (Subieta, 1999*c*; Booch i in., 2001; UML, 2003). Każdy z nich niezależnie opracował własne metodyki, które ujmują różne aspekty projektowania obiektowego. Obecnie ich prace są prowadzone w ramach firmy *Rational Software Corporation*. Pierwsza wersja języka (0.8) została opublikowana w roku 1995, a aktualną jest wersja 1.5 z roku 2003 (UML, 2003) ale już jest mowa o wersji 2.0.

Podstawowym celem *UML* jest modelowanie różnego rodzaju systemów (nie tylko z dziedziny oprogramowania) z wykorzystaniem pojęć obiektowych. Definicja

przytoczona za (Subieta, 1999*c*), którą można znaleźć w (Subieta, 1999*c*), brzmi następująco:

> „UML jest językiem do specyfikacji, konstruowania, wizualizacji i dokumentowania wytworów związanych z systemami intensywnie wykorzystującymi oprogramowanie."

Jak widać, celem tego języka jest stworzenie pomostu między ludzkim rozumieniem struktury i działania systemów, a kodem wynikowym. Ponadto język ten ma służyć do specyfikowania, konstrukcji, wizualizacji i dokumentacji systemów, posiadających cechy programów. Zakresem swym język ten (Subieta, 1999*a*):

- obejmuje specyfikację, konstrukcję, wizualizację i dokumentację,
- łączy w sobie różne dotychczas istniejące metodyki w jedną zunifikowaną,
- posiada wsparcie dla systemów współbieżnych i rozproszonych,
- skupia się na podejściu projektowym, w którym najistotniejszym jest postrzeganie projektowanego systemu z punktu wykorzystania przez przyszłego użytkownika lub innego nadrzędnego systemu.

Zazwyczaj proces projektowania wymaga od projektanta postrzegania systemu w różnych aspektach. Najczęściej trzeba odpowiedzieć na pytania, dla kogo system jest tworzony, jak się z niego będzie korzystać, co wchodzi w jego skład i jak zostanie zrealizowany. Podstawowym środkiem oferowanym przez *UML* do tego celu są diagramy. Zadaniem diagramów jest spoglądanie na system z różnych perspektyw. Oto niektóre z diagramów:

- diagramy przypadków użycia (*ang.* use case diagram) – rolą ich jest przedstawienie funkcji projektowanego systemu w taki sposób, w jaki będą je widzieć jego przyszli użytkownicy,
- diagramy klas (*ang.* class diagrams) – diagramy te, w przypadku modelowania programów w języku *Java* czy *C++*, przedstawiają typy danych, charakterystyczne dla tych języków i związki między nimi (np.: klasy, składowe, metody, dziedziczenie, osadzanie obiektów),
- diagramy odwzorowujące dynamiczne zachowanie systemu:
  - diagramy sekwencji (*ang.* sequential diagram) – przedstawiają sekwencje komunikatów i zdarzeń, przesyłanych między obiektami dla pewnego przypadku użycia,
  - diagramy współpracy (*ang.* collaboration diagram) – podobnie jak diagramy sekwencji, ukazują interakcje między modelowanymi obiektami, z uwzględnieniem ich statycznej struktury,
  - diagramy stanów (*ang.* statechart diagram) – przedstawiają dynamikę stanów obiektów,

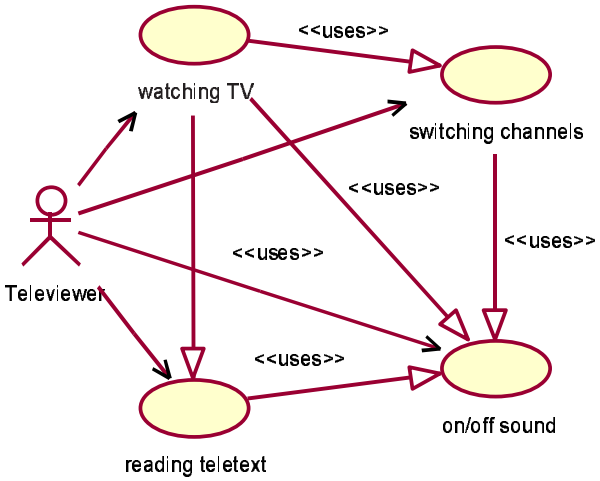
  - diagramy aktywności (*ang.* activity diagram),
  - diagramy przepływu sterowania (*ang.* flowchart diagram),

- diagramy implementacyjne:
  - diagramy komponentów (*ang.* component diagram),
  - diagramy wdrożeniowe (*ang.* deployment diagram).

Dla potrzeb projektowania współbieżnych kontrolerów cyfrowych, najistotniejszymi diagramami wydają się być diagramy przypadków użycia, aktywności, współpracy i oczywiście diagramy stanów, które wprost odwołują się do pojęcia automatu skończonego (Harel, 1987; Łabiak, 2001*b*; Łabiak, 2001*c*).

3.1.2. Zastosowanie UML w procesie projektowania układów cyfrowych

Język *UML* zawiera w sobie całe bogactwo środków. Nie wszystkie z nich dobrze nadają się do wykorzystania podczas projektowania sterowników (kontrolerów) cyfrowych, bo też i nie taka potrzeba przyświecała jego twórcom. Z drugiej jednak strony, język ten ma służyć pomocą w projektowaniu jak najszerszej gamy różnych systemów, a zatem wydaje się interesujące zbadanie możliwości języka pod kątem wykorzystania w procesie projektowania kontrolerów cyfrowych i ich implementacji, za pośrednictwem języka *VHDL*, w elementach *FPGA*.

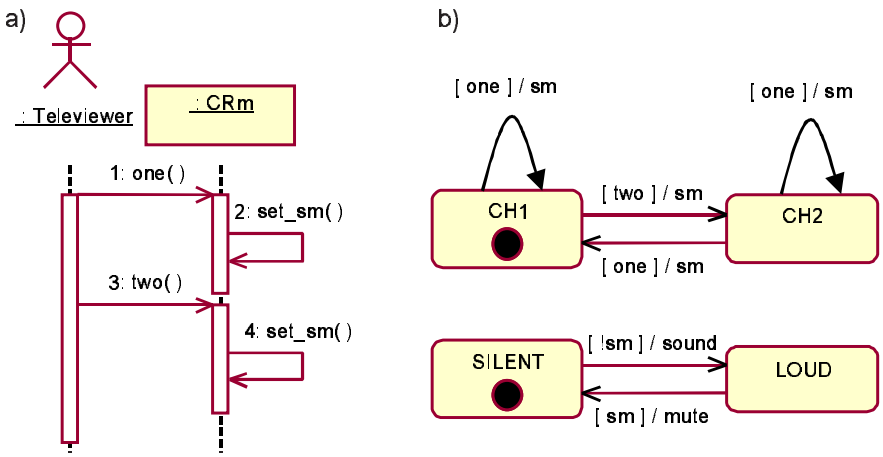


Rys. 3.1. Diagram przypadków użycia dla pilota telewizyjnego (oprac. na podst. (Nazareth i in., 1996))

Zadaniem pierwszego etapu projektowania, nazywanego specyfikacją (Adamski, 1990), jest analiza potrzeb użytkownika. Do tego celu można wykorzystać diagramy przypadków użycia, gdzie podaje się w jaki sposób system będzie współpracował z użytkownikiem lub innym systemem. Rysunek 3.1 przedstawia przykładowy

diagram przypadków użycia dla układu, który realizuje funkcje pilota telewizyjnego. Przykład ten jest zaczerpnięty z pracy (Nazareth i in., 1996), a diagramy zostały zrealizowane przy użyciu oprogramowania *Rational Rose* (Rat, 2004). Piktogram z rysunku 3.1, przedstawiający człowieka nazywanego w *UML* aktorem, symbolizuje system nadrzędny, tzn. taki system (w tym przypadku człowieka), który z układu będzie korzystać. Owale wraz z podpisami oznaczają realizowane przypadki użycia.

Poszczególne przypadki użycia można poddać bardziej szczegółowej analizie. Szczególnie istotnym jest przedstawienie zachowania układu pod wpływem przychodzących sygnałów. Rysunek 3.2a zawiera diagram sekwencji dla przypadku użycia *switching channels*. Przełączanie kanałów polega na naciskaniu przycisków *one* lub *two* (w przypadku telewizora posiadającego dwa kanały), w odpowiedzi na które jest ustawiany sygnał *set_sm*. Zadaniem tego sygnału jest krótkotrwałe wyłączenie niepożądanego szumu, związanego ze zmianą kanałów.



Rys. 3.2. Przypadek użycia switching channels: a) diagram sekwencji, b) diagram stanów

Diagramy współpracy pokazują działanie projektowanego układu (lub jak w tym przypadku jego fragmentu) z uwzględnieniem chronologii zdarzeń i przy ich pomocy nie jest możliwe zamodelowanie w sposób kompletny i jednoznaczny funkcjonowania całego systemu. Do tego celu można wykorzystać diagramy stanów (diagramy statechart) (Harel, 1987). Diagramy te pozwalają na kompletne i jednoznaczne specyfikowanie zachowanie układu. Ten rodzaj opisu spełnia wymogi formalne stawiane przez syntezę strukturalną (Adamski, 1990). Rysunek 3.2b przedstawia diagram stanów dla omawianego wcześniej przypadku użycia *switching channels*.

Wydaje się, że notacje języka *UML* mogą dobrze wspierać proces projektowania układów cyfrowych, a ich rola jest szczególnie istotna na etapie specyfikacji oraz w procesie dokumentowania projektu. Postępując zgodnie za przedstawionymi w pracy (Adamski, 1990) zasadami projektowania układów cyfrowych sys-

tematyczną metodą strukturalną, autor pokusił się o zaproponowanie metodologii projektowej wykorzystującej technologię *UML*, gdzie autorski system *HiCoS* (rozdział 8) wykonuje automatyczną syntezę. Tabela 3.1 wymienia fazy procesu projektowania, podając kolejno wykorzystywane diagramy *UML* oraz rolę systemu *HiCoS*. Kursywą zaznaczono elementy niewspierane przez *HiCoS* lub realizowane poza środowiskiem wykorzystując dane z programu.

Tab. 3.1. *UML* w projektowaniu układów cyfrowych (propozycja) oraz metodologia systemu *HiCoS*

Etapy projektowania	Język *UML*	System *HiCoS*
specyfikacja	diagramy stanów	opis graficzny/tekstowy (*ssf*)
uwiarygodnienie specyfikacji	przypadki użycia, diagramy stanów, diagramy sekwencji, diagramy aktywności	trawers przestrzeni stanów, *metody symboliczne*
synteza		opis równaniami logicznymi, *optymalizacja*, translacja na *VHDL*
uwiarygodnienie realizacji		*symulacja poziomu VHDL,*
dokumentowanie projektu	przypadki użycia, diagramy stanów, diagramy sekwencji, diagramy aktywności	opis graficzny/tekstowy (*ssf*)
przygotowanie testów		

System *HiCoS* został opracowany jako system *CAD*, dla którego technologia *UML* ma za zadanie, zgodnie ze swym założeniem, specyfikować, wizualizować i dokumentować takie artefakty dotyczące systemów reaktywnych jak interakcje, docelowe wykorzystywanie, przejścia, czy w ogólności zachowanie. Głównym zadaniem systemu jest synteza modelu sterownika binarnego, specyfikowanego diagramami statechart dla potrzeb implementacji w strukturach programowalnych.

W proponowanym procesie projektowania sterowników cyfrowych, specyfikacja ogranicza się do formalnego sporządzenia diagramów stanów, ilustrujących zachowanie. W systemie *HiCoS* jest to reprezentowane przez równoważną postać tekstową (format *SSF*, podrozdział 8.2 oraz dodatek A). Uwiarygodnienie specyfikacji odbywa się poprzez podanie diagramów przypadków użycia, diagramów sekwencji i aktywności. Również pomocne mogą być diagramy stanów. Te zapisy nie wymagają głębokiej wiedzy inżynierskiej i mogą służyć w rozmowach ze zleceniodawcą projektu układu, często nieposiadającego odpowiedniego przygotowania technicznego. Dodatkowo specyfikację można uzasadniać bardziej szczegółowo metodami symbolicznymi, które pozwalają ustalić, czy zadany układ jest np. żywotny czy

bezpieczny. Do tego celu wykorzystuje się przestrzeń stanów układu, np. reprezentowaną przez funkcję charakterystyczną (podrozdział 8.6). Etap syntezy jest wykonywany automatycznie, a jego wynikiem są równania logiczne opisujące działanie specyfikowanego układu (podrozdział 7.2). Jest to również miejsce dla wykonania stosownych optymalizacji. W tej fazie i fazie następnej – jaką jest uwiarygodnienie realizacji – technologia *UML* nie oferuje żadnych propozycji. Uwiarygodnienie projektu może być częściowo dokonane na drodze symulacji specjalnego modelu testowego (*ang.* testbenches) w języku *HDL*. Niektóre systemy, jak np. *Statemate MAGNUM* czy *visualSTATE*, do uwiarygodnienia oferują możliwość uruchomienia specyfikowanego modelu w taki sposób, jak by to miało być wykonane przez rzeczywisty układ. Dokumentowanie, jako jeden z głównych celów *UML*, jest zbiorem wszystkich diagramów, tworzonych w kolejnych fazach procesu projektowania.

3.1.3. Modelowanie systemów reaktywnych

Głównym środkiem specyfikowania zachowania systemu reaktywnego, w proponowanej metodyce, są diagramy statechart. Dodatkowo diagramy interakcji szczegółowo mogą opisywać zachowanie systemu związane z konkretnym przypadkiem użycia. Najważniejszą czynnością modelowania zachowaia jest określenie trzech rzeczy (Booch i in., 2001): stanów (czyli warunków w których system może się znaleźć przez pewien zauważalny czas), zdarzeń (uruchamiających przejścia) i akcji (głównie generowane zdarzenia związane z realizacją przejść). Ponadto w pracy (Booch i in., 2001) można znaleźć m.in. takie oto szczegółowe zalecenia dotyczące czynności projektowych:

- ustalenie otoczenia systemu, okrślenie stanu początkowego i końcowego,
- ustalenie możliwych stanów systemu, poczynając od stanów najwyższego poziomu,
- określenie częściowego porządku stanów w historii życia systemu,
- wskazanie zdarzeń uruchamiających przejścia między stanami, zapisanych w postaci listy wcześniej opracowanych sekwencji częściowego porządku stanów,
- skojarzenie akcji z przejściami,
- wprowadzenie podstanów, złączeń i rozgałęzień sterowania celem uproszczenia modelu,
- sprawdzenie podstawowych warunków bezpieczeństwa i żywotności,
- symulacja: ręczna lub automatyczna.

Tak określone zalecenia decydowały o funkcjonalności przygotowanego programu *HiCoS* oraz stanowiły bazę dla zdefiniowania semantyki diagramów wykorzystywanej przez system. Nie wszystkie założone funkcje udało się zrealizować w programie, lecz rezultaty już uzyskane stanowią solidną podstawę do dalszych prac.

3.2. Programowa maszyna stanów i język SpecCharts

Programowa maszyna stanów (Gajski i in., 1993; 1994; 2001) (*ang.* Program State Machine lub w skrócie *PSM*) jest modelem, który łączy w sobie cechy hierarchicznej współbieżnej maszyny o skończonej liczbie stanów (*ang.* Hierachical Concurrent Finite-State Machine lub w skrócie *HCFSM*) z właściwościami klasycznych języków opisu sprzętu (*HDL*). W modelu tym system jest przedstawiany jako hierarchia stanów programowych (*ang.* program-state), gdzie każdy ze stanów reprezentuje tryb przetwarzania.

Stan programowy może być albo stanem złożonym (*ang.* composite), albo stanem podstawowym, czasami nazywany liściem (*ang.* leaf). Programowemu stanowi złożonemu, w odróżnieniu od programowego stanu podstawowego, można hierarchicznie przyporządkować zbiór programowych stanów współbieżnych – tzn. takich, które w danym momencie czasu albo wszystkie są aktywne albo żaden nie jest aktywny lub zbiór programowych stanów sekwencyjnych – tzn. takich, z których w danym momencie czasu, co najwyżej jeden może być aktywny. Sekwencyjny stan programowy zawiera w sobie zbiór przejść, graficznie przedstawianych jako łuki, które określają sekwencje przekazywania sterowania między podległymi stanami programowymi. Rozróżnia się dwa rodzaje przejść między stanami. Pierwsze, zwane „przejściem po skończeniu" (*ang.* transition-on-completion arc, lub w skrócie TOC), realizowane jest, gdy źródłowy stan programowy skończy swoje obliczenia i przypisany przejściu warunek ma wartość prawdy. Drugi rodzaj przejścia to „przejście natychmiastowe" (*ang.* transition-immediately arc, lub w skrócie TI), które jest wykonywane natychmiast przy spełnieniu warunku skojarzonego z przejściem, niezależnie od stanu obliczeń w źródłowym stanie programowym. Programowe stany podstawowe znajdują się na najniżej położonych wierzchołkach drzewa hierarchii i obliczenia realizowane przez te stany, mogą być wyrażone poprzez instrukcje klasycznego języka programowania lub języka opisu sprzętu.

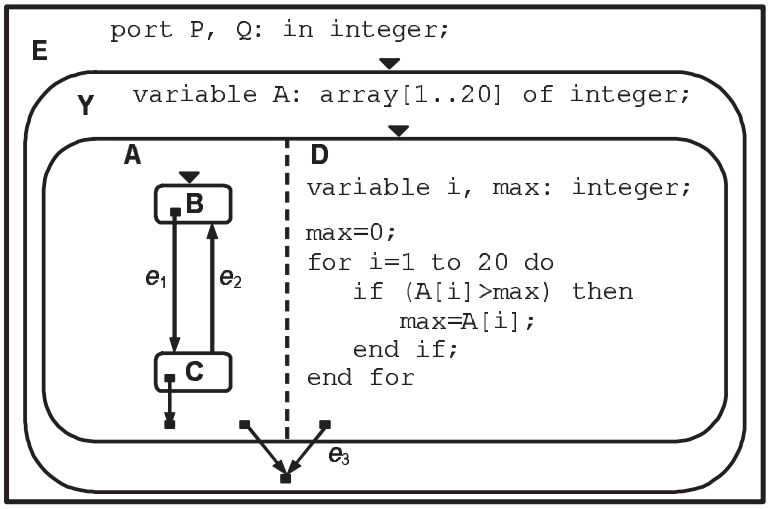
Język *SpecCharts* (kod 3.1) spełnia paradygmat modelu *PSM* i w swej idei jest bardzo podobny do języka *VHDL*. Konstrukcje językowe takie jak *process* czy *block* zostały zastąpione konstrukcją *behavior*, reprezentującą w języku stan programowy. Działanie modelowanego systemu opisywane jest jako zbiór hierarchicznych zachowań (*ang.* behavior). Opis zachowania jest zagnieżdżony w swoim zachowaniu rodzicielskim i może zawierać takie elementy języka *VHDL* jak deklaracje typów, sygnałów, zmiennych czy procedur, których zakres rozciąga się na podległe zachowania. Przejścia reprezentowane są przez trójkę $\langle T, C, NB\rangle$, gdzie T oznacza typ przejścia (TOC lub TI), C jest warunkiem, którego spełnienie jest konieczne do realizacji przejścia, NB jest następnym zachowaniem, do którego jest przekazywane sterowanie. O zachowaniu sekwencyjnym można powiedzieć, że kończy swoje obliczenia, gdy sterowanie zostanie przekazane do specjalnego punktu zakończenia przetwarzania, który jest określany poprzez słowo kluczowe *complete* umieszczone w części NB w trójce reprezentującej przejście (kod 3.1). Przejście typu TOC może być zrealizowane wówczas, gdy warunek związany z przejściem jest spełniony i zachowanie źródłowe wraz ze wszystkimi mu podległymi inny-

mi zachowaniami zakończyły swoje obliczenia. Natomiast realizacja przejścia typu TI uzależniona jest tylko od warunku związanego z tranzycją, co oznacza, że takie przejście może przerwać działanie podległych zachowań źródłowych.

Kod 3.1. Przykład specyfikacji w języku *SpecCharts* — postać tekstowa (Gajski i in., 1994)

```
entity E is
    port(P: in integer; Q: out integer);
end E;
architecture Arch of E is
begin
    behavior Y type councurrent subbehaviors is
        viariable A: array[1..20] of integer;
    begin
        A: (TOC, true, complete);
        D: (TOC, e3, complete);
        behavior A type sequential subbehaviors is
        begin
            B: (TOC, e1, C);
            C: (TI, e2, B);
            C: (TOC, true, complete);
            behavior B type code is ...
            behavior C type code is ...
        end A;
        behavior D type code is
            variable i, max: integer;
        begin
            max:=0;
            for i in 1 to 20 loop
                if (A(i)>max) then max:=A(i);
                end if;
            end for;
        end D;
    end Y;
end Arch;
```

Oprócz postaci tekstowej język *SpecCharts* oferuje inną, równoważną reprezentację modelowanego zachowania. Przedstawienie stanów i przejść w postaci graficznej, jak się powszechnie uważa, w poglądowy sposób pozwala rozumieć działanie modelu. Rysunek 3.3 przedstawia graficzną reprezentację modelu, równoważną reprezentacji tekstowej (kod 3.1). Jak to widać z przykładowego rysunku, postać graficzna jest bardzo podobna do diagramów statechart. Zachowanie jako całość reprezentowane jest przez prostokąt, a pozostałe zachowania reprezentowane są przez krągłokąty. Zachowania będące w relacji współbieżności oddzielone są linią przerywaną. Stany początkowe wskazywane są przez odwrócone czarne trójkąty. Przejścia typu TOC oznaczane są przez strzałki, które biorą swój początek w kwadraciku umieszczonym w obrębie stanu reprezentującego zachowanie źródłowe (na przykład przejście z etykietą e_1). Natomiast Przejścia typu TI, oznaczane również



Rys. 3.3. Przykład specyfikacji w języku *SpecCharts* — postać graficzna (Gajski i in., 1994)

jako strzałki, biorą swój początek z obrzeży stanów źródłowych (na przykład przejście z etykietą e_2). Zakończenie obliczeń dokonywanych przez stany sekwencyjne realizowane jest przez przejście do punktu końcowego (*ang.* completion point), oznaczanego jako zaczerniony kwadracik.

3.3. Notacja graficzna i znaczenie diagramów statechart

Diagramy statechart mogą być stosowane do opisu zachowania modelowych elementów, takich jak obiekty czy interakcje. Przedstawiają one możliwe sekwencje stanów i akcji, poprzez które modelowany element przechodzi w ciągu okresu swego życia. Kolejne sekwencje stanów stanowią reakcje na dyskretne zdarzenia stymulujące modelowany element.

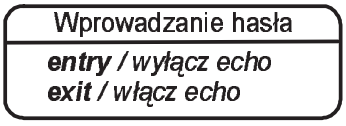
Diagram statechart jest grafem, który reprezentuje maszynę stanową (Booch i in., 2001). Stany przedstawione są poprzez symbole stanu, a przejścia (zwane również tranzycjami) poprzez skierowane łuki łączące symbole stanów. Konkretnym stanom można przypisywać poddiagramy. Każdy diagram statechart zawiera stan najwyższego poziomu hierarchii, któremu przyporządkowane są wszystkie elementy diagramu.

Notacja przedstawiona w niniejszym podrozdziale, jest notacją zasadniczo zgodną z definicją języka *UML* (UML, 2003). Dokonując doboru omawianych elementów zapisu graficznego, przede wszystkim kierowano się ich przydatnością w modelowaniu zachowania sterowników binarnych (punkt 3.1.1) oraz sprzętowymi możliwościami realizacyjnymi (punkt 7.2.2). Z tych powodów zrezygnowano z tranzycji czasowych oraz ze ścieżek obliczonych przejść (*ang.* factored transition path), stosowanie których wymaga operowania liczbami (np. całkowitymi). Do-

datkowo zrezygnowano z tranzycji przekraczających granice stanów. Ta ostatnia rezygnacja jest wynikiem założonego wstępnie podziału prowadzonych badań na etapy. W pierwszym etapie postanowiono zająć się prostszym podzbiorem diagramów, ograniczając ten podzbiór paradygmatem modularności. Zabronienie stosowania tranzycji przekraczających granice stanów wyraźnie wyznacza zbiór diagramów, które można określić wspólnym mianem modularnych. Wynikiem tego założenia jest niestosowanie tranzycji złożonych oraz stanów synchronizujących. Ponadto pewnej modyfikacji uległo operowanie atrybutem historii. Autor jednocześnie uważa, że pomyślne przeprowadzenie badań, dla tak przyjętego ograniczonego podzbioru, stanowi dobrą podstawę do kolejnego etapu badań, obejmującego całą semantykę diagramów statechart.

3.3.1. Stany proste

Dla modelowanego obiektu lub interakcji, przebywanie w stanie (*ang.* state) oznacza, spełnienie określonych warunków, wykonywanie pewnej akcji lub oczekiwanie na zdarzenie w czasie trwania interwału czasowego. Graficznie stan jest przedstawiany jako prostokąt o zaokrąglonych narożnikach (rys. 3.4), zwanymi krągłokątami (termin wzięty z polskiego wydania pozycji (Harel, 2001)).



Rys. 3.4. Przykład stanu prostego

Krągłokąt stanu prostego może być podzielony na kilka części oddzielonych linią poziomą. Część znajdująca się najwyżej, przeznaczona jest do przechowywania napisu będącego nazwą stanu. Dopuszcza się występowania stanów anonimowych, przy czym wszystkie takie stany są rozróżnialne. Część krągłokątu akcji wewnętrznych przeznaczona jest do przechowywania informacji o akcjach przypisanych stanowi, których wykonanie związane jest z aktywnością stanu. Akcje specyfikowane są według następującego formatu:

etykieta akcji / wyrażenie akcji.

Etykieta akcji określa okoliczności, przy spełnieniu których zostanie zrealizowana akcja określona wyrażeniem umieszczonym za znakiem ukośnika. W systemie stosowane są następujące etykiety:

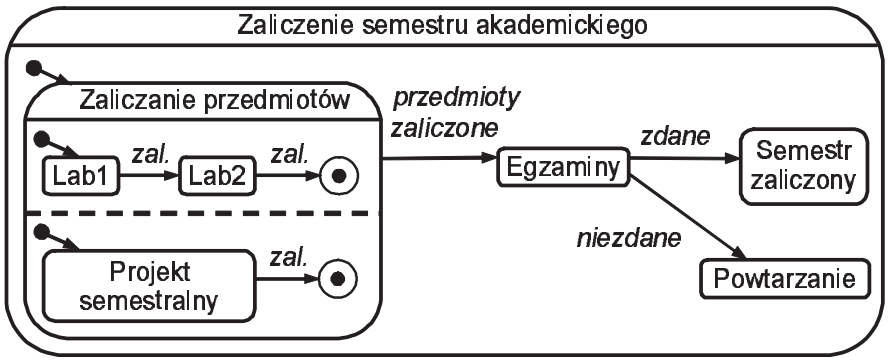
- *entry* – oznacza wykonanie akcji w momencie aktywowania stanu (tzw. akcja wejściowa),
- *exit* – oznacza wykonanie akcji gdy sterowanie opuszcza stan (tzw. akcja wyjściowa),

- *do* – oznacza wykonywanie akcji w całym czasie aktywności stanu (tzw. akcja statyczna).

W opracowanym systemie wyrażenie akcji jest zbiorem rozgłaszanych zdarzeń.

3.3.2. Stany złożone

Na stan złożony (*ang.* composite state) składają się dwa lub więcej podstanów współbieżnych, zwanych regionami, lub taka grupa podstanów rozłącznych, z których w danym momencie tylko jeden może być stanem aktywnym (automat sekwencyjny). Każdy region stanu złożonego posiada swój stan początkowy i może posiadać stan końcowy (rys. 3.5). Tranzycja dochodząca do stanu złożonego jest jednocześnie tranzycją do stanu początkowego regionu. Tranzycja do stanu końcowego reprezentuje zakończenie aktywności regionu. Zrealizowanie tranzycji we wszystkich współbieżnych regionach do ich stanów końcowych (o ile takie są zadeklarowane) oznacza zakończenie aktywności stanu złożonego i jednocześnie jest warunkiem koniecznym odpalenia tranzycji wychodzącej ze stanu złożonego.



Rys. 3.5. Przykład stanu złożonego

Na diagramie (np. rys. 3.5) stan złożony, obok części nazwowej i akcyjnej, zawiera dodatkową część, w której mogą znajdować się regiony diagramów podległych. Granice regionów współbieżnych oddzielone są od siebie linią przerywaną. Każdy z regionów może posiadać nazwę i musi zawierać diagram rozłącznych stanów.

Na rysunku stan początkowy oznaczany jest strzałką z czarnym kółkiem na początku, stan końcowy zaś, jako czarne kółko otoczone okręgiem (*ang.* bull's eye).

3.3.3. Zdarzenia

Zdarzenie (*ang.* event) jest informacją o czymś, co jest warte zauważenia. W realizacji fizycznej nośnikiem tej informacji mogą być sygnały. Zdarzenia mogą się pojawiać w wyniku akcji związanych ze stanem (punkt 3.1.1) lub mogą być wynikiem realizacji przejść (tranzycji, punkty 3.3.4 i 3.3.5), czy też mogą przychodzić

do układu ze świata zewnętrznego. Skutkiem wystąpienia zdarzeń może być realizacja tranzycji. Na zdarzeniach można wykonywać operacje logiczne, które z kolei mogą być predykatami nałożonymi na tranzycje. Wówczas ich spełnienie jest koniecznym warunkiem realizacji przejścia. Zasięg zdarzeń jest globalny, tzn. że każde rozgłaszane w układzie zdarzenie jest dostępne we wszystkich jego częściach.

3.3.4. Tranzycje proste

Tranzycja prosta (*ang.* simple transition), czasami zwana przejściem (Booch i in., 2001), jest pewnego rodzaju powiązaniem między dwoma stanami wskazującym, że przy spełnieniu określonego warunku modelowany obiekt lub interakcja, będące w pierwszym stanie, przejdą do stanu następnego i zostanie wówczas wykonana określona akcja. Mówi się że taka zmiana stanu jest „odpaleniem" (*ang.* fire) lub realizacją tranzycji.

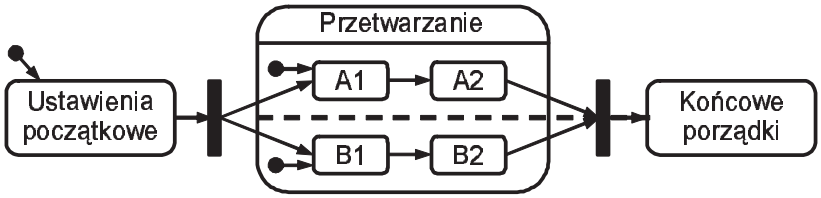
Na diagramach tranzycja jest przedstawiana jako łuk zakończony strzałką, wychodzący ze stanu źródłowego (*ang.* source) i dochodzący do stanu docelowego (*ang.* target). Tranzycje są etykietowane napisem, według następującego formatu:

predykat / akcja tranzycji.

Predykat jest boolowskim wyrażeniem operującym na nazwach zdarzeń. Wartość **1** predykatu jest warunkiem koniecznym do odpalenia tranzycji. Zdarzenie rozgłaszane w systemie nie wyzwalające żadnej tranzycji jest odrzucane i nie ma po nim żadnego śladu. Akcja tranzycji jest wykonywana, gdy tranzycja jest realizowana. Na wykonywaną akcję składają się rozgłaszane zdarzenia, które w etykiecie podaje się w postaci wyliczeniowej, np. $\{e_1, e_2, \ldots\}$.

3.3.5. Tranzycje złożone

Tranzycje złożone (*ang.* complex transition) (Booch i in., 2001), zapożyczone z teorii sieci Petriego, są takim elementem języka, który ze względu na założenie o nieprzekraczaniu granic stanu przez tranzycje proste, nie jest jeszcze wspierany w systemie autora. Jednak z uwagi na swą doniosłą i sprawdzoną rolę w modelowaniu, uważa się za konieczne przedstawienie tej techniki opisu.



Rys. 3.6. Diagram z tranzycjami złożonymi

Tranzycja złożona (rys. 3.6) wyraża związek między zbiorem stanów źródłowych, a zbiorem stanów docelowych. Związek ten może być synchronizacją sterowania i/lub jego rozwidleniem na wątki wykonywane współbieżnie. Tranzycja

złożona jest gotowa do realizacji, gdy wszystkie jej stany źródłowe są aktywne. Po odpaleniu stany źródłowe nie są już aktywne, aktywne są natomiast wszystkie stany docelowe.

Na rysunkach (np. rys. 3.6) tranzycja złożona przedstawiana jest jako pogrubiona kreska oznaczająca synchronizację, rozwidlenie lub oba te przypadki. Do kreski może dochodzić jeden lub więcej zorientowanych łuków (rys. 3.6), zaczynających się w stanach źródłowych. Od kreski może odchodzić jeden lub więcej łuków do stanów docelowych. Łuki przy takich tranzycjach mogą przecinać granice stanów (w propozycji autora – system *HiCoS* – tej możliwości nie ma). Złożonej tranzycji można również przyporządkować etykietę, umieszczaną przy pogrubionej kresce, której rola jest taka sama jak przy tranzycji prostej (punkt 3.3.4).

3.3.6. Atrybut historii

Ze względu na założoną modularność diagramów, omawiany tu atrybut historii nieco się różni do normy *UML* i jest taki, jak to przedstawiono w publikacji (Nazareth i in., 1996).

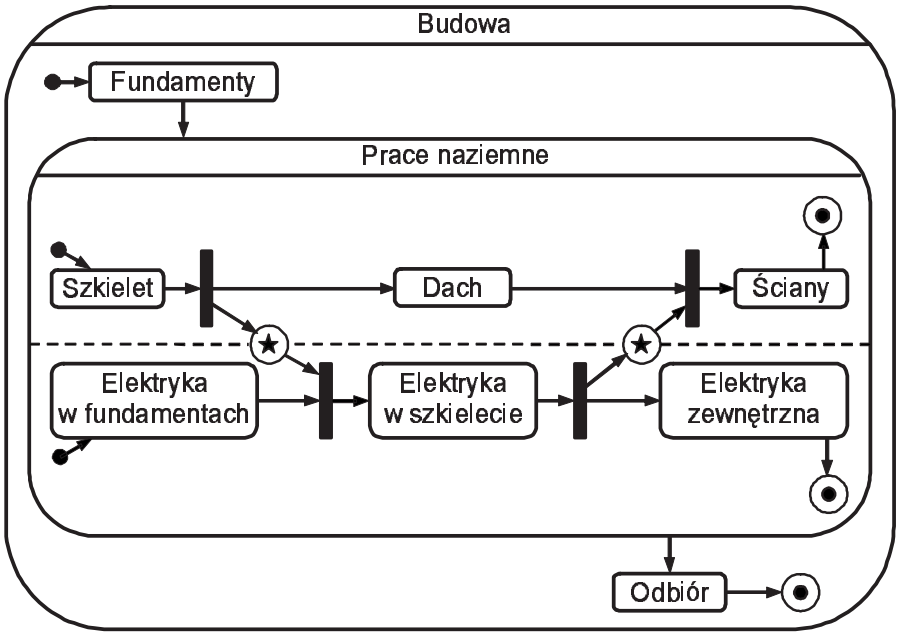
Stanowi można przypisać atrybut historii, na rysunku przedstawiany jako duża litera **H** w kółku (np. rys. 8.2). Atrybut ten stosuje się do wszystkich stanów znajdujących się w bieżącym regionie (punkt 3.3.2) i na tym poziomie hierarchii, gdzie atrybut ten został umieszczony. Taki rodzaj nadania atrybutu nazywany jest historią płytką (*ang.* shallow history), a o stanie z takim atrybutem można powiedzieć, że jest stanem z pamięcią. Nadanie grupie stanów atrybutu historii oznacza ponowne wznowienie działania automatu od stanu ostatnio aktywnego. Jeżeli automat jest uruchamiany po raz pierwszy, to wówczas działanie rozpocznie się od stanu wskazanego jako stan początkowy. Istnieje możliwość nadania atrybutu historii nie tylko stanom na wskazanym poziomie wystąpienia graficznego znacznika historii, ale też i wszystkim podległym im stanom. Ten rodzaj atrybutu nazywany jest historią głęboką (*ang.* deep history), a na rysunku oznaczany jest symbolem **H***, również umieszczonym w kółku (np. rys. 7.4). Należy zauważyć, że jednoczesne stosowanie atrybutu historii razem ze stanem końcowym (punkt 3.3.2) wzajemnie się wyklucza. Zaistnienie takiej sytuacji powoduje usunięcie właściwości pamięci, ponieważ stanowi końcowemu nadano wyższy priorytet.

3.3.7. Stany synchronizujące

Ten rodzaj stanów specjalnego przeznaczenia, z powodu niestosowania tranzycji złożonych, również nie jest wykorzystywany w zrealizowanym systemie autora, lecz ze względu na interesujące właściwości zostanie tu przedstawiony jako proponowany temat dalszych prac.

Stany synchronizujące służą do zsynchronizowania wykonywania współbieżnych regionów. Stosuje się je w połączeniu z tranzycjami synchronizującymi lub rozwidlającymi w celu zapewnienia, że np. jeden region wejdzie do określonego stanu, zaraz po tym jak inny region opuści pewien swój określony stan. Taka sytuacja jest przykładowo przedstawiona na rysunku 3.7, gdzie, jak to widać, zanim będzie

można położyć okablowanie w szkielecie budynku, budowa szkieletu musi być zakończona, co na diagramie (rys. 3.7) oznaczone jest poprzez aktywność pierwszego stanu synchronizującego. Stan synchronizujący przedstawiany jest jako gwiazdka w kółku i umieszczany jest, jeśli to jest możliwe, na granicy synchronizowanych regionów.



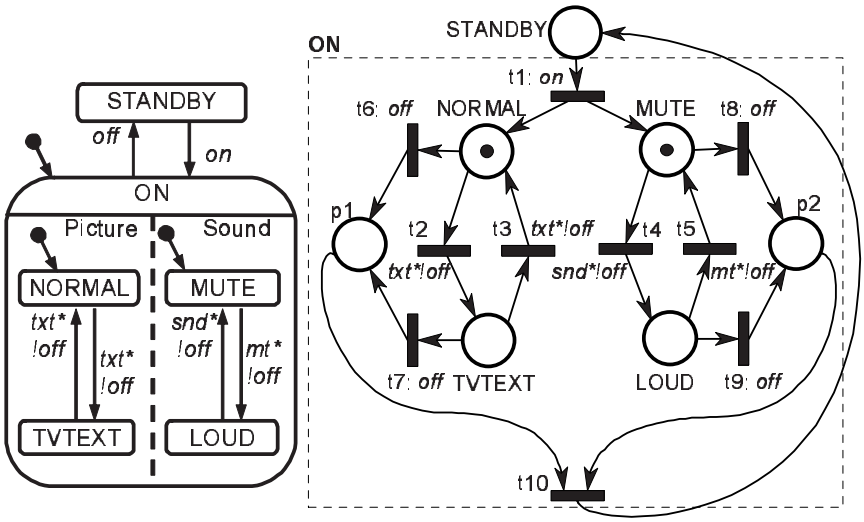
Rys. 3.7. Przykładowe zastosowanie stanów synchronizujących

3.4. Diagramy statechart a hierarchiczne sieci Petriego

Prezentując diagramy statechart bardzo często zadawane jest pytanie, jaka jest zasadnicza różnica między diagramami a hierarchiczną siecią Petriego. Aby na to pytanie odpowiedzieć należy najpierw uściślić z jakiego rodzaju hierarchiczną siecią Petriego diagramy są porównywane. Jak to zostało zaznaczone w rozdziale 2.4 istnieje wiele odmian sieci, co do których stosowany jest przymiotnik hierarchiczne. Kolejną kwestią jest kryterium porównawcze. Można zestawiać ze sobą przeróżne cechy obu modeli, takie jak na przykład właściwość historii czy istnienie stanu końcowego, lecz wydaje się, że cechą o pierwszorzędnym znaczeniu przy takim porównaniu jest reprezentacja hierarchii w modelowaniu. W diagramach statechart obok stanów wyższych poziomów hierarchii zwanych abstrakcyjnymi, występują również tranzycje wyższych poziomów związane z takimi stanami, zwane tranzycjami abstrakcyjnymi. Realizacja takiej tranzycji może mieć charakter wywłaszczeniowy (podrozdział 4.12), co w opinii autora stanowi istotę pojęcia hierarchii behawioralnej. Przy tak przyjętym kryterium porównawczym różnica

między diagramami statechart, a na przykład sieciami typu *PetriCharts* (Holvoet i Verbaeten, 1995) sprowadza się do cech drugorzędnych. Natomiast w przypadku sieci Petriego typu makro, omówionych w rozdziale 2.4, różnica ta jest o wiele bardziej istotna. W sieciach tego rodzaju brak jest tranzycji abstrakcyjnych, co znacznie utrudnia modelowanie polegające na podejściu od ogółu do szczegółu.

Ze względu na istniejące liczne opracowania (m.in.: Peterson, 1981; Reisig, 1988; Murata, 1989; Adamski, 1990; 1998; Kozłowski, 1993; Banaszak i in., 1993; Pastor i in., 1994; Kozłowski i in., 1995; Biliński, 1996; Esser, 1996; Wolański, 1998*b*) dotyczące analizy i syntezy sieci Petriego jak i na prowadzone badania (np.: Mirkowski i Skowroński, 1998; Skowroński, 1999; 2000), w których sieć Petriego jest formalnym modelem w syntezie systemowej, autor podjął próbę opracowania przejścia od modelu statechart do klasycznych sieci Petriego (Łabiak, 1999). Dzięki takiemu przejściu możliwe jest projektowanie na zasadzie od ogółu do szczegółu, gdzie końcową postacią projektu jest model sieci Petriego ze wszelkimi jej zaletami.



Rys. 3.8. Diagram statechart i równoważna mu sieć Petriego

Opracowane przejście polega na przyporządkowaniu każdemu nieabstrakcyjnemu (podstawowemu) stanowi z diagramu odpowiadającego mu miejsca w sieci Petriego. Ilustruje to rysunek 3.8. Następnie do każdego miejsca odpowiadającego stanowi, z którego tranzycja abstrakcyjna może wywłaszczyć sterowanie (w przykładzie stany: *NORMAL*, *TVTEXT*, *MUTE*, *LOUD*) należy dołączyć tranzycje (na rysunku 3.8 oznaczone kolejno jako t_6, t_7, t_8, t_9), których rolą jest zabranie żetonu na wypadek realizacji tranzycji abstrakcyjnej. Odebrane żetony są następnie deponowane w pomocniczych miejscach p_1 i p_2 – po jednym dla każdego z podległych automatów sekwencyjnych (odpowiednio *Picture* i *Sound*). Dalej poprzez tranzycję pomocniczą t_{10}, żeton jest kierowany do miejsca *STANDBY*, co kończy

wywłaszczeniowe przekazanie sterowania. Na realizację abstrakcyjnej tranzycji ze stanu *STANDBY* do stanu *ON* składają się tranzycje t_6, t_7, t_8, t_9 i t_{10} oraz miejsca p_1 i p_2.

Przedstawione przejście skupia się na modelowaniu najistotniejszych elementów diagramów statechart: stanów i tranzycji w tym tranzycji abstrakcyjnych. Pominięto zagadnienia stanów z pamięcią (atrybut historii), stanu końcowego, akcji związanych ze zdarzeniami. Porównując realizacje układowe, tego samego zachowania zamodelowanego diagramami statechart, systemu *HiCoS* z bezpośrednią implementację sieci otrzymanych w wyniku opisywanego przejścia można stwierdzić, że układowa realizacja tranzycji wywłaszczeniowej z otrzymanej sieci Petriego będzie wolniejsza (wywłaszczenia sterowania w przykładzie zawsze wymaga dwu tranzycji, np. t_6 i t_{10}) oraz, że realizacja podległego automatu sekwencyjnego pochłania co najmniej jeden dodatkowy przerzutnik, związany z implementacją miejsca pomocniczego. W przykładzie z każdym podległym automatem związane jest jedno miejsce pomocnicze, ale można sobie wyobrazić kolejne dodatkowe przerzutniki implementowane dla potrzeb wywłaszczenia sterowania do stanu, który w hierarchii, względem automatu, znajduje się wyżej niż przykładowy *STANDBY*.

3.5. Podsumowanie

W rozdziale opisano zdobywającą coraz większą popularność technologię *UML* oraz zaproponowano jej wykorzystanie w procesie projektowania układów cyfrowych – co jest nowym jeszcze nie w pełni zbadanym obszarem zastosowań tej technologii. W tym kontekście przedstawiono innowacyjną własną metodologię systemu *HiCoS*. Scharakteryzowano język *SpecChart* będący elementem metodyki stosowanej do projektowania cyfrowych systemów osadzonych. Przedstawiono postać graficzną diagramów statechart zgodną z językiem *UML*, definiując ich podzbiór wykorzystany do specyfikowania zachowania binarnych modularnych sterowników cyfrowych oraz wskazano możliwości dalszych zastosowań. Na zakończenie opisano związki diagramów statechart z konkurencyjną metodyką – hierarchicznymi sieciami Petriego.

Rozdział 4

CHARAKTERYSTYKA DIAGRAMÓW STATECHART – PRZEGLĄD WYBRANYCH ZAGADNIEŃ

Podstawowym celem, który przyświecał opracowywaniu i rozwojowi formalizmowi diagramów statechart (Harel, 1987), było specyfikowanie zachowania systemów reaktywnych (Harel i Politi, 1998). Powstały model stanowi naturalne rozwinięcie pojęcia grafu stanów automatu skończonego, wzbogaconego o współbieżność i hierarchię. W pracy (Harel i Politi, 1998) diagramy w sposób bardzo wymowny zostały scharakteryzowane następującym równaniem:

$$statechart = diagram\ stanów + głębia + ortogonalność + rozgłaszanie$$

Możliwość wyrażania stanów modelowanego zachowania w sposób hierarchiczny oraz wyraziste przedstawianie współbieżności przyczyniły się do dużego zainteresowania naukowców. Z kolei pewne kwestie składniowe i semantyczne, które w pierwszych publikacjach nie zostały precyzyjnie opisane, przyczyniły się do badań nad diagramami, które zaowocowały opracowaniem kilkunastu odmian diagramów (von der Beeck, 1994).

Niniejszy rozdział stanowi fragment dyskusji nad wybranymi aspektami składni i semantyki diagramów, szeroko toczonej w światowej literaturze przedmiotu oraz jest umiejscowieniem założeń przyjętych przez autora na tle dokonań innych badaczy. Przykłady oraz wybrane zagadnienia zostały opracowane na podstawie publikacji (von der Beeck, 1994), a każdy z podrozdziałów kończy się odwołaniem do autorskiego systemu projektowania *HiCoS*.

4.1. Hipoteza doskonałej synchroniczności

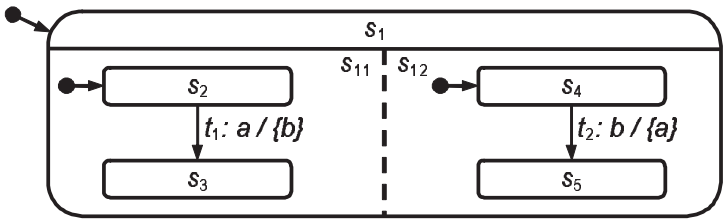
Hipoteza doskonałej synchroniczności (*ang.* perfect synchrony hypothesis), omawiana jako pierwsze zagadnienie, dotyczy chyba najbardziej fundamentalnej kwestii systemów reaktywnych, mianowicie sposobu w jaki system reaguje na pobudzenia. Określenie „perfect synchrony hypothesis" po raz pierwszy zostało zaproponowane przez twórców języka *Esterel* (Berry i Gonthier, 1992) i oznacza, że odpowiedź układu na pobudzenie powinna występować w tym samym momencie czasu co pobudzenie. Takie właśnie założenie zostało zaproponowane przez twórcę diagramów w pierwszej poświęconej wyłącznie na ich temat publikacji (Harel, 1987).

W rzeczywistości jednak, odpowiedź układu nigdy nie występuje jednocześnie z pobudzeniem. Z tego powodu ścisły wymóg jednoznaczności może być zredukowany do założenia, że układ nie będzie pobudzany następnym zdarzeniem wejściowym, zanim nie wypracuje odpowiedzi na poprzednie zdarzenie. Jednocześnie, dla obserwatora z zewnątrz, jest to postrzegane jako niepodzielna chwila w działaniu układu. Takie założenie upraszczające nazywane jest właśnie hipotezą doskonałej synchroniczności.

W opracowanym systemie, ze względu na ułatwienia realizacyjne, przyjęto, że w implementacji sprzętowej układ jest układem synchronicznym, którego działanie jest synchronizowane sygnałem zegarowym, w rytm którego układ zmienia swój stan i wypracowuje odpowiedź. Jeżeli generowane odpowiedzi wpływają na zmianę stanu układu, na zasadzie sprzężenia zwrotnego, to wówczas wypracowanie całej odpowiedzi układu na określone pobudzenie, zwane krokiem, może zająć kilka taktów sygnału zegarowego, przy czym przyjmuje się, że w tym czasie do układu nie docierają kolejne pobudzenia. Stąd wniosek, że częstotliwość sygnału taktującego powinna być ustalona na podstawie interwałów czasowych pomiędzy kolejno przychodzącymi pobudzeniami (o ile jest to możliwe), tak aby czas odpowiedzi układu był krótszy od odstępów czasowych między kolejnymi stymulacjami.

4.2. Samoistna realizacja tranzycji, przyczynowość

Określenie samoistna realizacja tranzycji (*ang.* transition self-triggering) odnosi się do sytuacji, gdy tranzycja jest wzbudzana przez zdarzenie (lub zdarzenia), które nie jest jednoznacznym wynikiem innego zdarzenia dochodzącego ze świata zewnętrznego. Rysunek 4.1 przedstawia opisywaną sytuację.



Rys. 4.1. Możliwa samoistna realizacja tranzycji

Tranzycja t_1 jest wzbudzana zdarzeniem a. Jej realizacja powoduje wygenerowanie zdarzenia b, które z kolei skutkuje odpaleniem tranzycji t_2, a ta następnie spowoduje ponowne wygenerowanie zdarzenia a. Pamiętając o założeniu jednoczesności przyczyny i skutku (lub jak w tym przypadku skutków, podrozdział 4.1), wygenerowane zostaną dwa zdarzenia a i b. W wyniku przyjętego założenia powstaje sytuacja, że w tym samym momencie czasu są zrealizowane dwie tranzycje (t_1 i t_2) i jednocześnie pojawiają się zdarzenia a i b, przy czym zdarzenie a będące hipotetycznie uznawane za pierwszą przyczynę, nie musi pochodzić ze świata zewnętrznego, gdyż jest generowane przez realizację tranzycji t_2. W tym przypadku, na podstawie wygenerowanych zdarzeń, można by stwierdzić, że tranzycja t_1

do swej realizacji nie potrzebuje wystąpienia zdarzenia przychodzącego ze świata zewnętrznego (realizuje się samoistnie).

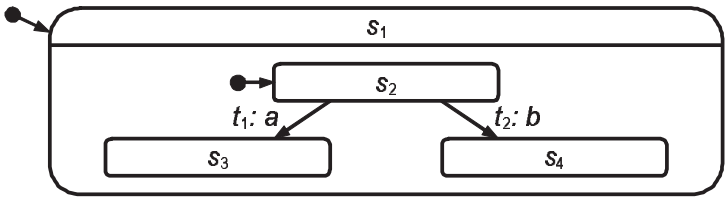
O semantyce diagramów mówi się, że jest przyczynowa (*ang.* causal), gdy dla każdej zrealizowanej tranzycji pojawiło się zdarzenie zewnętrzne, które w sposób pośredni lub bezpośredni jednoznacznie stanowi jedyną jej przyczynę. Innymi słowy, realizacji każdej tranzycji można jednoznacznie przypisać tylko jeden ciąg zdarzeń i realizacji tranzycji generujących kolejne zdarzenia w ciągu, które prowadzą do zdarzenia rozpoczynającego ów ciąg przyczynowo-skutkowy, będącego zdarzeniem zewnętrznym. Dodatkowo zakłada się, że nie mogą wystąpić cykle. Taki pogląd po raz pierwszy został przedstawiony w pracy (Huizing i Gerth, 1992). Jednak opisywana tam semantyka nie jest przyczynowa, gdyż wszystkie zdarzenia generowane w trakcie całego kroku są uznawane za już istniejące na samym jego początku.

Zestawiając opisywane w tym punkcie pojęcia, można powiedzieć, że dopuszczenie samoistnie realizujących się tranzycji powoduje, że taka semantyka nie jest przyczynowa.

W systemie *HiCoS* nie ma możliwości wystąpienia samoistnie realizujących się tranzycji, gdyż realizacja tranzycji w danym momencie dyskretnego czasu, powoduje wygenerowanie zdarzenia lub zdarzeń, które są dostępne dla układu dopiero podczas następnego najbliższego taktu zegara. W wyniku takiego założenia powstały system jest systemem przyczynowym, przez co opis zachowania odbywa się w sposób bardziej intuicyjny i jednoznaczny.

4.3. Wzbudzanie zdarzeniem zanegowanym

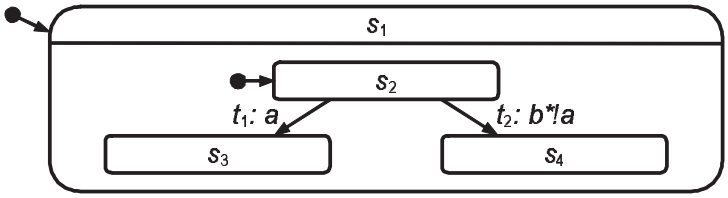
Zdarzenia na diagramie reprezentowane są przez swoją nazwę. Umieszczenie nazwy zdarzenia w części wyzwalającej tranzycji oznacza, że aby tranzycja mogła być zrealizowana konieczne jest wystąpienie tego zdarzenia. Rysunek 4.2 przedstawia diagram niedeterministyczny z tranzycjami t_1 i t_2, których realizacja uzależniona jest od zdarzeń, odpowiednio a i b.



Rys. 4.2. Diagram niedeterministyczny

Jednoczesne pojawienie się obydwu zdarzeń powoduje, że gotowymi do realizacji są obie tranzycje i stąd aktywnymi stanami stałyby się stany s_3 i s_4. Powstanie takiej sytuacji kłóci się z innym wymogiem działania diagramów, który mówi, że spośród wszystkich stanów bezpośrednio podległych stanowi o typie *OR*, tylko jeden z nich może być aktywny (podrozdział 6.1 definicja 6.4). Sprzętowa

realizacja układu z wykorzystaniem takiej specyfikacji zachowania jest niedopuszczalna ze względu na jawny niedeterminizm funkcjonowania modularnego układu. Wystąpienie takiej sytuacji jest nazywane konfliktem między tranzycjami (Harel i Naamad, 1996). Do rozwiązywania konfliktów między tranzycjami można stosować zdarzenia zanegowane (*ang.* negated event), które zamieszczone w warunku na pobudzenie tranzycji oznaczają, że rozpatrywane zdarzenia, w formie afirmacji, w danym momencie czasu nie występują. Na diagramie takie zdarzenie poprzedzone jest operatorem negacji (rys. 4.3).



Rys. 4.3. Diagram deterministyczny ze wzbudzaniem zdarzeniem zanegowanym

Rysunek 4.3 ilustruje przykładowe rozwiązanie konfliktu między tranzycjami przedstawionymi na rysunku 4.2 (patrz również podrozdział 7.3). W zmodyfikowanej wersji tranzycje t_1 i t_2 nigdy nie będą gotowe do realizacji w tym samym momencie czasu, gdyż predykaty na nie nałożone, z pomocą zdarzenia zanegowanego ($!a$), wzajemnie się wykluczają.

W systemie *HiCoS* zaadoptowano pojęcie zdarzenia zanegowanego, które jako element wyrażenia logicznego, będącego predykatem, stanowi główny środek rozwiązywania konfliktów między tranzycjami (podrozdział 7.3).

4.4. Sprzeczność skutku realizacji tranzycji z jej przyczyną

Założenie o jednoczesności występowania skutku i przyczyny (czyli hipoteza doskonałej synchroniczności, podrozdział 4.1) oraz operowanie pojęciem zdarzenia zanegowanego może powodować sytuacje, w których w najprostszym przypadku (rys. 4.4) zdarzenie wygenerowane w wyniku realizacji tranzycji jest w sprzeczności z warunkiem jej wzbudzenia. Sytuacja taka ma miejsce, gdy podczas wykonywania kroku, czyli w łańcuchu kolejno realizowanych tranzycji i generowanych zdarzeń, powstałe zdarzenie również występuje w części predykatowej w postaci zanegowanej.

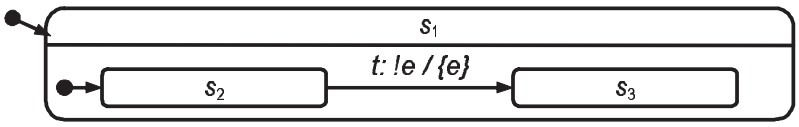
Rysunek 4.4 przedstawia przypadek najbardziej trywialny. Warunkiem realizacji tranzycji t jest niewystępowanie zdarzenia e. Jej realizacja z kolei, powoduje wygenerowanie zdarzenia e, co przy założeniu o jednoczesnym występowaniu skutku i przyczyny prowadzi do sprzeczności, polegającej na jednoczesnym wystąpieniu i niewystąpieniu zdarzenia e.

W celu zapobiegania występowaniu opisywanych niezgodności, w pracy (Pnueli i Shalev, 1991), zaproponowano dwie strategie dopuszczenia tranzycji do realizacji,

różniące się interpretacją zdarzeń zanegowanych występujących w części wzbudzającej tranzycji:

- Interpretacja zdarzenia zanegowanego zależna jest wyłącznie od przeszłości. W tym przypadku tranzycja zawierająca zdarzenie zanegowane w swej części wzbudzającej jest wykonywana w obrębie danego kroku, jeśli w dotychczas wykonanych mikrokrokach (mikrokrok rozumiany jest jako zbiór jednocześnie realizowanych tranzycji) nie ma takiej tranzycji, która by generowała rozpatrywane zdarzenie zanegowane, czyli oznacza to, że do rozpatrywanego momentu czasu zdarzenie rzeczywiście nie wystąpiło. Mówiąc inaczej, realizacja rozpatrywanej tranzycji nie zależy od innych tranzycji wykonywanych w obrębie bieżącego mikrokroku, ani od tranzycji wykonywanych w późniejszych mikrokrokach, składających się na cały krok. Taki warunek zwany jest spójnością lokalną (*ang.* local consistency).

- Interpretacja zdarzenia zanegowanego zależna jest zarówno od przeszłości i przyszłości.
  W tym przypadku zanegowanie zdarzenia nie tylko oznacza, że wśród dotychczas wykonanych mikrokroków nie ma tranzycji generującej rozpatrywane zdarzenie zanegowane, ale również, że takich tranzycji nie ma w bieżącym i w następnych mikrokrokach całego kroku. Innymi słowy, tranzycja zawierająca w swej części wzbudzającej zdarzenie zanegowane, zostanie zrealizowana w obrębie kroku wtedy i tylko wtedy, gdy w żadnym mikrokroku nie ma tranzycji generującej rozpatrywane zdarzenie. Tak sformułowany warunek zwany jest spójnością globalną (*ang.* global consistency). Dla przykładowej tranzycji t (rys. 4.4) opisywany warunek nie jest spełniony i tranzycja ta nie zostałaby zrealizowana.



Rys. 4.4. Sprzeczność między skutkiem a przyczyną realizacji tranzycji

Semantyka wymagająca zgodności lokalnej w istotnym stopniu polega na ustaleniu kolejności mikrokroków w kroku, co w przypadku zgodności globalnej nie ma aż tak dużego znaczenia. Tabela 4.1, na przykładzie dwu sekwencji mikrokroków, przedstawia omawiane zagadnienie w sposób szczegółowy.

W tabeli 4.1 zamieszczono sekwencje dwu mikrokroków, różniące się kolejnością realizacji tych samych tranzycji. W przypadku pierwszym zdarzenie a jest wygenerowane przed tranzycją, która w swej części wzbudzającej posiada zanegowane zdarzenie ($!a$), a w przypadku drugim jest odwrotnie. W obu przypadkach zgodności tranzycja wymieniona jako druga nie zostanie dopuszczona do realizacji. W przypadku drugim, spójność lokalna dopuszcza jednak realizację całej wymienionej sekwencji mikrokroków, gdyż tranzycja zależna od zdarzenia zanegowanego

Tab. 4.1. Sekwencje mikrokroków oraz ich spójności

Nr	Sekwencja mikrokroków	Spójność lokalna	Spójność globalna	system *HiCoS*
1.	$\ldots b/a, \ldots !a/\ldots, \ldots$	–	–	+
2.	$\ldots !a/\ldots, b/a, \ldots$	+	–	+

($!a$), realizowana jest przed tranzycją generującą zdarzenie a. W przypadku spójności globalnej kolejność tranzycji, jak widać, nie ma znaczenia i żadna z zamieszczonych sekwencji w pełni nie będzie zrealizowana.

Wydaje się, że semantyka wspierająca spójność lokalną jest bardziej intuicyjna niż semantyka operująca spójnością globalną. Zdaniem autora jest to związane z faktem, że w przypadku spójności lokalnej, na skutek uszeregowania mikrokroków, ma miejsce powiązanie przyczyn ze skutkami, które je wywołują. Dodatkowo na niekorzyść zgodności globalnej wpływa fakt, że jest ona w sprzeczności z omawianym wcześniej pojęciem przyczynowości.

W systemie *HiCoS* opisywana problematyka spójności zdarzeń nie występuje, gdyż jak to już wcześniej opisano, nie przyjęto założenia o doskonałej synchroniczności, realizacje mikrokroków powiązane są z kolejnymi momentami dyskretnego czasu, a generowane w trakcie ich realizacji zdarzenia dostępne są dla układu przy nadejściu kolejnego taktu sygnału zegarowego (szczegóły założeń implementacyjnych przedstawiono w punktach 7.2.1 i 7.2.2).

4.5. Tranzycje przekraczające granice stanów

Tranzycje przekraczające granice stanów (*ang.* inter-level transition) łączą ze sobą stany, które nie są bezpośrednimi potomkami swego najniższego wspólnego przodka (podrozdział 6.1 definicja 6.4 punkt 2). W drzewie hierarchii dwa stany połączone taką tranzycją zazwyczaj znajdują się na różnych poziomach. Omówienie problemów spowodowanych stosowaniem takich tranzycji znajduje się w podrozdziale 4.7.

4.6. Odwołania do stanów

Niektóre z proponowanych wariantów diagramów oferują warunkowe wykonanie tranzycji, uzależnione od aktywności pewnego innego stanu, znajdującego się w innym komponencie równoległym. W pierwotnej wersji diagramów, przedstawionej w referacie (Harel, 1987), zależność ta została zrealizowana poprzez umieszczenie w części predykatowej tranzycji operatora *in(stan)*, który poprzez odwołanie do stanu (*ang.* state reference), zwraca wartość *true*, gdy stan jest aktywny.

W systemie autora, odwołanie do stanu może być zrealizowane jedynie poprzez przypisanie stanowi tzw. akcji statycznej. Realizuje się to poprzez umieszczenie w stanie na diagramie nazwy zdarzenia poprzedzonego słowem kluczowym *do* (np.:

do / e). Wówczas takie zdarzenie jest rozgłaszane przy każdym takcie zegara tak długo, jak długo stan jest aktywny i może być częścią predykatu, funkcjonalnie pełniącą rolę odwołania do stanu.

4.7. Semantyka modułowa, samoistne wywłaszczenie sterowania

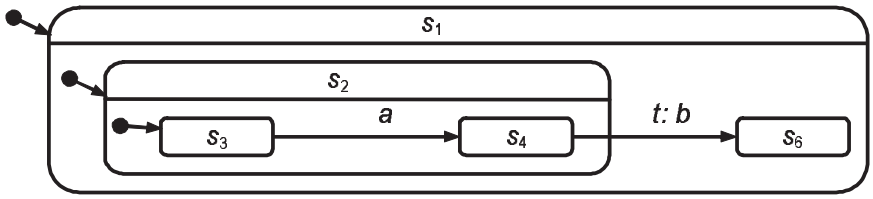
O funkcjach znaczeniowych konstrukcji gramatycznych pewnego języka można powiedzieć, że jego semantyka jest modułowa (*ang.* compositional semantics), gdy tworzone obiekty w tym języku, są definiowane wyłącznie w kategoriach jego elementów składowych, tzn. że tworzony obiekt nie jest definiowany poprzez odwoływanie się do szczegółów konstrukcyjnych jego komponentów składowych niższego poziomu (von der Beeck, 1994). Pierwsza semantyka diagramów zaproponowana w pracy (Harel, 1987), a następnie formalnie ujęta w publikacji (Harel i Politi, 1998), charakteryzowała się brakiem modułowości.

Główną korzyścią płynącą ze stosowania języków o semantyce modułowej, jest możliwość definiowania i operowania obiektami charakteryzującymi się dużą złożonością, wszakże jednak pod warunkiem, że opisywane przez projektanta obiekty są postrzegane modułowo. Kolejną zaletą modułowości jest łatwość weryfikacji, gdyż właściwości złożonego komponentu mogą zostać zbadane w kategoriach właściwości jego komponentów składowych, bez potrzeby wnikania w ich wewnętrzną budowę. Aby język posiadał takie cechy, obiekty którymi się w danym języku operuje, powinny być postrzegane poprzez jak najmniejszą ilość atrybutów, a jego składnia powinna zapewniać środki do ukrywania jak największej liczby (lub nawet wszystkich) wewnętrznych szczegółów. Jeśli język posiada takie właściwości, to o jego semantyce można powiedzieć, że jest abstrakcyjna (von der Beeck, 1994).

W przypadku diagramów statechart, według poglądów zamieszczonych w publikacjach (Classen, 1993) i (Maraninchi, 1991), tranzycje przekraczające granice stanów, odwołania do stanów oraz mechanizm pamięci (inaczej nazywany atrybutem historii) są tymi elementami, które zaprzeczają pojęciu modułowości. Zdaniem autora, do grupy wymienionych konstrukcji językowych należy również zaliczyć mechanizm globalnego rozgłaszania zdarzeń oraz stan końcowy, ze swą właściwością blokowania realizacji tranzycji wyższych poziomów hierarchii.

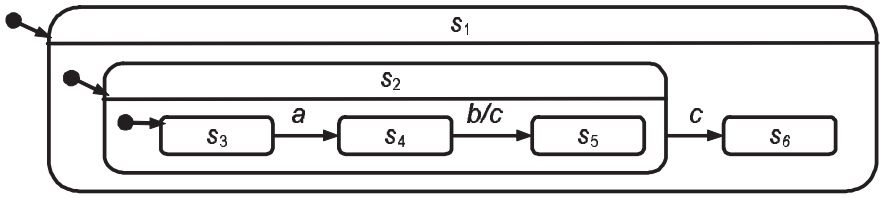
Spośród wymienionych powyżej elementów diagramów, tranzycje przekraczające granice stanów najsilniej ograniczają korzyści płynące z modułowości. W graficznej postaci diagramów linie krzyżujące się z brzegami krągłokątów znacząco utrudniają zrozumienie modelowanego zachowania. W związku z tym, w niektórych semantykach zabroniono stosowania tego mechanizmu, a diagramy rysowane z uwzględnieniem takiego ograniczenia, noszą nazwę modularnych. Podobne podejście zostało przyjęte przez autora. W pracy (Maraninchi, 1992) opisano koncepcję samoistnego wywłaszczenia sterowania ze stanu (*ang.* self- termination), która to pozwala na unikanie stosowania tranzycji przekraczających granice stanu.

Rysunek 4.5 prezentuje przykładową tranzycję t, która przecina granicę stanu s_2. Rysunek 4.6 przedstawia diagram modularny, bez takich tranzycji, funkcjonalnie równoważny poprzedniemu. W jednym i drugim przypadku, w momencie gdy aktywnym jest stan s_4, wystąpienie zdarzenia b powoduje wywłaszczenie sterowa-



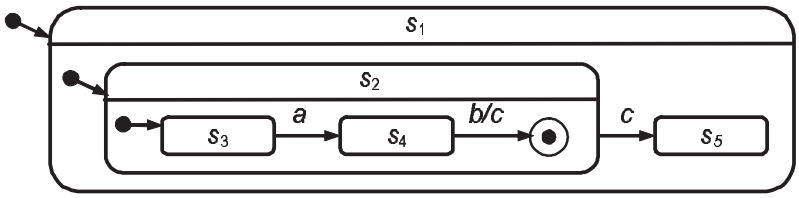
Rys. 4.5. Tranzycja przekraczająca granicę stanu

nia ze stanu s_2 i aktywację stanu s_6. W pierwszym przypadku efekt został uzyskany poprzez realizację tranzycji przecinającej granicę stanu, natomiast w przypadku drugim poprzez posłużenie się przejściem do dodatkowego stanu (s_5) i wygenerowanie pomocniczego zdarzenia c. W przypadku drugim realizacja tranzycji niższego poziomu, poprzez wystąpienie zdarzenia c, powoduje realizację tranzycji z poziomu wyższego, co skutkuje samoistnym wywłaszczeniem sterowania ze stanu s_2. Na cały krok składają się dwa mikrokroki, a w systemie *HiCoS* dwa takty zegara.



Rys. 4.6. Samoistne wywłaszczenie sterowania

Rysunek 4.7 przedstawia autorską propozycję dalszej modyfikacji diagramu. W tej wersji zastosowano, zamiast stanu pomocniczego s_5, stan końcowy będący regularnym elementem semantyki proponowanej przez autora. Diagram w tej wersji charakteryzuje się tym, że samoistne wywłaszczenie sterowania ze stanu s_2 wystąpi jedynie w sytuacji aktywności stanu s_4 i jednoczesnego pojawienia się zdarzenia b. Wadą wersji z rysunku 4.6 jest to, że sterowanie mogłoby być zabrane ze stanu s_2, przy aktywności któregokolwiek ze stanów podległych (s_3, s_4, s_5) i przy jednoczesnym wystąpieniu zdarzenia c (nawet bez udziału zdarzenia b), które może, na przykład, nadejść przypadkowo ze świata zewnętrznego lub dowolnej innej części diagramu.



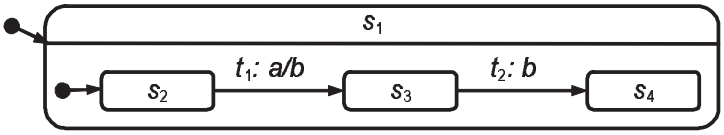
Rys. 4.7. Samoistne wywłaszczenie sterowania – wariant ze stanem końcowym

W opinii autora, samoistne wywłaszczenie sterowania w wersji ze stanem końcowym, w sposób bardziej rygorystyczny modeluje przypadek tranzycji przekraczającej granice stanu.

4.8. Stany przejściowe

Stan przejściowy (*ang.* instantaneous state) jest to stan, który rozpatrywany w kontekście kroku jednocześnie jest aktywowany i pozbawiany aktywności. W pierwszej publikacji nieformalnie prezentującej statecharty (Harel, 1987), kwestia ta została pominięta, natomiast w następnych referatach bardziej formalnie ujmujących diagramy (m.in. Harel i Politi, 1998), występowanie stanów przejściowych jest zabronione. Rysunek 4.8 przedstawia przykład możliwego stanu przejściowego. Realizacja tranzycji t_1 powoduje aktywację stanu s_3 i wygenerowanie zdarzenia b wzbudzającego tranzycję t_2. W takiej sytuacji powstaje pytanie, istotne zwłaszcza w kontekście hipotezy o doskonałej synchroniczności, czy w tym samym momencie czasu ma równocześnie zostać odpalona gotowa do realizacji tranzycja t_2, która wywłaszczy sterowanie ze stanu s_3 powodując, że będzie to stan przejściowy, czy też nie.

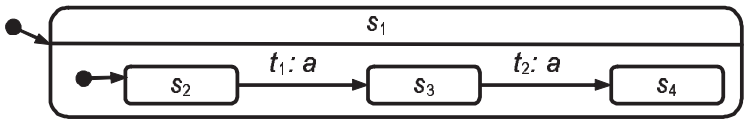
W publikacji (Harel i Politi, 1998) przyjęto, że rozpatrywane wyżej stany przejściowe nie mogą występować. Implikacją takiego założenia jest fakt, że w jednym kroku mogą zostać zrealizowane tylko tranzycje znajdujące się w osobnych komponentach współbieżnych. Dalszą konsekwencją tego założenia jest to, że liczba realizowanych tranzycji w tym samym momencie czasu (czyli w jednym kroku) jest ograniczona poprzez skończoną liczbę komponentów współbieżnych.



Rys. 4.8. Stan s_3 jako potencjalny stan przejściowy

Przykład stanu s_3 (rys. 4.8) ujawnia, że obecność stanu przejściowego jest w sprzeczności z podstawowym założeniem (zawartym w Harel, 1987) co do stanu o typie *OR*, mówiącym, że w jednym momencie czasu spośród stanów podległych bezpośrednio stanowi o typie *OR*, aktywnym może być co najwyżej tylko jeden stan. W pracy (Peron, 1993), prezentującej wariant diagramów zezwalający na obecność stanu przejściowego, zaproponowano złagodzenie warunku na aktywność stanów podległych stanowi typu *OR*, mianowicie dopuszcza się dodatkową aktywność stanu przejściowego pod warunkiem, że w mikrokroku stan przejściowy nie jest jednocześnie aktywny z żadnym innym stanem podległym. Przykładowy stan s_3 spełnia ten warunek.

Przyjmując występowanie stanu przejściowego za zgodne z przyjętymi założeniami, należy rozpatrzyć kwestię interpretacji wystąpienia samego zdarzenia, powodującego pojawienie się stanu przejściowego (rys. 4.9).



Rys. 4.9. Ile wystąpień zdarzeń a jest koniecznych do przejścia ze stanu s_2 do stanu s_4?

Rozpatrując przedstawioną sytuację (rys. 4.9), należy ustalić ile wystąpień zdarzenia a jest koniecznych aby sterowanie przeszło ze stanu s_2 do stanu s_4. Przyjmując, że występowanie stanów przejściowych jest zabronione, wówczas do takiego przejścia będą konieczne dwa wystąpienia zdarzenia a. Gdy natomiast obecność stanu przejściowego jest dopuszczona, dwie możliwe sytuacje mogłyby mieć miejsce:

- Jedno wystąpienie zdarzenia a jest wystarczające.
  Przy takiej interpretacji zdarzenie nie jest traktowane, jako coś co jest „konsumowane" po realizacji tranzycji. Wadą jest możliwość wystąpienia nieskończonej sekwencji realizowanych tranzycji.

- Dwa wystąpienia zdarzeń a są konieczne.
  W tym przypadku realizacja tranzycji powoduje „skonsumowanie" zdarzenia i do realizacji drugiej tranzycji t_2 (rys. 4.9), konieczne jest ponowne wystąpienie zdarzenia a. Zaletą takiego podejścia jest modelowania systemów licznikowych, które muszą posiadać zdolność rozróżniania kolejnych wystąpień zdarzeń.

Istnieje jeszcze trzecia możliwość, polegająca na jawnym określaniu właściwości konkretnego zdarzenia, która to jedynie w pracy (Classen, 1993) została rozpatrywana, aczkolwiek w pracy (von der Beeck, 1994) autor uważa to podejście za zalecane.

W proponowanym systemie *HiCoS*, występowanie stanu przejściowego jest dopuszczalne, przy czym aktywacja i deaktywacja stanu przejściowego odbywają się podczas kolejnych taktów sygnału zegarowego. Tak więc, zgodnie z przyjętymi przez autora założeniami opisywanymi w podrozdziale 4.1, na krok może składać się kilka kolejnych taktów zegara, powodujących wykonanie kolejnych mikrokroków. Zdarzenia nie są traktowane jako jednostki „konsumowane" przez tranzycję, a raczej jako coś, co trwa. W systemie *HiCoS*, w przykładzie z rysunku 4.9, do realizacji tranzycji t_1 i t_2 w jednym kroku potrzeba dwóch taktów zegara i występowania zdarzenia, reprezentowanego przez sygnał, trwającego przez cały ten okres. Jak widać, założenia autorskiej implementacji są wynikiem możliwości oferowanych przez sprzęt, a dokładnie przez układy cyfrowe, w których łatwo można zapamiętać dowolnie długo wystąpienie każdego zdarzenia.

4.9. Dostępność zdarzenia

Pojęcie dostępności zdarzenia (*ang.* durability of events) odnosi się do kwestii mówiącej o tym, jak długo zdarzenie, które pojawi się w układzie, może wpływać na jego działanie. Pojęcie to ściśle wiąże się z omawianym w podrozdziale 4.8 pojęciem stanu przejściowego. Niezależnie od przyjętych opisywanych tam założeń, należy ustalić jak długo zdarzenie może wpływać na działanie układu.

W większości wariantów diagramów statechart przyjmuje się, że zdarzenie ma naturę chwilową, tzn. że występuje w danym momencie czasu i istnieje tylko w tym momencie. Taki rodzaj zdarzenia nazywany jest zdarzeniem dyskretnym (*ang.* discrete event). Poruszanie kwestii związanej z czasem dostępności zdarzenia, w kontekście zdarzeń dyskretnych, może wydawać się nie na miejscu, lecz w podrozdziale 4.8 pokazano, że w trakcie jednego kroku, w grupie stanów podległych stanowi o typie *OR*, może zostać zrealizowanych kilka tranzycji, co do których przeważnie zakłada się, że są realizowane w czasie zerowym. Jeśli obie tranzycje są pobudzane tym samym zdarzeniem (rys. 4.9), to trzeba ustalić, czy realizacje kolejnych tranzycji są pobudzane tym samym zdarzeniem, czy też jego dwoma wystąpieniami.

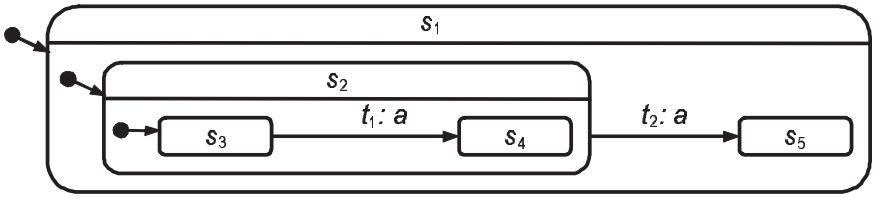
Rygorystycznie przestrzegając dyskretnej natury zdarzeń można by przyjąć, że zdarzenia są „konsumowalne" lub można by zabronić występowania stanów przejściowych. W pracy (Kesten i Pnueli, 1992) zaproponowano czasowy wariant diagramów. Przyjęto tam, że zdarzenia mają charakter dyskretny oraz zezwolono na występowanie stanów przejściowych. Dostępność zdarzenia w omawianym systemie, określona jest poprzez realizację tranzycji nieczasowych, czyli zdarzenie jest dostępne tak długo, jak długo czas nie postępuje, natomiast w przypadku realizacji tranzycji czasowych zdarzenia są konsumowane. Jeszcze inne podejście proponuje autor pracy (Peron, 1993), gdzie zamiast zdarzeń dyskretnych występują zdarzenia o określonym czasie trwania.

W systemie autora przyjmuje się, że zdarzenia mają charakter dyskretny, przy czym od momentu wystąpienia zdarzenia do momentu gdy jest ono dostępne dla układu, upływa jeden okres sygnału taktującego (patrz również punkty 7.2.1 i 7.2.2). Zdarzenia mogą być generowane w układzie poprzez realizacje tranzycji, akcji wejściowych (na diagramie słowo kluczowe *entry*) oraz akcji wyjściowych (słowo kluczowe *exit*); wówczas czas dostępności takiego zdarzenia uzależniony jest od okresu sygnału zegarowego. Natomiast w przypadku akcji statycznych (słowo kluczowe *do*) zdarzenie jest rozgłaszane przy każdym takcie zegara, tak długo jak długo stan, któremu jest przypisane jest aktywny. Wystąpienie i dostępność zdarzeń związanych z wejściem układu uzależniona jest od świata zewnętrznego.

4.10. Determinizm

Niedeterminizm w diagramach jest niejednoznaczną formą opisu występującą w sytuacji, gdy dwie tranzycje wychodzące z tego samego stanu są gotowe do realizacji. Większość omawianych wariantów diagramów, oprócz systemu Argos (Maraninchi, 1991), pozwala na stosowanie tego typu konstrukcji opisowych. W diagramach mo-

że pojawić się niedeterminizm innego rodzaju, mniej jawny, w sytuacji gdy jedna tranzycja wychodzi ze stanu s, a druga z jego stanu nadrzędnego i predykaty obu tranzycji są spełnione. Przykładem diagramu z niedeterminizmem jest diagram przedstawiony na rysunku 4.10. Powstanie takiej sytuacji bywa nazywane konfliktem między tranzycjami (Harel i Naamad, 1996).



Rys. 4.10. Modularny diagram z możliwym niedeterminizmem

Diagramy modelowane według autorskiego wariantu, które są docelowo realizowane jako układ cyfrowy, powinny charakteryzować się pełnym determinizmem. W związku z tym, dla zapewnienia pełnej jednoznaczności opisu stosuje się wzajemnie wykluczające się predykaty nałożone na tranzycje, będące wyrażeniami logicznymi operującymi na zdarzeniach i zdarzeniach zanegowanych (podrozdział 7.3).

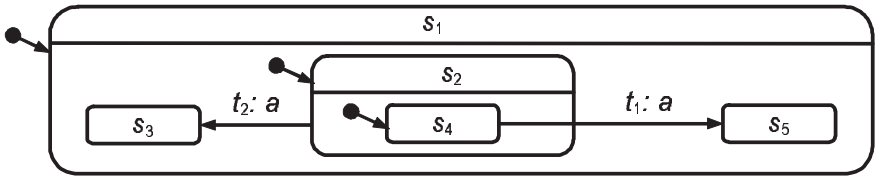
4.11. Priorytety realizacji tranzycji

W podrozdziale 4.10 przedstawiono zagadnienie determinizmu w diagramach. W systemie *HiCoS* proponuje się, stosowanie predykatów do rozwiązywania konfliktów między tranzycjami, gdyż te, w sposób dowolny dla projektanta, pozwalają ustalić priorytety realizacji tranzycji. Z drugiej jednak strony, umieszczanie na diagramach złożonych wyrażeń boolowskich, może zaciemniać czytelność diagramu (patrz przykład reaktora, dodatek C.4), a same wyrażenia mogą być źródłem prostych pomyłek. W związku z tym proponuje się pewne semantyczne reguły, które w prosty sposób, bez potrzeby pisania dodatkowych wyrażeń, ustalają priorytety tranzycji, rozwiązując w ten sposób część możliwych konfliktów.

W diagramie przedstawionym na rysunku 4.10, w momencie aktywności stanu s_3 i wystąpieniu zdarzenia a, tranzycje t_1 i t_2 będą w konflikcie, gdyż, jak widać, obie będą gotowe do realizacji. Według pierwszej publikacji opisującej diagramy w sposób formalny (Harel i Politi, 1998) jednoznacznie nie można stwierdzić, która tranzycja ma priorytet. Z kolei w publikacji (Peron, 1993), autor proponuje, aby priorytet realizacji tranzycji uzależnić od poziomu tzw. zakresu tranzycji. Im wyższy jest poziom hierarchii, z którym zakres jest związany, tym priorytet tranzycji jest wyższy. Poprzez pojęcie zakresu tranzycji, w innych publikacjach nazywanego areną (np. Maggiolo-Schettini i Merro, 1997), należy rozumieć taki najniższy, w sensie drzewa hierarchii diagramu, stan o typie *OR*, który jednocześnie, dla stanu początkowego i końcowego danej tranzycji, jest stanem nadrzędnym. Według tego kryterium w omawianym przykładzie tranzycja t_2 (jej zakresem jest stan s_1)

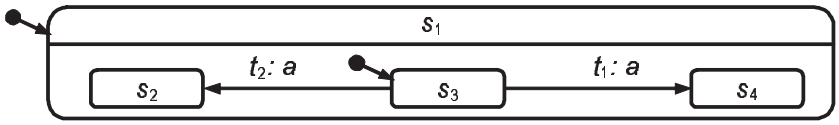
posiada priorytet przed tranzycją t_1 (której zakresem jest stan s_2).

Powiązanie priorytetu tranzycji z poziomem hierarchicznym poziomem zakresu nie we wszystkich przypadkach może rozwiązać konflikt. Taka sytuacja została przedstawiona na rysunku 4.11. Rozpatrywane tam tranzycje t_1 i t_2 są w konflikcie, gdyż ich zakresy są równoważne. Ze względu na występowanie możliwej równoważności zakresów tranzycji (zwłaszcza w sytuacjach gdy te tranzycje przekraczają granice stanów), proponuje się dodatkowo rozpatrywanie poziomu stanu źródłowego tranzycji. Podejście to zostało zaproponowane przez autora publikacji (Day, 1993). Według tego kryterium, tranzycja t_2 ma priorytet nad tranzycją t_1, gdyż jej stan początkowy znajduje się na wyższym poziomie hierarchii.



Rys. 4.11. Niemodularny diagram z możliwym niedeterminizmem

Ostatnim analizowanym wariantem zaistnienia konfliktu między tranzycjami jest sytuacja najbardziej trywialna, gdy tranzycje o takich samych warunkach wzbudzeń wychodzą z tego samego stanu. Ilustruje to kolejny przykład (rys. 4.12). W tym przypadku proponowane dwa kryteria nie rozstrzygną konfliktu między tranzycjami, gdyż zakresy tranzycji są tożsame, oraz stany początkowe obu tranzycji to ten sam stan s_3. Diagram z takimi tranzycjami jest więc diagramem niedeterministycznym.

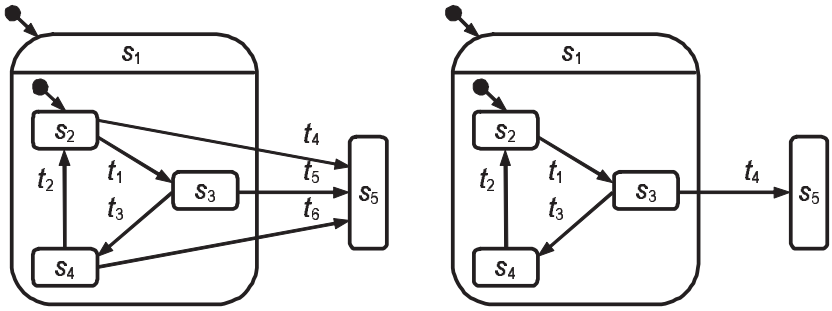


Rys. 4.12. Najprostszy przypadek tranzycji w konflikcie

W systemie *HiCoS* do rozwiązywania konfliktów wykorzystano wyłącznie predykaty, szczegółowo omówione w podrozdziale 7.3.

4.12. Wywłaszczeniowy i niewywłaszczeniowy tryby przekazywania sterowania

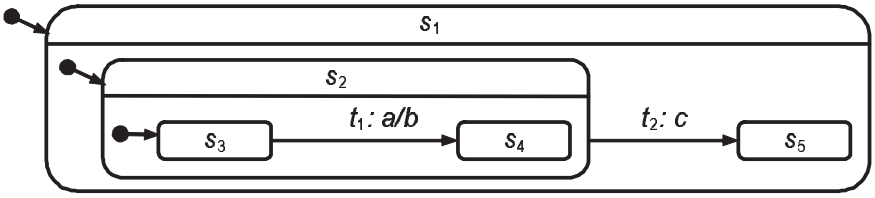
Jedną z najważniejszych właściwości diagramów statechart jest właściwość, która w porównaniu do tradycyjnego grafu stanów automatu sekwencyjnego, prowadzi do redukcji tranzycji na diagramie. Dzieje się to poprzez odpowiednie wykorzystanie właściwości stanów o typie *OR* i tranzycji wyższych poziomów hierarchii, niekiedy nazywanych abstrakcyjnymi.



Rys. 4.13. Uproszczenie diagramu poprzez zastosowanie tranzycji abstrakcyjnej

Rysunek 4.13 przedstawia dwa diagramy modelujące równoważne zachowania, różniące się liczbą tranzycji. Diagram pierwszy, bardziej złożony, zawiera wyłącznie tranzycje nieabstrakcyjne (podstawowe), poprowadzone między stanami najniższego poziomu hierarchii. Opis tego samego zachowania można uzyskać stosując mniejszą liczbę tranzycji, przez co diagram staje się prostszy i bardziej czytelny. Do tego celu wykorzystuje się tzw. tranzycje abstrakcyjne, poprowadzone między stanami wyższych poziomów hierarchii (zwanych abstrakcyjnymi). W omawianym przykładzie taką tranzycją jest tranzycja t_4 (poprowadzona z abstrakcyjnego stanu s_1), która w drugiej wersji modelu zastępuje grupę podstawowych tranzycji, poprowadzonych między wszystkimi stanami podległymi stanowi s_1 a stanem s_5, przy czym wszystkie tranzycje z opisywanej grupy mają takie same etykiety.

Jakkolwiek stosowanie tranzycji abstrakcyjnych upraszcza modelowanie, to istnieje pewien problem, związany z pozbawieniem sterowania stanów podległych stanowi początkowemu tranzycji abstrakcyjnej. Rysunek 4.14 przestawia taką sytuację, wymagającą szczegółowego rozpatrzenia. Abstrakcyjna tranzycja t_2 pełni rolę mechanizmu przerwania działania automatu sekwencyjnego podległego stanowi s_2. Sytuacja dyskusyjna powstaje w chwili, gdy aktywnym jest stan s_3 i jednocześnie wystąpią zdarzenia a oraz c.



Rys. 4.14. Realizacja tranzycji abstrakcyjnej a przekazanie sterowania

Rozróżnia się dwa możliwe sposoby przerwania działania podległego automatu: wywłaszczeniowy (*ang.* preemptive) i niewywłaszczeniowy (*ang.* non-preemptive).

- Przerwanie jest zwane wywłaszczeniowym, jeżeli realizacja tranzycji t_2 zapobiega wykonaniu tranzycji t_1.

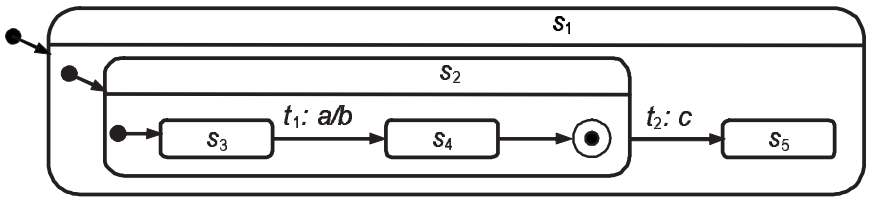
Przypadek można rozpatrzyć bardziej szczegółowo, w zależności od przyjęcia priorytetów realizacji tranzycji (podrozdział 4.11):

- Jedynie t_2 zostanie odpalona, gdyż tranzycje wyższych poziomów posiadają wyższy priorytet.
  Mimo że warunek wzbudzenia dla t_1 jest spełniony, to nie zostanie ona zrealizowana, gdyż znajduje się ona na niższym poziomie hierarchii. Akcja związana z jej realizacją, czyli rozgłoszenie wystąpienia zdarzenia b, również nie wystąpi. Sterowanie przejdzie ze stanów s_2, s_3 do stanu s_5.
- Jeżeli nie przyjęto omawianych reguł ustalania priorytetów, jedna spośród wzbudzanych tranzycji może zostać arbitralnie wybrana przez projektanta.
  Takie podejście zostało przyjete w systemie autora. Poziom hierarchii stanu początkowego tranzycji nie wpływa na priorytet jej realizacji. Wówczas taki diagram jest diagramem niedeterministycznym, a decyzje o realizacji tranzycji podejmowane są przez projektanta w oparciu o predykaty, będące wyrażeniami logicznymi tworzonymi przy użyciu zdarzeń i zdarzeń zanegowanych.

- Przerwanie jest nazywane niewywłaszczeniowym, jeżeli realizacja t_2 nie zapobiega wykonaniu tranzycji t_1.
  W tej sytuacji również można rozpatrzyć dwa przypadki uzależnione od przyjęcia opisywanej koncepcji priorytetów:
  - Priorytety obowiązują – tranzycja t_2 zostanie zrealizowana, lecz w tym trybie o tranzycji t_1 niczego się nie zakłada (realizacja uzależniona jest od istnienia priorytetów).
  - Priorytety nie obowiązują – obie tranzycje t_1 i t_2 zostaną zrealizowane jednocześnie.
    Realizacja tranzycji t_1 sprawi, że zostanie rozgłoszone wystąpienie zdarzenia b, mimo że jednocześnie zostanie zrealizowana t_2, pozbawiająca sterowania stany s_2 i stan s_3.

Istnieje jeszcze jedno możliwe rozwiązanie problemu przekazywania sterowania, zaimplementowane w systemie *HiCoS* i zademonstrowane na rysunku 4.15. Zastosowanie stanu końcowego, którego obecność przewyższa priorytet realizacji każdej tranzycji abstrakcyjnej sprawia, że nie ma konfliktów między tranzycjami. Obecność stanu końcowego decyduje o tym, że nadrzędna tranzycja t_2 zostanie zrealizowana, gdy sterowanie osiągnie stan końcowy (i jednocześnie wystąpi zdarzenie c).

Tabela 4.2 zawiera podsumowanie omawianych trybów przekazywania sterowania. Według autorów referatów (von der Beeck, 1994) oraz (Berry, 1993), kierujących się doświadczeniem zdobytym ze stosowania języka *Esterel*, zaleca się, aby w diagramach były dostępne zarówno tryb przekazywania wywłaszczeniowy, jak i niewywłaszczeniowy. W realizacji autora zdecydowano się na podejście określone mianem tryb wywłaszczeniowy bez obowiązywania priorytetów, oferując dodatkowo możliwość operowania stanem końcowym. Przyjęte rozwiązania wynikają



Rys. 4.15. Realizacja tranzycji abstrakcyjnej a stan końcowy

z pewnych przemyśleń dotyczących wygody opisu procesów sterowania oraz z możliwości oferowanych przez realizację sprzętową.

Tab. 4.2. Podsumowanie trybów przekazywania sterowania

Tryb przekazywania sterowania	Obowiązywanie priorytetów	Zrealizowane tranzycje
niewywłaszczeniowy	nie	t_1 i t_2
wywłaszczeniowy	nie	albo t_1 albo t_2
niewywłaszczeniowy	tak	na pewno t_2
wywłaszczeniowy	tak	tylko t_2
ze stanem końcowym	obojętne	tylko t_1

4.13. Różnica między zdarzeniami wejściowymi a pozostałymi zdarzeniami

Ze względu na miejsce powstawania oraz dostępność, zdarzenia w diagramach można podzielić na trzy grupy:

- dochodzące ze świata zewnętrznego – stanowiące wejście modelowanego systemu,
- widoczne dla świata zewnętrznego – stanowiące wyjście z systemu,
- generowane w modelowanym systemie i nie widoczne dla świata zewnętrznego – nazywane zdarzeniami wewnętrznymi lub lokalnymi.

Generalnie zdarzenia przynależne do każdej z grup posiadają takie same właściwości, jednak co do zdarzeń wejściowych obowiązuje pewne założenie. Otóż w celu spełniania hipotezy o doskonałej jednoczesności zakłada się, że w czasie realizacji kroku, zdarzenia ze świata zewnętrznego nie będą występować i w tym czasie nie będą wpływać na realizację kroku. Założenie to oczywiście nie dotyczy zdarzeń wewnętrznych i wyjściowych, których wystąpienia mogą być rozgłaszane w trakcie wykonywania kolejnych mikrokroków i wpływać na realizacje następnych tranzycji.

4.14. Tranzycje czasowe

Tranzycje czasowe stanowią najdogodniejszy środek modelowania zjawisk powiązanych z upływem czasu. Proponowane warianty diagramów dostarczają dwa rodzaje takich tranzycji, różniących się sposobem wyzwalania:

- stan źródłowy tranzycji musi być cały czas aktywny (do momentu realizacji tranzycji), a w przeciągu interwału czasowego, zadanego dolnym i górnym ograniczeniem czasowym, nałożonym na tranzycję, musi być spełniony warunek wzbudzenia tranzycji; w takim przypadku zdarzenie dyskretne nie może wyzwolić realizacji tranzycji,
- stan źródłowy tranzycji również musi być cały czas aktywny, a realizacja tranzycji występuje w chwili gdy w zadanym oknie czasowym (określonym dolną i górną granicą) wystąpi zdarzenie (lub grupa zdarzeń) określone w warunku wzbudzenia tranzycji; w tym przypadku zdarzenie dyskretne (lub grupa zdarzeń) mogą wyzwolić realizację tranzycji.

Ze względu na pewne trudności realizacyjne, autor w swym systemie nie wprowadził jeszcze tego rodzaju tranzycji.

4.15. Porównanie różnych wariantów diagramów

Pojawienie się w roku 1987 pierwszej o zasięgu światowym publikacji na temat diagramów statechart (Harel, 1987) spowodowało, że wielu naukowców zaczęło traktować diagramy jako wygodny i oczywisty sposób modelowania złożonego zachowania, głównie systemów reaktywnych. Diagramy zaczęto traktować jako naturalne rozwinięcie pojęcia grafu stanu automatu sekwencyjnego a ostatnio również jako rozwinięcie automatu współbieżnego graficznie reprezentowanego hierarchiczną siecią Petriego. Takie postrzeganie diagramów sprawiło, że znalazły one bardzo wiele praktycznych zastosowań, m.in. w projektowaniu systemów reaktywnych, systemów operacyjnych, sterowników cyfrowych, czy w modelowaniu obiektów z dziedziny inżynierii oprogramowania. Pewnym efektem ubocznym, zjawiskiem typowym przy tak szerokim zainteresowaniu, jest powstanie wielu różnych wariantów. Niniejszy podrozdział opisuje wybrane warianty diagramów (tab. 4.3) wraz z uwzględnieniem pewnych cech charakterystycznych i dokonuje porównania istniejącego stanu wiedzy z propozycją autora (tab. 4.4). Zawarte tu tabele są wzorowane na publikacji (von der Beeck, 1994).

Tabela 4.3 wymienia dziewiętnaście wybranych wariantów podając nazwę propozycji, nazwiska twórców, główną publikację omawiającą daną propozycję oraz pewne szczegółowe informacje dodatkowe. Tabela 4.4 dokonuje porównania wybranych wariantów (tab. 4.3) pod kątem dwudziestu czterech wybranych właściwości objaśnionych poniżej. Oto lista właściwości:

- postać graficzna/tekstowa – graficzna postać specyfikacji jest głównym atutem diagramów, lecz dla wygody prowadzonych prac badawczych nad semantyką opracowano dodatkowo postać tekstową,

- zdarzenia zanegowane – podrozdział 4.3,
- tranzycje czasowe – podrozdział 4.14,
- dysjunkcja wzbudzających zdarzeń – zasadniczo warunek wzbudzenia tranzycji jest zbiorem zdarzeń, przy czym do realizacji tranzycji konieczne jest wystąpienie wszystkich zdarzeń wymienionych w zbiorze (koniunkcja zdarzeń), niektóre z wariantów (np. *HiCoS*) oferują środki do budowy warunku wzbudzenia będącego dysjunkcją zdarzeń,
- warunek wzbudzenia – pozwala nie tylko na sprawdzanie czy określone zdarzenia wystąpiły, ale również uwzględnia zmienne, będącymi liczbami naturalnymi, o ile dany wariant umożliwia posługiwanie się takimi zmiennymi,
- odwołania do stanu – podrozdział 4.6,
- przypisania do zmiennych – niektóre warianty obok zdarzeń oferują operowanie zmiennymi globalnymi, których wartość jest nadawana przy realizacji tranzycji i może być sprawdzana w warunku wzbudzenia,
- tranzycje między poziomami hierarchii – podrozdział 4.5,
- mechanizm historii – pozwala na aktywacje automatu nie od stanu początkowego, lecz od stanu, który jest tzw. stanem ostatnio aktywnym,
- modułowość – podrozdział 4.7,
- przyjęcie hipotezy synchroniczności – podrozdział 4.1,
- determinizm – podrozdział 4.10,
- współbieżność pozorna/rzeczywista – w większości systemów współbieżność jest symulowana poprzez przydział czasu procesora kolejnym automatom sekwencyjnym, będącym w relacji współbieżności,
- czas dyskretny/ciągły – w większości przypadków diagramów czas ma naturę dyskretną, jedynie w przypadku diagramów z tranzycjami czasowymi jest inaczej,
- spójność globalna – podrozdział 4.4,
- przyczynowość – podrozdział 4.2,
- stany przejściowe – podrozdział 4.8,
- skończona liczba tranzycji – pewne warianty posiadają ograniczenie co do liczby tranzycji realizowanych w ciągu jednego momentu czasu (np. patrz podrozdział 4.8),
- priorytety – podrozdział 4.11,
- tryb niewywłaszczeniowy – podrozdział 4.12,

- tryb wywłaszczeniowy – podrozdział 4.12,
- rozróżnianie zdarzeń wejściowych – podrozdział 4.13,
- zasięg zdarzeń – pewne warianty oferują występowanie zdarzeń, które mogą być obserwowane w zakresie określonym przez poziom hierarchii gdzie są generowane,
- zdarzenia dyskretne/ciągłe – zdarzenie dyskretne istnieje tylko w jednym momencie czasu, podczas gdy istnienie zdarzeń ciągłych określone jest skończonym interwałem czasowym.

Tab. 4.3. Wybrane warianty diagramów statechart

Nr	Nazwa	Twórca (publikacja)	Inf. dodatkowe
1.	statecharty	Harel (Harel, 1987)	
2.	statecharty	Harel, Pnueli (Harel i Politi, 1998)	
3.	statecharty	Huizing, Gerth (Huizing i Gerth, 1992)	przyczynowe, spójność lokalna
4.	statecharty	Huizing, Gerth (Huizing i Gerth, 1992)	przyczynowe, spójność globalna
5.	statecharty	Huizing, Gerth (Huizing i Gerth, 1992)	
6.	statecharty	Pnueli, Shalev (Pnueli i Shalev, 1991)	sem. operecyjna
7.	statecharty	Pnueli, Shalev (Pnueli i Shalev, 1991)	semantyka dekl.
8.	Argos	Maraninchi (Maraninchi, 1992)	
9.	statecharty modularne	Classen (Classen, 1993)	
10.	statecharty	Maggiollo-Schettini, Peron (Maggiolo-Schettini i Peron, 1993)	
11.	statecharty	Day (Day, 1993)	
12.	statecharty	Peron (Peron, 1993)	bez st. przejśc., bez prioryt.
13.	statecharty	Peron (Peron, 1993)	bez st. przejśc., z prioryt.
14.	statecharty	Peron (Peron, 1993)	ze st. przejść., bez prioryt.
15.	statecharty	Peron (Peron, 1993)	ze st. przejśc., z prioryt.
16.	statecharty czasowe	Kesten, Pnueli (Kesten i Pnueli, 1992)	
17.	statecharty hybrydowe	Kesten, Pnueli (Kesten i Pnueli, 1992)	
18.	statecharty	von der Beeck (von der Beeck, 1994)	
19.	*HiCoS*	Łabiak (Łabiak, 2002*a*)	

Tab. 4.4. Porównanie wybranych właściwości diagramów statechart

Właściwość	Numer wariantu																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
postać graficzna/tekstowa	g	g	g	g	g	g	g	g	g/t	g	g	g	g	g	g	g	g	g	g/t
zdarzenia zanegowane	+	+	+	+	+	+	+	+	-	+	+	-	-	-	-	+	+	+	+
tranzycje czasowe	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	-
dysjunkcja wzbudz. zdarzeń	-	+	+	+	+	-	-	-	-	+	+	-	-	-	-	-	-	+	+
warunek wzbudzenia	+	+	-	-	-	-	-	-	-	-	+	-	-	-	-	+	+	-	+
odwołania do stanu	+	+	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	+
przypisania do zmiennych	+	+	-	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	+
tranzycje między poziomami	+	+	-	-	-	+	+	-	-	+	+	+	+	+	+	-	-	-	-
mechanizm historii	+	+	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	+
modułowość	-	-	-	-	-	-	-	+	+	-	-	+	+	+	+	+	+	+	+
przyjęcie hipotezy synchr.	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-
determinizm	-	-	-	-	-	-	-	+	-	-	-	-	-	-	-	-	-	-	+
współbieżność sym./rzecz.	s	s	s	s	s	s	s	s	s	s	s	r	r	r	r	r	r	?	r
czas dyskretny ciągły	d	d	d	d	d	d	d	d	d	d	d	c	c	c	c	c	c	c	d
spójność globalna	-	-	-	+	+	+	+	-	+	-	-	+	+	+	+	-	-	-	-
przyczynowość	+	+	+	+	-	+	+	-	-	+	+	+	+	+	+	+	+	+	+
stany przejściowe	?	-	-	-	-	-	-	-	+	-	-	-	-	+	+	+	+	+	+
skończona liczba tranzycji	?	+	+	+	+	+	+	+	-	+	+	+	+	+	+	-	-	+	-
priorytety	-	-	-	-	-	-	-	+	-	-	+	-	+	-	-	+	+	+	-
tryb niewywłaszczeniowy	?	-	-	-	-	-	+	-	-	-	-	-	-	+	+	?	?	+	+
tryb wywłaszczeniowy	?	+	-	-	-	+	+	+	-	+	+	+	+	-	-	+	+	+	+
rozróżnianie zdarzeń wjśc.	+	+	+	+	+	+	+	+	+	+	-	+	+	+	+	+	+	+	+
zasięg zdarzeń	-	-	-	-	-	-	-	+	+	-	-	-	-	-	-	-	-	+	-
zdarzenia dyskretne/ciągłe	d	d	d	d	d	d	d	d	d	d	d	c	c	c	c	d	d	d	d

Z przeprowadzonych ustaleń wynika, że zdecydowana większość wariantów (tab. 4.3) są to propozycje przeznaczone do implementacji programowych. Realizacja autora (w tabeli 4.4 punkt 19), system *HiCoS*, stanowi rzadki przykład implementacji sprzętowej (jako układ cyfrowy), co wyjaśnia w jaki sposób uzyskano współbieżność rzeczywistą. Trudności realizacyjne tłumaczą również, dlaczego zrezygnowano z takich właściwości jak tranzycje czasowe, skończona liczba tranzycji czy zdarzenia ciągłe. W tym miejscu należy zaznaczyć, że diagramy statechart nie są jedynym modelem formalnym dedykowanym systemom reaktywnym. Na Uniwersytecie Zielonogórskim opracowany został system (Andrzejewski, 2001) wykorzystujący model sieci Petriego (pewną hierarchiczną odmianę), która następnie jest programowo implementowana w mikrosystemie cyfrowym (Andrzejewski, 2002*b*). Inną równie ciekawą propozycją specyfikowania zachowań reaktywnych, również opracowaną na Uniwersytecie Zielonogórskim, jest system *PeNCAD*, gdzie wykorzystuje się hierarchiczną sieci Petriego typu makro i następnie taki model jest syntezowany i implementowany jako układ cyfrowy w strukturach *FPGA* (Węgrzyn, 1998*a*; Wolański, 1998*a*). Wyniki tych prac zostały zamieszczone m.in. w publikacjach (Węgrzyn, 1998*a*; 1998*b*; Węgrzyn i Adamski, 1999).

4.16. Podsumowanie

W rozdziale dokładnie omówiono problemowe kwestie semantyczne, typowe dla modelowania współbieżnego zachowania systemów reaktywnych czasu rzeczywistego. W czytelnej tabelarycznej formie odniesiono się do dokonań innych uznanych naukowców w tej dziedzinie, zestawionymi z własnymi rozwiązaniami.

Ustosunkowując się do przedstawionych właściwości systemu *HiCoS*, autor chciałby wskazać pewne dalsze możliwości poszerzenia własnej propozycji. Wydaje się, że interesującym jest wzbogacenie opracowanego systemu o następujące cechy: priorytety, zasięg zdarzeń, tranzycje przekraczające granice stanów. Priorytety w postaci opisywanej w punkcie 4.11 upraszczają specyfikowanie zachowań deterministycznych. Mechanizm zasięgu zdarzeń pełni podobną rolę jak mechanizm ukrywania danych w programowaniu obiektowym, oraz zwiększa przestrzeń nazw dla sygnałów i zmiennych. Tranzycje przekraczające granice stanów co prawda są mechanizmem, który pogarsza czytelność diagramu, lecz posiada pewne sensowne zastosowania. W swej naturze podobny jest on do niesławnej instrukcji *goto* i może być wykorzystany do nagłego przerwania złożonych obliczeń jedną prostą tranzycją. Podobne zalecenie co do instrukcji *goto* można znaleźć w książce B. Stroustrupa (Stroustrup, 2002) twórcy języka *C++*. Dodatkowo wprowadzenie tranzycji przekraczających granicę stanu pociąga za sobą możliwość operowania tranzycjami złożonymi, czyli takimi jakie występują w sieciach Petriego, które to tranzycje są doskonałym narzędziem synchronizowania procesów współbieżnych.

Rozdział 5

WYBRANE SYSTEMY CAD WYKORZYSTUJĄCE DIAGRAMY STATECHART

Rozdział omawia główne przemysłowe wykorzystanie diagramów statechart. Przedstawione tu systemy stanowią fragment dostępnej oferty rynkowej i zostały zestawione tak, aby odzwierciedlały przekrój możliwych zastosowań. Szczególnie dużo miejsca zostało poświęcone pakietowi *Statemate MAGNUM*, który jest pierwszym komercyjnym środowiskiem wykorzystującym diagramy statechart i zarazem jedynym znanym autorowi pakietem produkującym równoważny opis w językach *HDL*.

5.1. BetterState

Program *BetterState* firmy *Wind River* (Bet, 2004), przeznaczony na platformę *Windows 95/98/NT*, stanowi narzędzie do tworzenia oprogramowania przy użyciu konstrukcji graficznych. Główną notacją stosowaną w programie są diagramy statechart, diagramy stanów i sieć działań. Oprócz graficznego modelowania zachowania systemu, możliwe jest uruchomienie i graficzne prześledzenie działania tworzonego projektu oraz wygenerowanie równoważnego mu kodu w jednym z języków programowania wysokiego poziomu. Realizowane projekty docelowo mogą znajdować zastosowanie w systemach osadzonych (*ang.* embedded system) oraz w systemach operacyjnych czasu rzeczywistego (*ang.* Real Time Operating System).

Diagramy statechart wykorzystane w środowisku *BetterState* charakteryzują się występowaniem następujących konstrukcji: atrybut historii, stan początkowy, stan końcowy, akcje związane ze stanem (wyjściowe, statyczne, wyjściowe), asercje wykonywane w przypadku nie spełnienia określonych warunków, operowanie warstwami definiującymi przynależność stanów, tranzycje wzbudzane zmiennymi i zdarzeniami (którym można przypisać akcje), hierarchia, współbieżność oraz stany przejściowe. Dodatkowo w tworzonych projektach istnieje możliwości stosowania takich konstrukcji programistycznych jak: regiony krytyczne, diagramy wielokrotnego wykorzystania, zmienne lokalne, komentarze.

Głównym celem stosowania systemu jest stworzenie modelu i wygenerowanie działającego oraz sprawnego kodu. System umożliwia generowanie kodu w językach *C*, *C++* i *Java*, przy czym, poprzez zestaw dostępnych licznych opcji, użytkownik ma możliwość wpływania na ostateczną postać otrzymywanego kodu. Pewną

ciekawostką może być fakt, że w swej jednej z pierwszych wersji, program *BetterState* oferował możliwość przejścia z opisu zachowania sieciami Petriego zarówno na język *VHDL* jak i *Verilog* oraz, że modele w tych językach były syntezowalne. Jednak, jak to pokazano w pracy (Wolański, 1998*b*), wyniki uzyskane w przypadku syntezy sieci Petriego w strukturach programowalnych dalece odbiegały od wyników konkurencyjnych metod syntezy, również przedstawionych w pracy (Wolański, 1998*b*) oraz w pracach (Biliński, 1996; Kozłowski, 1993; Kozłowski i in., 1995; Puczyńska i in., 2000; Wolański, 1998*a*).

5.2. IAR visualSTATE

Program *visualSTATE* (IAR, 2004) jest jednym z czterech głównych produktów firmy *IAR System*. Firma została założona w 1983 roku, a od lipca roku 2000 jest notowana na giełdzie w Sztokholmie. Głównym celem stosowania systemu *visualSTATE* jest kompletna realizacja całego procesu rozwoju oprogramowania przeznaczonego dla systemów osadzonych, które następnie znajdują swoje zastosowanie w przemyśle samochodowym (np. zabezpieczenia przed kradzieżą i włamaniem), telekomunikacyjnym (zwłaszcza telefonia komórkowa), w systemach nawigacyjnych (np. *GPS*), w bankowości czy w służbie zdrowia. Praca z systemem składa się z kilku etapów intensywnie wspieranych przez graficzne środowisko: projektowanie, prototypowanie, generowanie kodu, testowanie, dokumentowanie. Działanie programu tworzonego w systemie modelowane jest jako sekwencja przejść pomiędzy zbiorami stanów. Stan modelowanego programu może się zmieniać zarówno pod wpływem zdarzeń dochodzących z zewnątrz jak i generowanych wewnętrznie. Inaczej mówiąc, tworzony system dokonuje odwzorowania zdarzeń wejściowych na odpowiednie zdarzenia wyjściowe, czyli projektowany program zachowuje się jak system reaktywny.

Projektowanie oprogramowania w systemie odbywa się poprzez rysowanie diagramu statechart. Do dyspozycji projektanta są następujące elementy składniowe: stany, tranzycje, zdarzenia, akcje, stan początkowy, zmienne, przypisania, regiony współbieżne, atrybuty historii płytkiej i głębokiej, sygnały, akcje wejściowe, wyjściowe i statyczne – tak jak to jest proponowane w standardzie języka *UML* (UML, 2003).

Celem procesu prototypowania w środowisku *visualSTATE* jest usprawnienie procesu komunikowania pomiędzy inżynierami, pracownikami działu marketingu i potencjalnymi użytkownikami na wszystkich etapach projektowania i implementacji produktu. Realizowane jest to poprzez stworzenie wirtualnego środowiska, które interaktywnie współdziała z opracowywanym modelem.

Generowanie kodu odbywa się w sposób automatyczny. Działanie otrzymanego kodu jest w 100% zgodne z graficzną specyfikacją, a docelowym językiem programowania jest język *ANSI-C*. Jak to można przeczytać w materiałach firmowych (IAR, 1999), objętość wygenerowanego kodu zazwyczaj jest mniejsza od objętości kodu pisanego ręcznie, a jest to uzyskane dzięki tablicowemu kodowaniu tabel przejść automatów. Kod generowany przez środowisko jest przeznaczony na mikrokontrolery jednoukładowe 8-, 16- i 32-bitowe.

Oprócz metod testowania kierowanego przez użytkownika, środowisko oferuje dynamiczną weryfikację formalną, realizowaną z wykorzystaniem grafu osiągalności. W systemie, tą metodą można sprawdzić między innymi istnienie następujących niepożądanych sytuacji: zakleszczenie, tranzycje nieosiągalne, konflikty. Proces dokumentowania jest zapisem ze wszystkich kolejnych etapów projektowych. Realizowane jest to w postaci diagramów tworzonych przez projektanta lub raportów automatycznie generowanych przez środowisko.

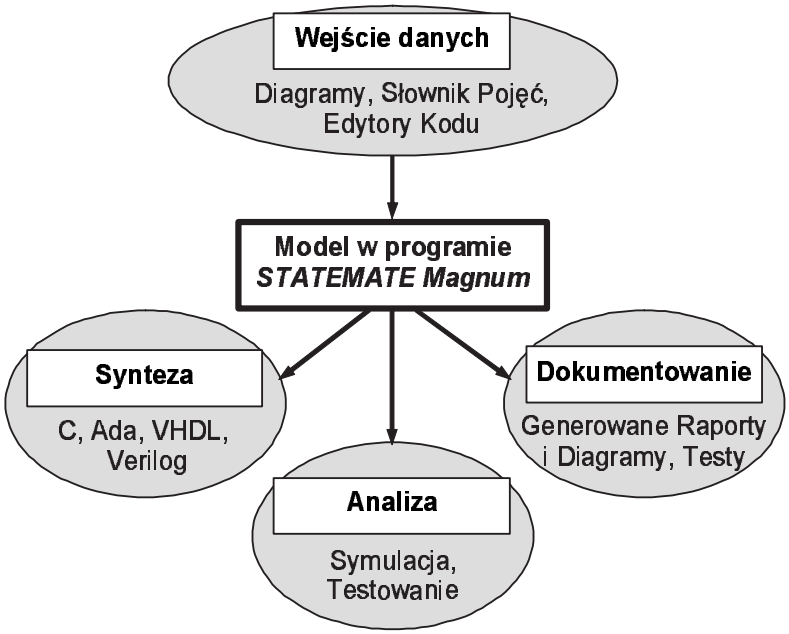
5.3. Statemate MAGNUM

Program *Statemate MAGNUM* (STA, 2004) jest produktem amerykańskiej firmy *I-Logix*, której założycielami są pomysłodawca diagramów statechart dr Dawid Harel oraz dr Amir Pnueli, zdobywca nagrody Turinga za całokształt kariery naukowej (STA, 2004). Celem powstania firmy w roku 1987 były komercjalizacja i dalszy rozwój systemu *Statemate MAGNUM*, stanowiącego praktyczną implementację dotychczasowych osiągnięć obu naukowców w zakresie formalnych technik walidacji złożonych systemów osadzonych, stosowanych głównie w przemyśle lotniczym. Dynamicznie prowadzone prace nad systemem *Statemate MAGNUM* zaowocowały opracowaniem szeregu nowych technik i możliwości systemu, czego wyrazem było w roku 1996 przemianowanie nazwy na *Statemate MAGNUM*. Obecnie firma *I-Logix* jest wiodącym producentem i dostawcą oprogramowania i metodologii automatyzujących proces tworzenia systemów osadzonych czasu rzeczywistego. Drugim flagowym produktem firmy, z którym *Statemate MAGNUM* ściśle współpracuje, jest pakiet *Rhapsody* (podrozdział 5.4) wspomagający programowanie metodami obiektowymi z wykorzystaniem technologii *UML*.

Zadaniem programu *Statemate MAGNUM* jest umożliwienie projektantowi przeprowadzenie szybkiej specyfikacji oraz walidacji poziomu systemu, złożonych modułów osadzonych, zwłaszcza posiadających cechy systemów reaktywnych (rys. 5.1). Realizowane jest to poprzez fakt, że specyfikowane zachowanie jest w pełni wykonywalne, dzięki czemu taki model może być animowany, symulowany, uwiarygodniany i analizowany zanim jeszcze zostanie poddany fazie implementacji. Specyfikacja modelu dokonywana we wczesnej fazie projektowania – analizy wymagań stawianych projektowi – jest realizowana w postaci przejrzystych notacji graficznych, nie wymagających przygotowania inżynierskiego, przez co staje się doskonałą dokumentacją projektu oraz środkiem komunikacji dla wszystkich uczestników procesu realizacji projektu, poczynając od zleceniodawcy, poprzez inżynierów wykonawców, podwykonawców, a na pracownikach działu marketingu i wdrożeń kończąc.

Praca z systemem polega na specyfikacji modelu przy użyciu trzech języków modelowania (Harel i Politi, 1998):

- diagram aktywności – wyglądem swym przypomina wielopoziomowy diagram przepływu danych, a jego głównym zdaniem jest przedstawienie tego, „co" projektowany system robi, czyli jakie funkcje są realizowane przez system i jakie informacje przepływają pomiędzy przedstawianymi jednostkami funkcyjnymi (*ang.* activities),



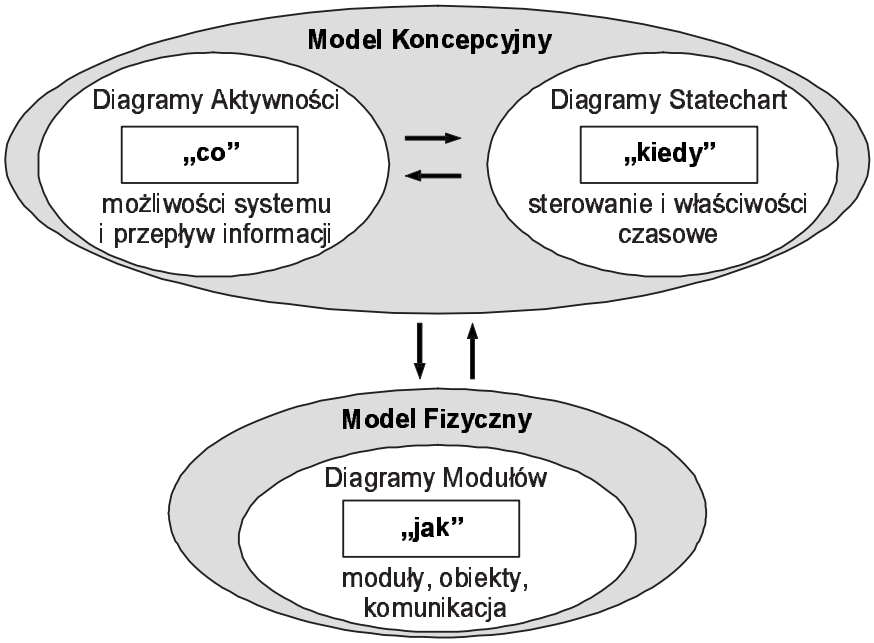
Rys. 5.1. Praca w systemie *Statemate MAGNUM* (Harel i Politi, 1998)

- diagram statechart – wykorzystana semantyka diagramów w systemie to m.in.: mechanizm rozgłaszania, właściwości czasowe, atrybut historii (dokładny opis znajduje się w pracach: Harel i Politi, 1998; Harel i Naamad, 1996), a celem stosowania tego modelu jest opis zachowania na zasadzie „kiedy", czyli przy spełnieniu jakich warunków jednostki funkcyjne z diagramu aktywności są aktywne i kiedy ma miejsce przepływ informacji między nimi,

- diagram modułów – łączy w sobie cechy diagramu przepływu danych oraz diagramu blokowego i w sposób strukturalny przedstawia elementy składające się na implementację modelowanego systemu, mówi o tym „jak" system będzie funkcjonował; na tym diagramie można dokonać podziału na część sprzętową i programową wraz z uwzględnieniem przepływu danych i sterowania, dodatkowo można zawrzeć bloki funkcyjne (takie jak zamieszczone na diagramie aktywności) oraz uwzględnić urządzenia i układy peryferyjne.

Rysunek 5.2 przedstawia zależności między diagramami w programie. Dwa pierwsze z wymienionych diagramów odpowiadają koncepcyjnemu spojrzeniu na model, natomiast diagram modułów odpowiedzialny jest za fizyczną implementację projektu.

Jednym z głównych celów pracy z programem jest uzyskanie modelu projektowanego systemu posiadającego właściwość zadziałania (*ang.* executable). Możliwe jest to do uzyskania poprzez fakt, że języki modelowania posiadają ścisłą seman-



Rys. 5.2. Modelowanie w systemie *Statemate MAGNUM* (Harel i Politi, 1998)

tykę i dodatkowo wszelkie pojęcia wprowadzane na diagramach są umieszczane w tzw. słowniku danych (*ang.* data dictionary). Pozycję w słowniku można opisywać w kategoriach innych pojęć obecnych w modelu, stosując do tego celu dedykowany język, co sprawia, że zamieszczone opisy łączą trzy rodzaje diagramów w jeden wykonywalny model.

Mając przygotowany model w postaci wykonywalnej, można przystąpić do jego uruchomienia i dokonać sprawdzenia czy proponowany model spełnia stawiane wymagania. Jeżeli model zachowuje się zgodnie z oczekiwaniami, można przystąpić do generowania kodu w następujących językach (I-L, 2001): *ANSI-C*, *Ada*, *Embedded C*. Działanie programów opisanych otrzymanymi kodami ściśle pokrywa się z funkcjonowaniem modelu.

Kod 5.1. Fragment kodu w języku *VHDL* wygenerowany przez program *Statemate MAGNUM* (I-L, 2000*b*)

```
            . . .
architecture Arch_PAGER of PAGER is
            . . .
type tpALERT_MODE_states is (VIBRATION_MODE, BEEPER_MODE);
signal ALERT_MODE_isin: tpALERT_MODE_states;
            . . .
procedure
exec_ALERT_MODE is
    begin
```

```
        case ALERT_MODE_isin is
            when VIBRATION_MODE =>
                if SWITCH_POS = 2 then
                    ALARM <= not ALARM;
                    BEEPER <= '1';
                    ALERT_MODE_isin <= BEEPER_MODE;
                end if;
            when BEEPER_MODE =>
                if SWITCH_POS = 1 then
                   ALARM <= not ALARM;
                   BEEPER <= '0';
                   ALERT_MODE_isin <= VIBRATION_MODE;
                end if;
        end case;
    end exec_ALERT_MODE;
          . . .
```

Dodatkową opcją jest możliwość wygenerowania specyfikacji w językach *HDL* (*VHDL* oraz *Verilog*). Kod generowany w języku *HDL* posiada modularną budowę z wyraźnie zdefiniowanymi interfejsami, co sprawia, że w łatwy sposób może być połączony z innymi fragmentami kodu, napisanymi ręcznie lub wygenerowanymi przez inne narzędzia. Kod 5.1 zawiera przykładowy fragment kodu w języku *VHDL* (zaczerpnięty z opracowania I-L, 2000*b*), będący równoważnym opisem zachowania zamodelowanego prostym diagramem statechart. Produkowane przez środowisko modele *HDL* są tworzone na podstawie wzorców projektowych, na podobnych zasadach, jak się modeluje klasyczny układ sekwencyjny (zasady tworzenie takich modeli dla układów sekwencyjnych można znaleźć m.in. w pracach: Łuba i Zbierzchowski, 2000; Perry, 1993; Rushton, 1998; Skahill, 2001; Zwoliński, 2002). Ponadto, poprzez liczne możliwe ustawienia w programie, projektant może wpływać na ostateczną postać generowanego kodu. Otrzymany kod w języku *VHDL* charakteryzuje się wystąpieniem takich konstrukcji składniowych jak *procedure*, instrukcja procesu, instrukcje *case* czy *when*.

W systemie *CAD* autora, w odniesieniu do pliku wyjściowego *VHDL*, zostały przyjęte odmienne założenia – wygenerowany model jest całkowicie opisany na niskim poziomie *RTL*. Takie założenie oraz fakt, że program *HiCoS* ma charakter systemu otwartego – łatwego w modyfikacji i rozbudowie – sprawiają, że system ten stanowi doskonałe środowisko do eksperymentów nad nowymi koncepcjami. W tym miejscu należy zauważyć, że dokonywanie jakichkolwiek zmian w systemie komercyjnym, bez względu na to czy wynikają one z pobudek naukowych czy też jakichś innych, jest przez prawo autorskie zabronione. Dedykowany charakter systemu *HiCoS* – zastosowanie dla potrzeb projektowania cyfrowych układów sterowania binarnego – pozwala na stosowanie pewnych metod – zwłaszcza symbolicznych (podrozdział 8.6) oraz procedur optymalizacji i minimalizacji, które w metodyce programu *Statemate MAGNUM* wydają się być trudne, czy wręcz niemożliwe do zaimplementowania. Niebagatelną zaletą systemu *HiCoS* są również jego wymogi sprzętowe i programowe – program może być uruchomiony praktycznie na każdej instalacji systemu Windows i nie jest wymagane żadne dodatkowe oprogramowa-

nie (w przypadku *Statemate MAGNUM* wymagana jest zasobochłonna nakładka *XVision*).

5.4. Rhapsody

Pakiet Rhapsody (Rha, 2004) firmy *I-Logix* jest wizualnym środowiskiem programistycznym, usprawniającym pracę programistów na etapie analizy i projektowania, stosujących techniki zorientowane obiektowo i tworzących oprogramowanie dla osadzonych systemów czasu rzeczywistego (I-L, 2000*a*). Modelowanie oprogramowania dla takich systemów musi uwzględniać ich typowe właściwości takie jak: reaktywność, pobudzanie zdarzeniami, ścisłe uwarunkowania czasowe, architektura wielozadaniowa.

Modelowanie w pakiecie w głównej mierze opiera się na technologii *UML*, co sprawia, że proces projektowy odbywa się według znanych i sprawdzonych wzorców. Pierwszym etapem jest analiza modelowanego systemu wraz z jego możliwym obszarem zastosowań. Wynikiem analizy jest wyodrębnienie obiektów świata rzeczywistego i ustalenie powiązań między nimi, co stanowi podstawę tworzonego modelu. Kolejnym etapem jest analiza zachowania systemu (przedstawiana za pomocą diagramów użycia), a zwłaszcza tego w jaki sposób z systemu będzie się korzystało (do tego celu wykorzystywany jest diagram sekwencji). Wyodrębnionym obiektom przyporządkuje się klasy (np. w sensie języka *C++* czy *Java*), a następnie za pomocą diagramów statechart, które w pakiecie stanowi główną notacją opisu zachowania, modeluje się zachowanie obiektów budowanych według ustalonych klas. Na tym etapie uwzględniane są zdarzenia występujące w systemie oraz pobudzenia przychodzące ze świata zewnętrznego. Semantyka diagramów jest taka sama jak w pakiecie *Statemate MAGNUM*. Tak przygotowany model jest modelem wykonywalnym i poprzez śledzenie wykonania oraz animowanie diagramów statechart i diagramów sekwencji może być testowany. Głównym celem testów jest sprawdzenie, czy otrzymane zachowanie modelu odpowiada stawianym oczekiwaniom. Powstały model stanowi dokumentację systemu i jednocześnie pełni rolę środka komunikacji w szerokiej grupie osób związanej z projektem (m.in. menedżerowie, kooperanci, klienci). Docelowymi językami implementacji modelu są: *ANSI-C*, *C++*, *Java* czy *Ada* w zależności do rodzaju nabytego pakietu.

Pakiet Rhapsody jest jednym z wielu dostępnych komercyjnych narzędzi wspierających programowanie technikami obiektowymi. O jego roli i pozycji na rynku świadczy podpisanie wielomilionowego kontraktu (Rea, 2001) prze firmę *I-Logix* z amerykańską firmą militarnych technologii lotniczych *Lockheed Martin*. Ocenia się, że firma *Lockheed Martin* oraz jej partnerzy w ramach projektu nad samolotem szturmowym *Joint Strike Fighter* (robocza nazwa *F-35*), mającym być głównym samolotem myśliwskim Stanów Zjednoczonych i państw sprzymierzonych, wykorzysta ponad tysiąc stanowisk pakietu Rhapsody. Głównym celem stosowania pakietu na bieżącym etapie prac w projekcie, jest modelowanie poziomu systemu i demonstracja koncepcji (*ang.* System Development and Demonstration).

5.5. Rational Rose

Pakiet Rational Rose (Rat, 2004) jest klasycznym przykładem środowiska programistycznego wspierającego modelowania oprogramowania przeznaczonego do implementacji w środowisku tradycyjnych systemów operacyjnych takich jak *Windows*, *Unix* czy *Linux*. Producentem pakietu jest korporacja *Rational* (Rat, 2004), notowana na nowojorskiej giełdzie spółek nowoczesnych technologii *NASDAQ*. Metodologia wykorzystana w pakiecie wspiera dwa zasadnicze elementy nowoczesnej inżynierii oprogramowania (Rat, 2000): programowanie z użyciem komponentów oraz kontrolowany, iteracyjny proces rozwoju oprogramowania. Pakiet *Rational Rose* umożliwia posługiwanie się następującymi technologiami: *UML*, *COM* (*ang.* Component Object Modelling) i *OMT* (*ang.* Object Modelling Technique). W pakiecie tym diagramy statechart są głównie wykorzystane do modelowania dynamicznego zachowania klas (klas w sensie obiektowych języków programowania). Przedstawiają one sekwencje dyskretnych stanów życia obiektu, poprzez które obiekt może przechodzić, uwzględniając zdarzenia będące przyczynami przejść i akcje będące ich rezultatami. Należy zaznaczyć, że semantyka diagramów statechart wykorzystana w pakiecie nie wspiera współbieżności. Opisywane modele mogą być implementowane w takich językach programowania jak: *Ada*, *C++*, *Java*, *Visual Basic*.

5.6. System COSMA

System *COSMA* (skrótowiec od słów ***CO****ncurrent* ***S****tate* ***MA****chine*) jest środowiskiem, które nie wprost wykorzystuje diagramy statechart, lecz zawarte w nim koncepcje i techniki są bardzo podobne do systemu *HiCoS*, opisywanego w niniejszej pracy. System ten jest akademickim produktem opracowanym w Instytucie Informatyki Politechniki Warszawskiej i jest przeznaczony do modelowania reaktywnych dyskretnych systemów sterowania (Daszczuk i in., 2001; COS, 2004). Podstawowym modelem wykorzystywanym w środowisku jest współbieżna maszyna stanów (*ang.* Concurrent State Machine). System ten składa się z trzech modułów:

- *Grapher* – jest edytorem graficznym umożliwiającym wyrysowanie grafu stanów oraz dokonuje zamiany graficznej postaci specyfikacji na postać tekstową w języku zbliżonym do języka *XML*, zwanym *CXL*,
- *Product Engine* – jest modułem zamieniającym opis w języku *CXL* na wyrażenia logiczne reprezentowane w systemie za pomocą binarnych diagramów decyzyjnych, na podstawie których generowana jest przestrzeń stanów w postaci symbolicznej, również przedstawiona za pomocą diagramów *BDD*,
- *TempoRG* – zadaniem tego modułu jest formalna analiza właściwości badanego modelu takich jak żywotność i bezpieczeństwo, z wykorzystaniem pojęć logiki temporalnej.

Jak można przeczytać w publikacji (Daszczuk i in., 2001) twórcy środowiska *COSMA* planują wzbogacenie systemu o możliwość operowania pojęciem hierarchii, co w istotnym stopniu upodobni model wykorzystywany w systemie do hierarchicznego współbieżnego automatu. Ponadto, planowane jest zrealizowanie przejścia na język *C* oraz na język opisu sprzętu *Verilog*. Innym kierunkiem prac jest współpraca z technologią *UML*.

5.7. Podsumowanie

Z przedstawionego zestawienia wynika, że głównymi komercyjnymi zastosowaniami diagramów są:

- modelowanie zachowania systemów reaktywnych czasu rzeczywistego,
- modelowanie zachowania obiektów z dziedziny technik programowania obiektowego.

Diagramy statechart ze względu na swoją czytelność dodatkowo są wykorzystywane do dokumentowania, a także jako środek do komunikowania w grupie, tak jak to jest proponowane w standardzie *UML*. Docelowo diagramy mogą być implementowane w postaci programowej z zastosowaniem języków wysokiego poziomu lub jako układ cyfrowy przy pomocy języków *HDL*. O rynkowym znaczeniu tej formy wizualizacji może świadczyć wielkość i powaga firm producentów oprogramowania wykorzystującego diagramy oraz długa lista najpoważniejszych korporacji stosujących te oprogramowanie. Można tu m.in. przytoczyć następujące przykłady: *Daimler Chrysler*, *Ericsson*, *Hitachi*, *Intel*, *Lockheed Martin*, *Mitsubishi*, *Motorola*, *NEC*, *Nokia*, *Oki*, *Samsung*, *Sony*, *Toshiba*, *Volvo*.

Jak widać diagramy statechart w bardzo małym stopniu są wykorzystywane do behawioralnej specyfikacji układów cyfrowych. Jedyną propozycją jest system *Statemate MAGNUM*, który jak się wydaje, ze względów cenowych, jest poza zasięgiem małego przedsiębiorstwa, zwłaszcza w warunkach polskich. Tutaj jest miejsce dla systemu *HiCoS*, w którym położono nacisk na efektywność implementacji w strukturach *FPGA*, co jest istotne przy projektowaniu małych systemów osadzonych o niskiej cenie.

Rozdział 6

SKŁADNIA I SEMANTYKA DIAGRAMÓW STATECHART – OPIS MATEMATYCZNY

W podrozdziale 3.3 zaprezentowano diagramy statechart jako podzbiór języka *UML*, koncentrując się głównie na aspektach postaci graficznej. Z kolei w rozdziale 4, szczegółowo na przykładach, omówiono zasadnicze kwestie semantyczne. Niniejszy rozdział stanowi podsumowanie dyskusji prowadzonej w obu podrozdziałach, czyni to w sposób bardziej ścisły, opisując diagramy aparatem matematycznym.

Przystępując do tworzenia opisu matematycznego, jako punkt wyjścia realizowanych prac, przyjęto definicje zawarte w publikacji (Maggiolo-Schettini i Merro, 1997) (oraz w mniejszym stopniu w Kyeyune, 2000). Przedstawiony tam aparat matematyczny dotyczył diagramów, które nie są modularne, nie obejmują takich elementów jak stan końcowy, atrybut historii, predykaty, akcje wejściowe, wyjściowe i statyczne. Ponadto składnia i semantyka tam przedstawione w żadnym stopniu nie uwzględniają ograniczeń realizacji sprzętowej. Powstały opis jest więc wynikiem adaptacji istniejącego modelu do własnych potrzeb autora.

6.1. Składnia diagramów

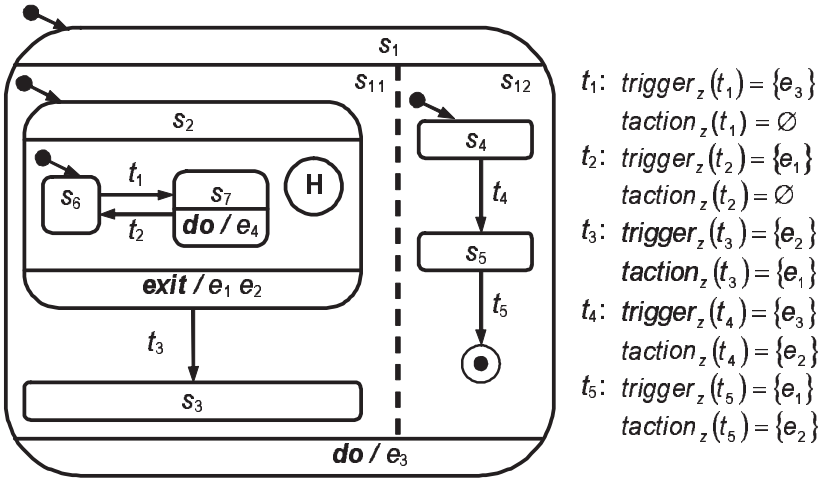
Formalne zdefiniowanie diagramów statechart wymaga wprowadzenia kilku pojęć pomocniczych. Zasadniczym pojęciem w głównej definicji (def. 6.4) jest pojęcie funkcji hierarchii, określonej przy pomocy relacji hierarchii. Aby zdefiniować relację hierarchii trzeba się posłużyć pojęciem przechodniego i zwrotnego domknięcia relacji. Poniżej są podane trzy definicje, zaczerpnięte z (Kyeyune, 2000), które wprowadzają wymienione pojęcia.

Definicja 6.1. *Niech $\rho \subseteq S \times S$, ρ^i jest określone dla $i \geqslant 0$, wówczas* **przechodnie i zwrotne domknięcie relacji** *ρ, oznaczane jako ρ^*, jest określone w sposób następujący:*

1. $\rho^0 = \{(s, s) \,|\, s \in S\}$,
2. $\rho^1 = \rho$,
3. $\rho^{n+1} = \rho \circ \rho^n$ *dla* $n \geqslant 1$, *przy czym dla dwu dowolnych relacji ξ oraz ζ, zdefiniowanych nad zbiorem S, zachodzi następująca zależność:*

$$\xi \circ \zeta = \left\{ (s,t) \in S \times S | \underset{u \in S}{\exists} : (s,u) \in \xi \wedge (u,t) \in \zeta \right\},$$

4. $\rho^* = \bigcup_{n \geqslant 0} \rho^n$ *oznacza zwrotne i przechodnie domknięcie relacji* ρ.



Rys. 6.1. Diagram statechart wraz z opisem tranzycji

Za przykład niech posłuży diagram z rysunku 6.1. Zbiór S – wierzchołków diagramu, relacja ρ – związek przodek-potomek są określone następująco:

$$S = \{s_1, s_2, s_{11}, s_{12}, s_3, s_4, s_5, endst, s_6, s_7\},$$

$$\rho = \left\{ \begin{array}{l} (s_1, s_{11}), (s_1, s_{12}), (s_{11}, s_2), (s_{11}, s_3), (s_{12}, s_4), (s_{12}, s_5), \\ (s_{12}, endst), (s_2, s_6), (s_2, s_7) \end{array} \right\}.$$

Punkt 1 w definicji 6.1 definiuje ρ^0 jako relację zwrotną określoną na zbiorze S:

$$\rho^0 = \left\{ \begin{array}{l} (s_1, s_1), (s_{11}, s_{11}), (s_{12}, s_{12}), (s_2, s_2), (s_3, s_3), (s_4, s_4), \\ (s_5, s_5), (endst, endst), (s_6, s_6), (s_7, s_7) \end{array} \right\}.$$

Punkt 2 mówi, że $\rho^1 = \rho$, a punkt 3 definiuje podzbiory relacji przechodniości ρ^n, dla $n \geqslant 1$:

$$\rho^2 = \rho \circ \rho^1 = \left\{ \begin{array}{l} (s_1, s_2), (s_1, s_3), (s_1, s_4), (s_1, s_5), (s_1, endst), (s_{11}, s_6), \\ (s_{11}, s_7) \end{array} \right\},$$

$$\rho^3 = \rho \circ \rho^2 = \{(s_1, s_6), (s_1, s_7)\},$$

$$\rho^4 = \rho \circ \rho^3 = \emptyset.$$

Z kolei zwrotne i przechodnie domknięcie jest uogólnioną sumą otrzymanych podzbiorów (punkt 4):

$$\rho^* = \rho^0 \cup \rho^1 \cup \rho^2 \cup \rho^3 = \left\{ \begin{array}{l} (s_1, s_1), (s_1, s_{11}), (s_1, s_{12}), (s_1, s_2), (s_1, s_3), \\ (s_1, s_4), (s_1, s_5), (s_1, endst), (s_1, s_6), \\ (s_1, s_7), (s_{11}, s_{11}), (s_{11}, s_2), (s_{11}, s_3), \\ (s_{11}, s_6), (s_{11}, s_7), (s_{12}, s_{12}), (s_{12}, s_4), \\ (s_{12}, s_5), (s_{12}, endst), (s_2, s_2), (s_2, s_6), \\ (s_2, s_7), (s_3, s_3), (s_4, s_4), (s_5, s_5), \\ (endst, endst), (s_6, s_6), (s_7, s_7) \end{array} \right\}.$$

Definicja 6.2. *Relacja ρ jest nazywana* **relacją hierarchii** *nad zbiorem S, z pewnym wyróżnionym stanem $r \in S$(zwanym korzeniem), wtedy i tylko wtedy, gdy:*

1. $\rho \subseteq S \times S$,
2. $\underset{s \in S}{\forall} : (s, r) \notin \rho$,
3. $\underset{s \in S \setminus \{r\}}{\forall} : (r, s) \in \rho^*$ *oraz* $\left(\overset{=1}{\exists}\, n \in N \right) : \left(\overset{=1}{\exists}\, s_1, s_2, \ldots, s_n \in S \right)$ $s_1 = r$ i $s_n = s$ *oraz* $(s_i, s_{i+1}) \in \rho$ *dla* $i = 1, \ldots, n-1$.

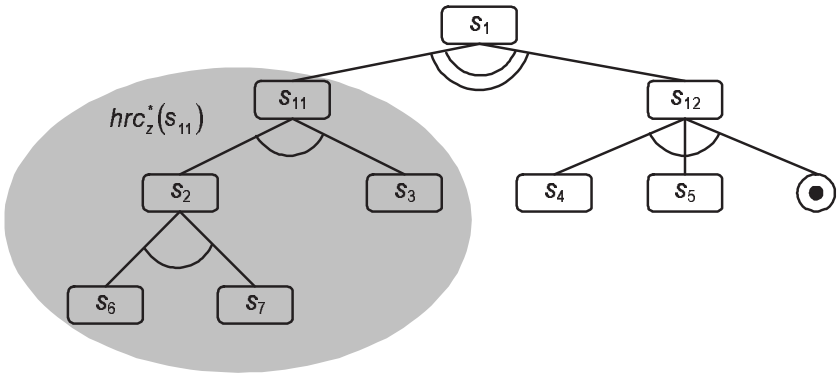
Przykładowa relacja jest relacją hierarchii gdyż spełnia warunki wymienione w definicji 6.2. Relacja ta jest określona na zbiorze S (punkt 1). Stanem wyróżnionym r, zwanym korzeniem, jest stan s_1 i w relacji tej nie ma takiego elementu, gdzie następnikiem byłby s_1 (punkt 2). Punkt 3 mówi o tym, że dla każdego stanu różnego od wyróżnionego stanu r, można utworzyć tylko jeden taki ciąg, którego pierwszym wyrazem jest wyróżniony stan r, pary sąsiadujących elementów ciągu należą do ρ, a ostatni wyraz z pierwszym wyrazem ciągu tworzą parę (r, s), która należy do domknięcia relacji ρ. Innymi słowy, Definicja 6.2 mówi, że gdyby stworzyć graf, którego wierzchołkami byłyby elementy zbioru S, a krawędziami związki między elementami tego zbioru określone przez relację, to otrzymane by zostało drzewo o korzeniu w wierzchołku r (rys. 6.2).

Do graficznego zilustrowania hierarchii, zawartej w diagramie statechart, najlepiej nadaje się drzewo hierarchii zwane czasami grafem *AND-OR*. Rysunek 6.2 przedstawia taki diagram hierarchii, bardzo podobny do opisywanego w (Gajski i in., 1994) diagramu Jacksona, który uwzględnia relacje między stanami diagramu. Podwójne łuki łączące krawędzie drzewa oznaczają, że podległe stany znajdują się w relacji współbieżności, np. stanowi s_1 przyporządkowano dwa stany s_{11} i s_{12}, co oznacza, że oba stany są jednocześnie aktywne. Łuk pojedynczy oznacza relację sekwencyjności, np. stanowi s_{11} przyporządkowano w ten sposób stany s_2 i s_3, co oznacza, że co najwyżej tylko jeden z takich stanów w danym momencie może być aktywny.

Definicja 6.3. Funkcje hierarchii *hrc, hrc^i, hrc^* odwzorowujące $S \to 2^S$ są zdefiniowane jak następuje (z uwzględnieniem relacji hierarchii nad zbiorem S):*

1. $hrc^0 = (S)$,

2. $hrc^i(s) = \{t \in S | (s,t) \in \rho^i\}$,
3. $hrc(s) = hrc^1(s)$,
4. $hrc^*(s) = \{t \in S | (s,t) \in \rho^*\}$.



Rys. 6.2. Diagram hierarchii dla diagramu statechart z rysunku 6.1

Funkcje hierarchii dla rozważanego przykładu i stanu s_1 są następujące:

$hrc^0(s_1) = \{s_1\}$,

$hrc^1(s_1) = hrc(s_1) = \{s_{11}, s_{12}\}$ – zbiór bezpośrednich potomków stanu s_1,

$hrc^2(s_1) = \{s_2, s_3, s_4, s_5, endst\}$,

$hrc^3(s_1) = \{s_6, s_7\}$,

$hrc^*(s_1) = \{s_1, s_2, s_{11}, s_{12}, s_3, s_4, s_5, endst, s_6, s_7\}$ – zbiór wszystkich potomków stanu s_1, łącznie ze stanem s_1.

Mając zdefiniowane funkcje hierarchii hrc i hrc^*, można formalnie zdefiniować statechart. Niech **S** będzie nieskończonym zbiorem stanów, **T** nieskończonym zbiorem tranzycji i **E** nieskończonym zbiorem zdarzeń. Symbolami s, s', s_1, s_2, s_3, ... określa się stany, które należą do **S**, symbolami t, t', t_1, t_2, t_3, ... określa się tranzycje, które należą do **T**, symbolami e, e', e_1, e_2, e_3, ... określa się zdarzenia, które należą do **E**. S oznacza podzbiór zbioru **S**, T oznacza podzbiór zbioru **T**, E oznacza podzbiór zbioru **E**.

Definicja 6.4. *Diagramem* **statechart** *Z nazywa się krotkę składającą się z następujących elementów:*

$$(S_z, hrc_z, type_z, default_z, history_z, E_z, T_z, out_z, in_z, tlabel_z, saction_z)$$

gdzie:

1. $S_z \subseteq$ S *jest skończonym niepustym zbiorem stanów.*

2. $hrc_z : S_z \rightarrow 2^{S_z}$ *jest* **funkcją hierarchii** *(*ang. *hierarchy function), która dla stanu $s \in S_z$, określa zbiór bezpośrednich podstanów stanu s. Stan $s \in S_z$ jest* **stanem podstawowym** *(prostym), jeżeli $hrc_z(s) = \emptyset$, w przeciwnym razie jest* **stanem złożonym**. *Funkcja $hrc_z^* : S_z \rightarrow 2^{S_z}$ dla stanu $s \in S_z$, określa zbiór wszystkich potomków stanu s. Istnieje jeden wyróżniony stan $s \in S_z$, oznaczany przez $root_z$, taki że $hrc_z^*(s) = S_z$. Dla każdego $s \in S_z$, dla którego $s \neq root_z$, istnieje jeden taki stan $s' \in S_z$, oznaczany przez $parent_z(s)$, dla którego $s \in hrc_z(s')$, w wyniku czego funkcja hrc_z przedstawia sobą strukturę drzewiastą. Każdemu $s \in S_z$ dla którego $hrc_z(s) \neq \emptyset$ i $type_z(s) = OR$, można przyporządkować jeden taki stan oznaczany przez $endst_z$, dla którego $endst_s \in hrc_z(s)$ i $hrc_z(endst_s) = \emptyset$ oraz dla każdej $t \in T_z$ zachodzi $out_z(t) \neq endst_s$. Dla niepustego zbioru stanów $S \subseteq S_z$* **najniższym wspólnym przodkiem** *(*ang. *lowest common ancestor) zbioru S, przedstawianym jako $lca_z(S)$ jest stan $s \in S_z$ taki że:*

   (a) $S \subseteq hrc_z^*(s)$,

   (b) $\underset{s' \in S_z}{\forall} (S \subseteq hrc_z^*(s') \Rightarrow s \in hrc_z^*(s'))$.

3. $type_z : S_z \rightarrow \{AND, OR\}$ *jest* **funkcją typu stanu** *(*ang. *state-type function) która przyporządkowuje typ każdemu stanowi złożonemu i jest nieokreślona dla stanów podstawowych. Bezpośrednim podstanem stanu o typie* AND *nie może być stan o typie* AND.

4. $default_z : S_z \rightarrow S_z$ *jest* **funkcją stanu początkowego** *(*ang. *default function), która dla każdego stanu $s \in S_z$, takiego że $hrc_z(s) \neq \emptyset$ i $type_z(s) = OR$ określa stan początkowy s_d związany ze stanem s. W pozostałych przypadkach funkcja ta jest nieokreślona.*

5. $history_z : S_z \rightarrow \{true, false\}$ *jest dwuwartościową* **funkcją historii** *(*ang. *history function), która dla każdego stanu $s \in S_z$, dla którego zachodzi warunek $type_z(s) \neq AND$, przyporządkowuje atrybut historii w ten sposób, że każdy ze stanów należący do $hrc_z(parent(s))$, posiada tę samą wartość atrybutu. Dla stanu, dla którego $type_z(s) = AND$ funkcja ta jest nieokreślona.*

6. $E_z \subseteq$ E *jest skończonym* **zbiorem zdarzeń** *(*ang. *event).*

7. $T_z \subseteq$ T *jest skończonym* **zbiorem tranzycji** *(*ang. *transition).*

8. $out_z : T_z \rightarrow S_z \setminus \{root_z\}$ *jest funkcją w pełni określoną, zwaną* **funkcją źródła tranzycji** *tranzycji (*ang. *transition source function), taką że $out_z(t) = s$ jeżeli tranzycja t bierze swój początek w stanie s. Tranzycja nie może brać swego początku od stanu który jest bezpośrednim podstanem stanu o typie* AND, *mianowicie dla każdej tranzycji $t \in T_z$ zachodzi $type_z(parent_z(out_z(t))) = OR$.*

9. $in_z : T_z \rightarrow S_z \setminus \{root_z\}$ *jest funkcją w pełni określoną zwaną* **funkcją celu tranzycji** *tranzycji (*ang. *transition target function), taką że $in_z(t) = s$*

jeżeli tranzycja t ma swój koniec w stanie s. Analogicznie do funkcji źródła, dla każdej tranzycji $t \in T_z$ zachodzi $type_z\,(parent_z\,(in_z\,(t))) = OR$.

10. *Dla każdej tranzycji $t \in T_z$ jest spełniony następujący warunek: $parent\,(out_z\,(t)) = parent_z\,(in_z\,(t)) = s$, przy czym $type_z\,(s) = OR$, z którego wynika, że* **tranzycja nie może przekraczać granic stanu**.

11. *$tlabel_z : T_z \rightarrow 2^{E_z} \times 2^{E_z}$ jest* **funkcją etykietowania tranzycji** *(*ang. *transition labelling function). Dwie składowe funkcji $tlabel_z$ są nazywane odpowiednio $trigger_z\,(t)$ oraz $taction_z\,(t)$.*

12. *$saction_z : S_z \rightarrow 2^{E_z} \times 2^{E_z} \times 2^{E_z}$ jest* **funkcją etykietowania stanu** *(*ang. *state labelling function), która określa zbiory zdarzeń związanych kolejno z przekazaniem sterowania do stanu – $entry_z\,(s)$, przebywaniem w stanie – $do_z\,(s)$ i opuszczeniem stanu – $exit_z\,(s)$.*

Symbol **Z** oznacza klasę diagramów statechart zgodnych z definicją 6.4, symbole Z, Z', Z_1, Z_2, Z_3, ... oznaczają diagramy statechart należące do klasy **Z**.

Zbiór stanów na jednym poziomie hierarchii, mających ten sam stan nadrzędny, nazywa się automatem sekwencyjnym, a stany – stanami lokalnymi. Stan wyróżniony strzałką o początku w czarnym kółku, jest stanem początkowym automatu (lub zwanym inaczej stanem domyślnym). Stan oznaczony jako czarne kółko w okręgu zwany jest stanem końcowym. Automat sekwencyjny posiadający taki stan nie może zostać pozbawiony sterowania przed osiągnięciem stanu końcowego. Każdemu stanowi można przyporządkować podległy mu automat sekwencyjny lub współbieżny. Automaty współbieżne rysowane są jako grupy podległych stanowi automatów sekwencyjnych, przedzielonych linią przerywaną. Szczegółowy opis diagramów jako połączonych automatów sekwencyjnych znajduje się w (Łabiak, 2001*b*) oraz w podrozdziale 3.3.

Przykłady pojęć wprowadzonych w definicji głównej (def. 6.4) zilustrowano diagramem przedstawionym na rysunku 6.1. Kolejne punkty odpowiadają kolejnym punktom z definicji.

1. Zbiór wszystkich stanów w systemie jest następujący: $S_z = \{s_1, s_2, s_{11}, s_{12}, s_3, s_4, s_5, endst, s_6, s_7\}$. Stany bezpośrednio podległe stanowi s_1 o typie AND, na diagramie są zaznaczane jako regiony stanu nadrzędnego, (s_1) oddzielone są linią przerywaną. Stan o nazwie *endst* jest stanem końcowym automatu podległego stanowi s_{12}.

2. Funkcja hierarchii hrc_z przyporządkowuje każdemu stanowi, zbiór stanów, które na diagramie hierarchii są bezpośrednimi jego potomkami. Na przykład dla stanów s_{11} i s_2, (rys. 6.2), zbiory te wynoszą odpowiednio: $hrc_z\,(s_{11}) = \{s_2, s_3\}$, $hrc_z\,(s_2) = \{s_6, s_7\}$. Ponieważ stan $s_2 \in hrc_z\,(s_{11})$, to stany należące do zbioru $hrc_z\,(s_2) = \{s_6, s_7\}$, są podrzędne w stosunku do stanu s_{11}. Zapis $parent_z$ ustala relacje w przeciwną stronę, na przykład $parent_z\,(s_2) = s_{11}$ oznacza, że stan s_{11} jest stanem bezpośrednio nadrzędnym w stosunku do s_2. Funkcja $hrc_z^*(s)$, dla dowolnego stanu s, określa wszystkie stany podrzędne

(bezpośrednio i pośrednio) stanowi s. Na przykład dla stanu s_{11} zbiór wszystkich stanów podległych jest następujący $hrc_z^*(s_{11}) = \{s_{11}, s_2, s_3, s_6, s_7\}$. Najniższym wspólnym przodkiem dla zbioru stanów S jest taki stan s, który w drzewie hierarchii, w grupie stanów nadrzędnych do każdego stanu należącego do S, leży najniżej. Na przykład chcąc znaleźć $lca_z(\{s_7, s_3\})$, na mocy punktu 2a stanami nadrzędnymi są s_1 i s_{11}. Stan s_1 nie spełnia implikacji 2b, gdyż można znaleźć taki stan s', dla którego implikacja nie zachodzi. Na przykład dla $s' = s_{11}$ poprzednik implikacji jest spełniony, stany s_3 i s_7 są podrzędne stanowi s_{11}, natomiast następnik nie jest spełniony, gdyż s_1 nie należy do $hrc_z^*(s_{11})$, zatem w sytuacji gdy z prawdy wynika fałsz implikacja nie jest spełniona. Dla stanu s_{11} i na mocy warunku 2a, nie ma takiego s' aby implikacja nie zachodziła, zatem $lca_z(\{s_3, s_7\}) = s_{11}$.

3. Wartości funkcji typu stanu dla stanów s_1 i s_2: $type_z(s_1) = AND$, $type_z(s_2) = OR$.

4. Wartości funkcji stanu domyślnego dla stanów s_{11} i s_2: $default_z(s_{11}) = s_2$, $default_z(s_2) = s_6$.

5. Wartości funkcji historii dla stanów s_2 i s_7: $history_z(s_2) = false$, $history_z(s_7) = true$.

6. Zbiór wszystkich zdarzeń w systemie: $E_z = \{e_1, e_2, e_3, e_4\}$.

7. Zbiór wszystkich tranzycji w systemie: $T_z = \{t_1, t_2, t_3, t_4, t_5\}$.

8. Wartość funkcji źródła dla tranzycji t_4: $out_z(t_4) = s_4$.

9. Wartość funkcji celu dla tranzycji t_4: $in_z(t_4) = s_5$.

10. Tranzycja t_3 nie przekracza granicy stanu s_{11}, nadrzędnego do stanów początkowego ($out_z(t_3) = s_2$) i końcowego ($in_z(t_3) = s_3$) tranzycji t_3 oraz jest typu OR. Formalnie zapisuje się to następująco:
$parent_z(out_z(t_3)) = parent_z(in_z(t_3)) = s_{11} \wedge type_z(s_{11}) = OR$. Łatwo sprawdzić, że w przykładzie (rys. 6.1) nie ma takiej tranzycji dla której warunek nie byłby spełniony.

11. Warunki wzbudzeń tranzycji: $trigger_z(t_1) = \{e_1\}$, $trigger_z(t_2) = \{e_2\}$, akcja tranzycji: $taction_z(t_1) = \emptyset$, $action_z(t_2) = \{e_3\}$.

12. Akcje w systemie: $exit_z(s_2) = \{e_1, e_2\}$, $do_z(s_1) = \{e_3\}$, $do_z(s_7) = \{e_4\}$.

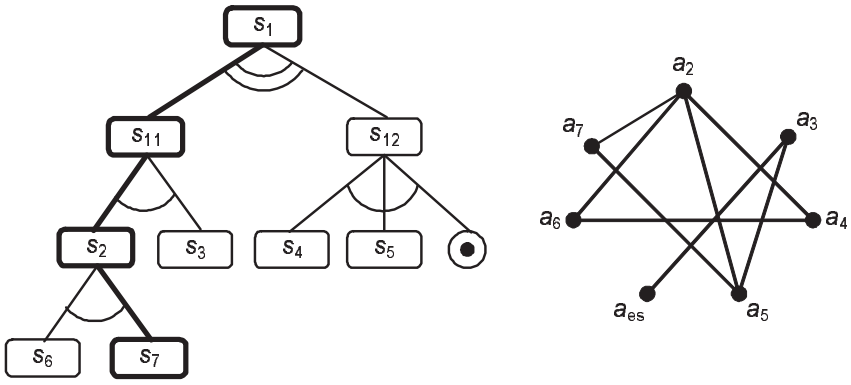
Definicja 6.5. *Dwie tranzycje $t_1, t_2 \in T_z$ są* **strukturalnie zgodne** *(*ang. *structurally consistent), jeżeli $type_z(lca_z(\{out_z(t_1), out_z(t_2)\})) = AND$. Zbiór tranzycji $T \subseteq T_z$ jest* **strukturalnie zgodny***, jeżeli każde dwie tranzycje $t_1, t_2 \in T$ są strukturalnie zgodne.*

Dwie strukturalnie zgodne tranzycje, znajdują się w dwóch różnych współbieżnych komponentach tego samego stanu o typie AND i potencjalnie mogą być

zrealizowane w tym samym mikrokroku. Przykładami tranzycji strukturalnie zgodnych są tranzycje t_3 i t_4 , dla których najniższym wspólnym przodkiem jest stan s_1 o typie *AND*.

Definicja 6.6. *Dwa stany* $s_1, s_2 \in S_z$ *są* **zgodne** *(*ang. *consistent), jeżeli* $s_1 \in hrc_z^*(s_2)$ *albo* $s_2 \in hrc_z^*(s_1)$ *albo* $type_z(lca_z(\{s_1, s_2\})) = AND$*. Zbiór stanów jest zgodny, jeżeli każde dwa stany* $s_1, s_2 \in S$ *są zgodne. Zbiór stanów* $S \subseteq S_z$ *jest* **maksymalnie zgodny**, *inaczej zwanym* **konfiguracją**, *i oznaczany* C*, jeżeli dla każdego stanu* $s \in S_z \backslash S$*, zbiór* $S \cup \{s\}$ *nie jest zgodny.*

Pojęcie zbioru stanów zgodnych służy określeniu zbioru wszystkich stanów diagramu aktywnych współbieżnie. Takimi stanami nie mogą być stany, które są bezpośrednio podległymi stanowi o typie *OR*. W nawiązaniu do rysunku 6.1, stany s_6 i s_7 nie są zgodne, stan s_2 jest zgodny ze stanami s_4 i s_6 , natomiast nie jest zgodny ze stanem s_3. Zbiór stanów $S = \{ s_1, s_{11}, s_{12}, s_2, s_4, s_6\}$ jest maksymalnie zgodny, $S\backslash \{s_{11}\}$jest tylko zgodny, a $S \cup \{s_3\}$ nie jest zgodny. Rysunek 6.3 przedstawia graf współbieżności dla przykładu z rysunku 6.1, gdzie stany aktywne współbieżnie (oznaczane literą a), będące czteroma zbiorami stanów zgodnych, są przedstawione jako kliki. Na rysunku pominięte zostały wierzchołki związane ze stanami s_1, s_{11}, s_{12}, które są zawsze aktywne i które w realizacji sprzętowej są pomijane (rozdział 7).



Rys. 6.3. Przykłady ścieżki $path\,(s_1, s_7) = \{s_1, s_{11}, s_2, s_7\}$ oraz graf współbieżności dla diagramu z rysunku 6.1

Definicja 6.7. *Dla dwóch stanów* $s_1, s_2 \in S$*, dla których* $s_2 \in hrc_z^*\,(s_1)$*,* **ścieżką** *(*ang. *path) ze stanu* s_1 *do stanu* s_2*, oznaczaną jako* $path\,(s_1, s_2)$*, jest zbiór* $\{s \in S_z | s \in hrc_z^*\,(s_1) \wedge s_2 \in hrc_z^*\,(s)\}$.

Ilustracją ścieżki jest podgraf diagramu hierarchii (rys. 6.3), na którym pogrubioną linią zaznaczono ścieżkę $path\,(s_0, s_7)$ wraz z należącymi do niej stanami.

Poniżej zostaną wprowadzone pojęcia mikrokonfiguracji. Mikrokonfiguracja jest zapisem zbioru stanów opuszczonych przez sterowanie, zbioru stanów bieżących, i zbioru aktualnych zdarzeń.

Definicja 6.8. Mikrokonfiguracją μC *jest trójka* $(S_{exit}, S_{curr}, E_{curr})$ *gdzie:*

1. $S_{exit} \subseteq S_z$ *zbiór stanów opuszczonych przez sterowanie,*
2. $S_{curr} \subseteq S_z$ *maksymalnie zgodny zbiór stanów bieżących,*
3. $E_{curr} \subseteq E_z$ *zbiór aktualnie dostępnych zdarzeń.*

Przykładem mikrokonfiguracji dla diagramu z rysunku 6.1 i sytuacji początkowej jest następująca trójka: $S_{exit} = \emptyset$, $S_{curr} = \{ s_1, s_{11}, s_{12}, s_2, s_4, s_6\}$, $E_{curr} = \{e_3\}$.

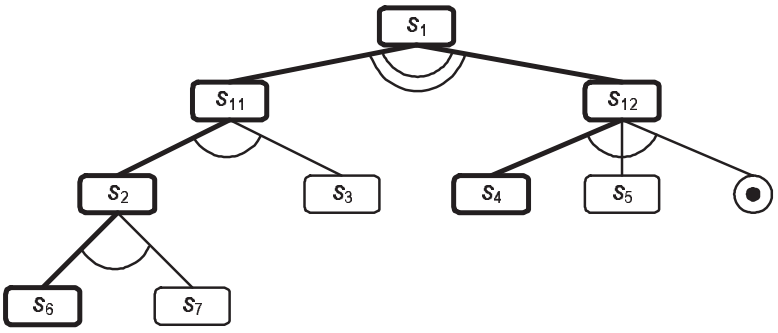
Definicja 6.9. Domknięciem ku dołowi *(*ang. *downward closure) zbioru stanów zgodnych* $S \subseteq S_z$*, oznaczanym jako* $downclos\,(S)$*, jest najmniejszy nadzbiór zbioru* S*, którego elementy spełniają przynajmniej jeden z poniższych czterech (1, 2a, 2b, 2c) warunków:*

1. $(s \in downclos\,(S) \wedge type_z\,(s) = AND) \Rightarrow hrc_z\,(s) \subseteq downclos\,(S)$,
2. $s \in downclos\,(S) \wedge type_z\,(s) = OR \wedge hrc_z\,(s) \neq \emptyset \wedge hrc_z\,(s) \cap S = \emptyset \wedge$

   (a) $\wedge \left(\underset{s' \in hrc_z(s)}{\forall} history\,(s') = false \right) \Rightarrow default_z\,(s) \in downclos\,(S)$,

   (b) $\wedge \left(\underset{i}{\forall} s \notin S^i_{exit} \right) \Rightarrow default_z\,(s) \in downclos\,(S)$,

   (c) $\wedge \left(\underset{s' \in hrc_z(s)}{\forall} history\,(s') = true \right) \wedge \left(\underset{i}{\exists}\, s \in S^i_{exit} \right) \wedge \left(\underset{i<m<k}{\forall} s \notin S^m_{exit} \right) \wedge$
   $\wedge \left(s' \in S^i_{exit} \right) \Rightarrow s' \in downclos\,(S)$.



Rys. 6.4. Domknięcie ku dołowi stanów s_1 i s_4 i zarazem konfiguracja początkowa dla diagramu z rysunku 6.1

W przykładzie z rysunku 6.1 $downclos\,(\{s_1, s_4\}) = \{s_1, s_{11}, s_{12}, s_2, s_4, s_6\}$. Warunek 1 określa przynależność do domknięcia wszystkich stanów bezpośrednio podległych stanowi współbieżnemu. Wyrażenie w warunku 2 określa stan złożony o typie *AND*. Warunki 2a, 2b i 2c są przedłużeniem warunku 2. Warunek 2a

określa przynależność do konfiguracji stanu początkowego automatu bez historii. Warunek 2b określa przynależność do konfiguracji stanu początkowego automatu z historią w sytuacji, gdy automat nie był jeszcze aktywowany. W obu tych przypadkach do domknięcia należą stany początkowe automatów podległych. Zapis S^i_{exit}oznacza zbiór stanów, które utraciły sterowanie w i-tej iteracji. Warunek 2b ustala przynależność do konfiguracji ostatnio aktywnego stanu automatu sekwencyjnego z atrybutem historii. Zmienna k w warunku oznacza aktualną iterację.

Rysunek 6.4 przedstawia domknięcie ku dołowi dla zbioru stanów $\{s_1, s_4\}$. Linią pogrubioną zaznaczono stany należące do domknięcia. Przedstawiony przypadek jest jednocześnie konfiguracją początkową, gdyż $downclos\,(\{s_1, s_4\})$ jest równe $downclos\,(\{root_z\})$.

6.2. Semantyka diagramów

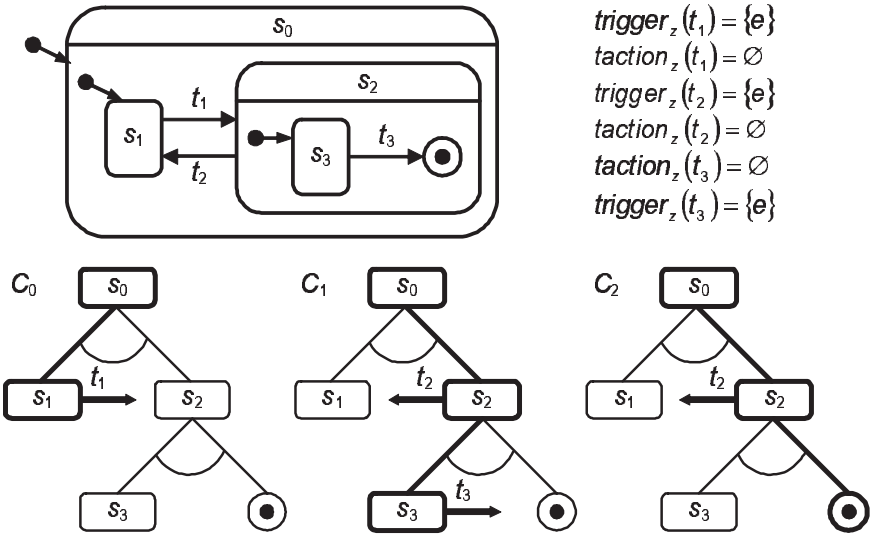
Definicje przedstawione w rozdziale 6.1 określają zasady przynależności diagramów do pewnej klasy **Z** zgodnej z definicją 6.4. Poniżej zostaną wprowadzone pojęcia i reguły ustalające zachowanie diagramów. Głównym omawianym pojęciem jest pojęcie kroku, rozumiane jako przejście od jednej konfiguracji do następnej.

Definicja 6.10. *Dla danej tranzycji $t \in T_z$ i mikrokonfiguracji $\mu C = (S_{exit}, S_{curr}, E_{curr})$, t jest* **związana** *(*ang. *relevant) z μC jeżeli $out_z\,(t) \in S_{curr}$,* **wzbudzana** *(*ang. *triggered) w μC, jeżeli $trigger_z\,(t) \subseteq E_{curr}$.*

W przykładzie z rysunku 6.5 tranzycją związaną z konfiguracją C_0 jest t_1. Jeżeli przy tej konfiguracji wystąpi zdarzenie e, wówczas t_1 będzie nie tylko związana z C_0, ale będzie również wzbudzana i zostanie zrealizowana. Podobna sytuacja zachodzi dla tranzycji t_3 w konfiguracji C_1. Nieco inaczej natomiast wygląda sytuacja dla tranzycji t_2 w konfiguracji C_1. Warunki związania i wzbudzania są warunkami koniecznymi, lecz niewystarczającymi do realizacji tranzycji. Do zrealizowania tej tranzycji konieczne jest również spełnienie tzw. wewnętrznego warunku wywłaszczenia sterowania ze stanu początkowego (spełnienie dla $out_z\,(t_2)$).

Definicja 6.11. Wewnętrznym warunkiem wywłaszczenia sterowania ze stanu *s, dla zbioru stanów bieżących S_{curr}, jest wyrażenie oznaczane jako $intcond(s, S_{curr})$, które przyjmuje wartość* ***true****, gdy* $\underset{s' \in hrc^*_z(s)}{\forall} endst_{s'} \in hrc_z\,(s') \Rightarrow endst_{s'} \in S_{curr}$*, a w przeciwnym razie przyjmuje wartość* ***false****.*

Spełnienie wewnętrznego warunku wywłaszczenia jest niezbędne, aby sterowanie mogło opuścić stan i sprowadza się do sytuacji, gdy wszystkie podległe stany końcowe, jeżeli zostały zadeklarowane, należą do zbioru stanów bieżących. Tranzycja t_2 związana z konfiguracją C_1, mimo wystąpienia zdarzenia e, będzie wzbudzana, lecz nie zostanie zrealizowana, gdyż wewnętrzny warunek wywłaszczenia sterowania z jej stanu początkowego s_2 nie jest spełniony. Aby warunek wywłaszczenia został spełniony automat sekwencyjny, ze stanem akceptującym podległy stanowi s_2, musi przejść do stanu akceptującego. W konfiguracji C_1 stan końcowy nie jest aktywny. Tranzycja t_2 może zostać zrealizowana dopiero przy konfiguracji C_2 i wystąpieniu zdarzenia e.



Rys. 6.5. Diagram statechart oraz konfiguracje ze związanymi tranzycjami

Warunki, których spełnienie jest niezbędne, aby tranzycja mogła zostać zrealizowana są określone w definicji 6.12.

Definicja 6.12. *Tranzycja jest* **gotowa do zrealizowania** *(*ang. *enabled) w mikrokonfiguracji* $\mu C = (S_{exit}, S_{curr}, E_{curr})$ *jeżeli:*

1. *t jest związana z* μC*,*
2. $intcond\left(out_z\left(t\right), S_{curr}\right) = true$*,*
3. *t jest wzbudzana w* μC*.*

Definicja 6.13. Mikrokrokiem $\mu\Sigma$*, przy mikrokonfiguracji* μC*, jest niepusty zbiór tranzycji zawierający się w* T_z*, spełniający następujące warunki:*

1. $\underset{t \in \mu\Sigma}{\forall}$ *t jest gotowa do zrealizowania w* μC*,*
2. $\mu\Sigma$ *jest strukturalnie zgodny.*

Mikrokrok, będący składową kroku, jest zbiorem tranzycji transformujących statechart z jednej mikrokonfiguracji do następnej. Dla sytuacji początkowej diagramu przedstawionego na rysunku 6.1 $\mu\Sigma = \{t_1, t_4\}$. Stany początkowe obu tranzycji, odpowiednio s_6 i s_4, są aktywne (związane z bieżącą mikrokonfiguracją), wewnętrzne warunki wywłaszczenia sterowania są prawdziwe oraz występuje zdarzenie e_3, wzbudzające te tranzycje.

Definicja 6.14. *Dla danej mikrokonfiguracji* $\mu C = (S_{exit}, S_{curr}, E_{curr})$*, mikrokroku* $\mu\Sigma \subseteq T_z$ *i tranzycji* $t \in \mu\Sigma$ *związanej z* μC*, zachodzą następujące zależności:*

1. $exited\,(t, S_{curr}) = \{s \in S_z | s \in hrc_z^*\,(out_z\,(t)) \cap S_{curr}\}$,
2. $entered\,(t) = downclos\,(in_z\,(t))$,
3. $exitedev\,(t, S_{curr}) = \bigcup_{s \in (exited(t, S_{curr}) \setminus entered(t))} exit_z\,(s)$,
4. $enteredev\,(t, S_{curr}) = \bigcup_{s \in (entered(t) \setminus exited(t, S_{curr}))} entry_z\,(s)$.

Wówczas **mikrokonfiguracja** $nextt\mu C$, **osiągnięta poprzez wykonanie mikrokroku** $\mu\Sigma$ *przy bieżącej mikrokonfiguracji* μC, *określona jest następująco:*

5. $next\mu C\,(\mu\Sigma, \mu C) = (S'_{exit}, S'_{curr}, E'_{curr})$ *gdzie:*
   (a) $S'_{exit} = S_{exit} \cup \bigcup_{t \in \mu\Sigma} exited\,(t, S_{curr})$,
   (b) $S'_{curr} = \left(S_{curr} \setminus \bigcup_{t \in \mu\Sigma} exited\,(t, S_{curr})\right) \cup \left(\bigcup_{t \in \mu\Sigma} entered\,(t)\right)$,
   (c) $E'_{curr} = \bigcup_{t \in \mu\Sigma} taction_z\,(t) \cup \bigcup_{t \in \mu\Sigma} enteredev\,(t, S_{curr}) \cup$
   $\cup \bigcup_{t \in \mu\Sigma} exitedev\,(t, S_{curr}) \cup \bigcup_{s \in S'_{curr}} do_z\,(s)$.

Dla mikrokonfiguracji $\mu C = (S_{exit}, S_{curr}, E_{curr})$, przy realizacji tranzycji t, zbiory $exited\,(t)$ i $entered\,(t)$ określają odpowiednio stany, które utraciły sterowanie i stany, które otrzymały sterowanie. Zbiory $exitedev\,(t, S_{curr})$ i $enteredev(t, S_{curr})$ określają zdarzenia, które są związane odpowiednio ze stanami, które utraciły sterowanie (i w następnej iteracji już nie są aktywne) i które otrzymały sterowanie. Zapis $next\mu C\,(\mu\Sigma, \mu C)$ określa mikrokonfigurację uzyskaną poprzez realizacją mikrokroku $\mu\Sigma$ przy mikrokonfiguracji μC. W realizacji układowej konsekwencją zapisu 5c jest fakt, że zdarzenia generowane w bieżącym mikrokroku, są dostępne dla układu przy następnym takcie zegara.

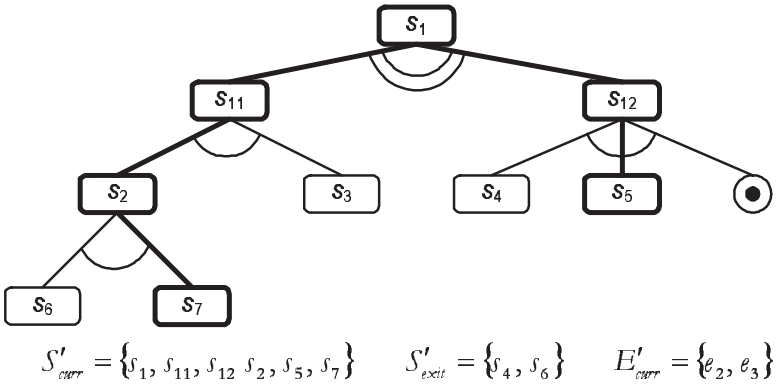
Jako ilustracja definiowanego pojęcia niech posłuży mikrokrok $\mu C = (S_{curr}, S_{exit}, E_{curr})$ z konfiguracji początkowej, przedstawionej na rysunku 6.4. Zbiory stanów bieżących, stanów opuszczonych przez sterowanie i zbiór zdarzeń aktualnie dostępnych w systemie są następujące: $S_{curr} = \{s_1, s_{11}, s_{12}, s_2, s_4, s_6\}$, $S_{exit} = \emptyset$, $E_{curr} = \{e_3\}$. Tranzycjami gotowymi do zrealizowania są: $\mu\Sigma = \{t_1, t_4\}$. Pomocnicze zbiory stanów i zdarzeń przy tak określonym przejściu, są zawarte w tabeli 6.1.

Mikrokonfiguracja następna $\mu C' = (S'_{curr}, S'_{exit}, E'_{curr})$, uzyskana w wyniku takiego przejścia $next\mu C\,(\mu\Sigma, \mu C) = \mu C'$, określona jest przez następujące zbiory: $S'_{curr} = \{s_1, s_{11}, s_{12}s_2, s_5, s_7\}$, $S'_{exit} = \{s_4, s_6\}$, $E'_{curr} = \{e_2, e_3\}$. W wyniku przejścia, stany s_6 i s_4 utraciły sterowanie, stany s_5 i s_7 otrzymały sterowanie. Diagram hierarchii układu wraz z zaznaczonymi stanami aktywnymi, uzyskanymi w wyniku realizacji mikrokroku przedstawiony jest na rysunku 6.6.

Definicja 6.15. *Domknięcie ku dołowi wyróżnionego stanu* $root_z$ – $downclos\,(\{root_z\})$, *dla warunków początkowych jest nazywane* **konfiguracją początkową** C_0 *(*ang. *default configuration).*

Tab. 6.1. Zbiory stanów i zdarzeń dla mikrokonfiguracji początkowej

Zbiory stanów i zdarzeń	t_1	t_4
exited	s_6	s_4
entered	s_7	s_5
exitedev	$\emptyset$	$\emptyset$
enteredev	$\emptyset$	$\emptyset$



Rys. 6.6. Mikrokonfiguracja dla diagramu z rysunku 6.1 uzyskana po pierwszym mikrokroku

Definicja 6.16. *Dla danego otoczenia* **krokiem** *(*ang. *step), również zwanym* **krokiem dopuszczalnym** *(*ang. *admissible step), oznaczanym jako* Σ*, z konfiguracji* C *do konfiguracji* C'*, nazywana jest sekwencja mikrokroków* $\langle \mu\Sigma_0, \ldots \mu\Sigma_n \rangle$*, która spełnia następujące warunki:*

1. $\mu C_0 = (\emptyset, C, E)$ *jest mikrokonfiguracją początkową,*
2. $\mu\Sigma_i$ *jest mikrokrokiem z mikrokonfiguracji* $\mu C_i = \left(S^i_{exit}, S^i_{curr}, E^i_{curr}\right)$ *dla* $0 \leqslant i \ \leqslant n$*,*
3. $\mu C_{i+1} = next\mu C\left(\mu\Sigma_i, \mu C_i\right)$ *dla* $0 \leqslant i \leqslant n$*,*
4. $\forall t \in T_z$ *t nie jest gotowa do zrealizowania w* $\mu C_{i+1} = \left(S^{i+1}_{exit}, S^{i+1}_{curr}, E^{i+1}_{curr}\right)$*,*
5. $C' = S^{n+1}_{curr}$*, oznaczane jest* $nextC\left(\Sigma\right)$*.*

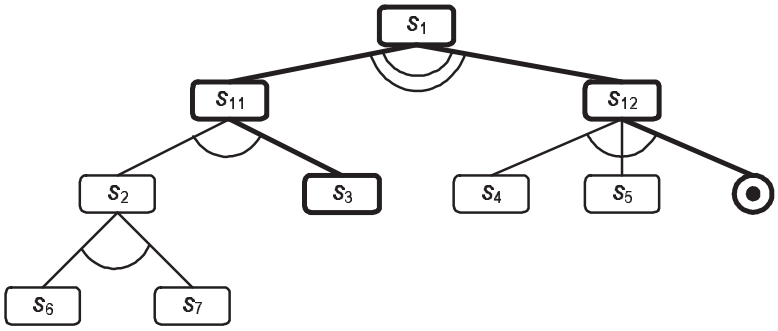
Zakłada się, że w trakcie realizacji kroku do układu nie dochodzą zdarzenia ze świata zewnętrznego, które mogłyby wpływać na zachowanie układu. Punkt 4 w definicji 6.16 mówi, że warunkiem zakończenia kroku jest brak gotowych do zrealizowania tranzycji w układzie.

Definicja 6.17. Konfiguracją bieżącą, *jest nazywana konfiguracja uzyskana w wyniku iteracyjnej realizacji kroków (lub mikrokroków), począwszy od konfiguracji początkowej.*

Tab. 6.2. Stany, zdarzenia i tranzycje w realizacji kroku

t_i	S^i_{exit}	S^i_{curr}	E^i_{curr}	$\mu\Sigma_i$
$t_{i=0} \in (250, 350)$	$\emptyset$	$\{s_1, s_{11}, s_{12}, s_2, s_4, s_6\}$	$\{e_3\}$	$\{t_1, t_4\}$
$t_{i=1} \in (350, 450)$	$\{s_1, s_4\}$	$\{s_1, s_{11}, s_{12}, s_2, s_5, s_7\}$	$\{e_2, e_3, e_4\}$	$\{t_3\}$
$t_{i=2} \in (450, 550)$	$\{s_2, s_7\}$	$\{s_1, s_{11}, s_{12}, s_3, s_5\}$	$\{e_1, e_2, e_3\}$	$\{t_5\}$
$t_{i=3} \in (550, 650)$	$\{s_5\}$	$\{s_1, s_{11}, s_{12}, s_3, endst\}$	$\{e_2, e_3\}$	$\emptyset$

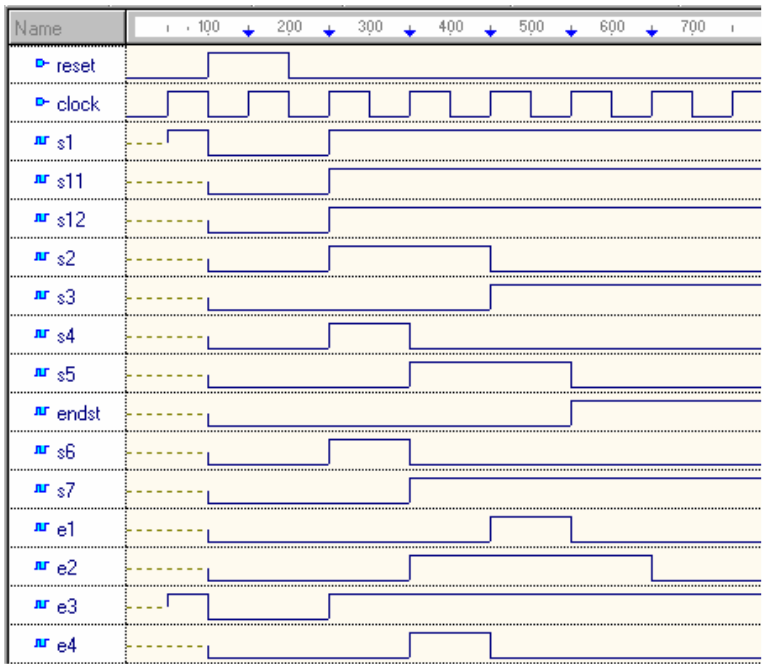
Diagram przedstawiony na rysunku 6.1 ma charakter układu autonomicznego, tzn. przyczyną sekwencji mikrokroków nie są zdarzenia dochodzące z otoczenia, lecz zdarzenie e_3 związane z aktywnością stanu początkowego s_1. Działanie układu składa się tylko z jednego kroku. Od konfiguracji początkowej układ przechodzi do następnej konfiguracji, w której kończy swoje działanie, bez względu na zdarzenia dochodzące do układu. Sekwencja mikrokonfiguracji oraz realizowane kolejne tranzycje są zawarte w tabeli (tab. 6.2). Jak to wynika z definicji 6.16, krok jest zakończony w momencie, gdy zbiór gotowych do realizacji tranzycji związanych z kolejną mikrokonfiguracją jest zbiorem pustym. Sytuacja taka ma miejsce w chwili t_3. Wartości czasowe w tabeli odnoszą się do przebiegów z rysunku 6.8. Krok układu składa się z trzech mikrokroków $\Sigma = (\mu\Sigma_0\ ,\mu\Sigma_1, \mu\Sigma_2)$, a na uzyskaną konfigurację końcową składają się stany $next\mu C(\Sigma) = \{\ s_1, s_{11}, s_{12}, s_3, endst_{s_{12}}\}$. Rysunek 6.7 przedstawia diagram hierarchii uzyskanej konfiguracji końcowej.



Rys. 6.7. Konfiguracja końcowa

Równocześnie z pracami nad podstawami teoretycznymi diagramów statechart, autor opracował środowisko projektowe, system *HiCoS*, realizujące opis diagramów w języku *VHDL*. Rysunek 6.8 przedstawia przebiegi czasowe układu zamodelowanego diagramem z rysunku 6.1, realizującego krok (tab. 6.2). Otrzymany model układu cyfrowego reaguje na narastające zbocze sygnału zegarowego

i rozpoczyna swoje działanie od chwili $t = 250$. Symulację przeprowadzono w środowisku *Active HDL* firmy *ALDEC*.



Rys. 6.8. Przebiegi czasowe działania modelowanego układu

Dotychczas omawiany model matematyczny nie uwzględnia sposobów technicznej realizacji mechanizmu pamięci stanu, co najwyżej mówi, że stan może posiadać pamięć (def. 6.4, punkt 5). W implementacji układowej diagramów z każdym stanem jest związany przerzutnik, którego wzbudzenie dla stanów z atrybutem historii może oznaczać aktywność lub pamiętanie, że dany stan jest tak zwanym stanem ostatnio aktywnym. Z tego względu, w celu jednoznacznego ustalenia aktywności stanu należy wziąć pod uwagę całą ścieżkę, prowadzącą od rozpatrywanego stanu do stanu $root_z$. Poniższa definicja wprowadza pojęcie aktywności stanu.

Definicja 6.18. *Lokalny stan s jest* **stanem aktywnym** *jeżeli ścieżka $path\,(root_z, s)$ należy do konfiguracji bieżącej.*

Dobrą ilustracją pojęcia stanu aktywnego jest stan s_7. Na przebiegu czasowym (rys. 6.8), od chwili $t = 350$ do chwili $t = 450$, stan s_7 jest stanem aktywnym – należy do domknięcia (rys. 6.6). Od momentu $t = 450$, s_7 przestaje być stanem aktywnym, lecz jako stan z historią, jest stanem ostatnio aktywnym automatu podległego stanowi s_2, czyli przerzutnik związany z s_7 nadal jest wzbudzony. Sytuacja ta ma również miejsce w konfiguracji końcowej (rys. 6.7), przerzutnik z nim

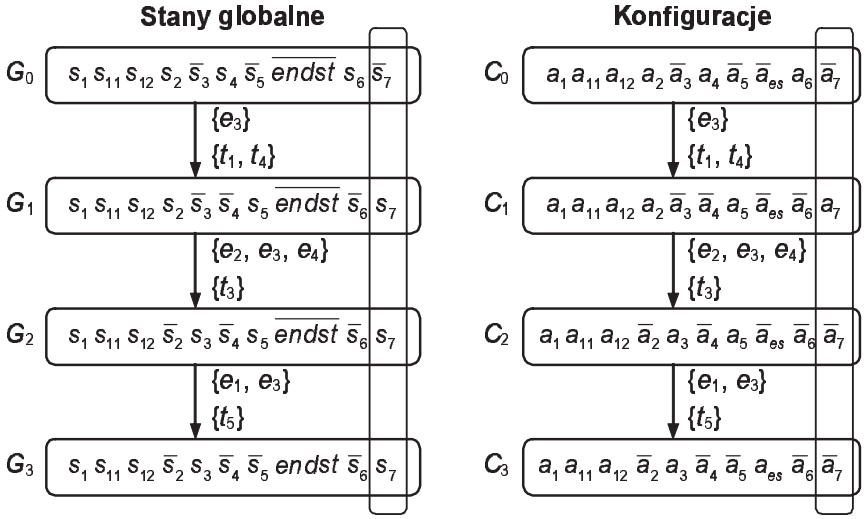
związany jest wzbudzony, lecz s_7 nie należy do domknięcia konfiguracji końcowej i przez co nie jest już stanem aktywnym.

Z realizacją układową można związać pojęcie stanu globalnego układu rozumianego następująco:

Definicja 6.19. Stanem globalnym G *nazywany jest zbiór stanów wszystkich przerzutników w systemie, związanych zarówno ze stanami lokalnymi, jak i ze zdarzeniami.*

Zachowanie układu cyfrowego może zostać przedstawione za pomocą grafu osiągalności. Rysunek 6.9 przedstawia graf osiągalności dla wszystkich stanów globalnych, w których może znaleźć się układ przedstawiony na rysunku 6.1. Dane uzyskano z autorskiego systemu *HiCoS*, stosując algorytmy przedstawione w publikacji (Łabiak, 2001*c*). Symbole w węzłach grafu oznaczają wzbudzenie przerzutników, związanych ze stanami układu. Na przykład s_1 oznacza jedynkę logiczną na wyjściu przerzutnika związanego ze stanem s_1, natomiast $\overline{s_7}$ oznacza zero logiczne na wyjściu przerzutnika odpowiadającego stanowi s_7. Skierowane krawędzie między wierzchołkami, oznaczają przejścia od jednego stanu globalnego układu do następnego stanu. Etykiety przy krawędziach opisują, jakie zdarzenia towarzyszą przejściom i jakie tranzycje są realizowane przy przejściu. Rysunek 6.9, obok grafu osiągalności dla stanów globalnych, przedstawia graf osiągalności dla wszystkich możliwych konfiguracji układu, czyli zbioru stanów aktywnych. W grafie tym, symbole oznaczają aktywność stanu. Przykładowo, a_1 oznacza, że w danej konfiguracji stan s_1 jest aktywny, natomiast $\bar{a}_7$ oznacza, że stan s_7 nie jest aktywny. Różnice między grafami występują tylko dla stanów z atrybutem historii. Przykładem jest stan s_7, który jest aktywny tylko w konfiguracji C_1, podczas gdy przerzutnik z nim związany pamięta jego aktywność dodatkowo w stanach globalnych G_2 i G_3. Konsekwencją tego faktu jest generowanie zdarzenia e_4 tylko wtedy, gdy s_7 jest aktywny, czyli w stanie globalny G_1 (rys. 6.8).

Zachowanie układu cyfrowego zamodelowanego diagramami, a w szczególności układu z rysunku 6.1, cechuje się dopuszczeniem występowania tranzycji będących w konflikcie. Na przykład w diagramie na rysunku 6.1, gdyby zdarzenia e_2 i e_3 były wejściem systemu, mogłaby zaistnieć taka sytuacja, że tranzycje t_1 i t_3 mogłyby być jednocześnie gotowe do realizacji. Z kolei definicja kroku (def. 6.16), u podstaw której leżą definicje 6.8 i 6.8, mikrokonfiguracji (def. 6.8) i mikrokroku (def. 6.13), mówi o kroku, jako o jednym z wielu dopuszczalnych ciągów zbiorów realizowanych tranzycji, określonych, odpowiednio zbiorami maksymalnie zgodnych stanów bieżących (def. 6.6) i zbiorami tranzycji strukturalnie zgodnych (def. 6.5). Zatem sama definicja kroku wprowadza zachowanie niedeterministyczne, gdyż w danym momencie czasu może zaistnieć więcej niż jeden gotowych do realizacji mikrokroków. W realizacji diagramów jako układ cyfrowy, taka sytuacje jest niepożądana, lecz podobnie jak to zostało napisane o diagramach statechart w pracy (Lavagno i in., 1998), kwestie determinizmu układu we wczesnym etapie jego modelowania, prowadzą do zawiłości w tworzonym opisie, stąd w podobny sposób, postępowanie z determinizmem diagramów zostało w tym rozdziale pominięte (podrozdział 7.3).



Rys. 6.9. Grafy osiągalności stanów globalnych i konfiguracji

6.3. Podsumowanie

W rozdziale formalnie zdefiniowano binarny modularny model diagramów statechart, tzn. taki, w którym operuje się wartościami binarnymi i zabrania się tranzycjom przekraczania granic stanów. Ograniczenie, co do operowania przez diagramy tylko dwoma wartościami logicznymi wynika z faktu, że do zastosowań dotyczących sterowania rozpatrywanego w pracy, takie założenie jest wystarczające, a zarazem istotnie upraszcza problematykę syntezy. Modularność z kolei jest wynikiem założenia o podziale badań na etapy (podrozdział 3.3). W przedstawionym modelu pominięto takie elementy, jak tranzycje czasowe czy ścieżki obliczonych przejść, które sprawiają istotne trudności realizacyjne. W przyszłości planuje się rozszerzenie modelu o tranzycje przekraczające granice stanów i tranzycje złożone (punkt 3.3.5) oraz o stany synchronizujące (punkt 3.3.7), dążąc przy tym do zachowania zgodności z technologią *UML*. Sformułowane w rozdziale składnia i semantyka stanowią podstawę realizacji układowej, szczegółowo opisane w rozdziale następnym.

Rozdział 7

SYNTEZA DIAGRAMÓW STATECHART METODĄ BEZPOŚREDNIEGO ODWZOROWANIA

Rozdział zawiera opis autorskiej koncepcji układowej realizacji modelu zachowania specyfikowanego diagramami statechart. Wszystkie informacje zawarte w tym rozdziale: definicje, reguły, rysunki i przykłady są wyłącznym wynikiem pracy autora.

7.1. Interpretowany diagram statechart i pojęcia pomocnicze

Aby opisywany diagram, formalnie zdefiniowany w rozdziale poprzednim, mógł służyć jako środek specyfikacji behawioralnej sterowników cyfrowych, konieczne jest określenie fizycznej interpretacji takich pojęć abstrakcyjnych jak zdarzenie, zbiór zdarzeń czy etykieta. Poniższa definicja określa model interpretowanego diagramu statechart, w którym zamiast pojęć abstrakcyjnych wprowadza się takie rzeczywiste pojęcia jak sygnał czy wyrażenie logiczne. Pojęcie interpretowanego diagramu statechart stanowi bezpośrednie połączenie między ludzkim postrzeganiem opisywanego zachowania, a możliwościami implementacyjnymi w strukturach programowalnych. W oparciu o tę definicję możliwe jest, wykorzystanie diagramów do projektowania układów cyfrowych.

Definicja 7.1. Interpretowanym diagramem statechart *jest diagram jak w definicji 6.4, gdzie:*

1. $X \subseteq E_z$ *jest zbiorem zdarzeń przychodzących ze środowiska,* $Y \subseteq E_z$ *jest zbiorem zdarzeń widocznych dla środowiska. Zbiory zdarzeń* X *i* Y *są rozłączne.*

2. *Zdarzenie, które może być obecne lub nie, jest sygnałem.* I *jest zbiorem wszystkich sygnałów w układzie, dla* $i \in I$ *i oznacza, że zdarzenie jest obecne,* $!i$*, że zdarzenie nie występuje.*

3. $input : X \leftrightarrow I_X$*, gdzie* $I_X \subseteq I$ *– jest funkcją przyporządkowującą wzajemnie jednoznacznie sygnałowi zdarzenie przychodzące ze środowiska.*

4. $output : Y \leftrightarrow I_Y$*, gdzie* $I_Y \subseteq I$ *– jest funkcją przyporządkowującą wzajemnie jednoznacznie sygnałowi zdarzenie widoczne dla środowiska.*

5. *$I_X \cap I_Y = \emptyset$ – sygnały związane ze zdarzeniami przychodzącymi z zewnątrz i widoczne na zewnątrz są odpowiednio wejściem i wyjściem z układu. Zbiory sygnałów są rozłączne.*

6. *Składnik $trigger_z\,(t)$ funkcji etykietowania tranzycji $tlabel_z$, nazywany predykatem, jest wyrażeniem boolowskim, tworzonym według następującej gramatyki:*

$$g \quad ::= \quad true \quad | \quad false \quad | \quad i \quad | \quad !g \quad | \quad g + g \quad | \quad g * g \quad | \quad (g),$$

gdzie $i \in I$ jest sygnałem związanym ze zdarzeniem $e \in E_z$. Wartość sygnału jest true*, gdy zdarzenie jest obecne lub* false*, gdy jest nieobecne. Operatory* !*, $+$ oraz $*$ odpowiadają odpowiednio logicznym operatorom negacji, sumy i iloczynu.*

7. *Zbiory zdarzeń, związane z funkcjami $taction_z(t)$ oraz $saction_z(t)$, określone są w następujący sposób:*

$$A \quad ::= \quad nil \quad | \quad b \qquad\qquad b \quad ::= \quad i \quad | \quad b, b$$

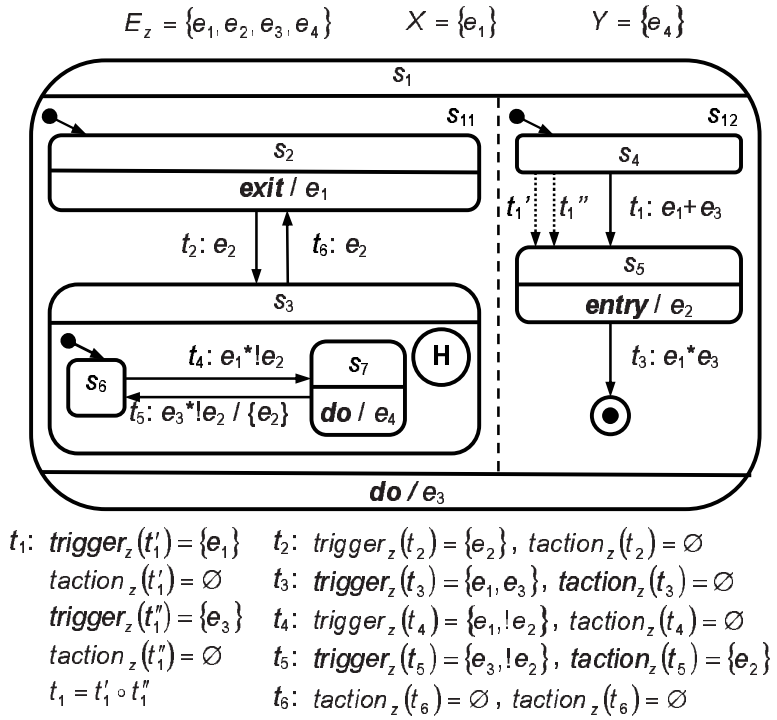
gdzie $A \subseteq I$, $i \in I$ a „,” oznacza dwa różne sygnały.

Rysunek 7.1 przedstawia interpretowany diagram statechart. Sygnałem docierającym do układu ze świata zewnętrznego – czyli jego wejściem – jest sygnał e_1, sygnałem widocznym dla otoczenia diagramu – stanowiącym wyjście z układu – jest sygnał e_4. Pozostałe sygnały mają charakter lokalny. W dolnej części rysunku zamieszczono opisy tranzycji zgodne z modelem matematycznym (def. 6.4). składniku $trigger_z$, dla zaznaczenia faktu, że zdarzenie nie występuje, stosuje się zdarzenia zanegowane (np. $!e_2$). Opisy tranzycji w górnej części rysunku, zamieszczone przy strzałkach (składnik $trigger_z$), są wyrażeniami boolowskimi. Wyrażenia te odpowiadają zbiorom zdarzeń w sposób następujący: wyrażenie boolowskie przyjmuje wartość *true*, gdy wystąpią wszystkie zdarzenie wymienione w opisującym je zbiorze (wartość określona przez funkcję $trigger_z$ w definicji 6.4). Na przykład dla tranzycji t_3 jest to wystąpienie zdarzeń e_1 i e_3. Tranzycja t_1 jest przykładem złożenia dwóch tranzycji t_1' i t_1''. Złożenia tranzycji można dokonać tylko wtedy, gdy zbiory zdarzeń generowanych przy ich realizacjach są takie same. Wówczas wyrażenie reprezentujące cześć $trigger_z$ tranzycji złożonej, odpowiada dwóm zbiorom zdarzeń (jeden związany z t_1' a drugi z t_1'') i jest sumą logiczną predykatów tranzycji składowych. Przykładem zbioru akcji generowanej przy realizacji tranzycji jest $taction\,(t_5) = \{e_2\}$. Przykładami zbiorów akcji związanych ze stanami są zbiory: akcji wejściowej – $entry\,(s_5) = \{e_2\}$, akcji wyjściowej – $exit\,(s_2) = \{e_1\}$ i akcji statycznej – $do\,(s_7) = \{e_4\}$.

Do sporządzenia opisu równaniami logicznymi konieczne jest operowanie pojęciem zbioru tranzycji wejściowych do stanu i wyjściowych ze stanu.

Definicja 7.2. Zbiorem bezpośrednich tranzycji wejściowych do stanu s *jest zbiór określony w następujący sposób:*

$${}^\bullet s = \{t \in T_z | in_z\,(t) = s\}\,.$$



Rys. 7.1. Interpretowany statechart wraz z opisem tranzycji

Definicja 7.3. Zbiorem bezpośrednich tranzycji wyjściowych ze stanu s *jest zbiór określony następująco:*

$$s^{\bullet} = \{t \in T_z | out_z(t) = s\}.$$

Przykładem zbioru bezpośrednich tranzycji wejściowych dla stanu s_6 jest ${}^{\bullet}s_6 = \{t_5\}$, a przykładem zbioru bezpośrednich tranzycji wyjściowych jest $s_6^{\bullet} = \{t_4\}$.

W implementacji układowej z każdym stanem jest związany jeden przerzutnik i wszelkie zmiany stanu układu odbywają się wyłącznie w rytm taktów sygnału zegarowego, czyli układ cyfrowy jest układem synchronicznym. Wartość **1** na wyjściu przerzutnika informuje o tym, że stan związany z danym przerzutnikiem, jest aktywny lub, że jest tzw. stanem ostatnio aktywnym (co ma miejsce w przypadku stanów z nadanym atrybutem historii). Zatem na podstawie stanu tylko jednego przerzutnika nie można jednoznacznie ustalić, czy stan jest rzeczywiście aktywny, czy tylko jest pamiętane, że był on ostatnio aktywny. Aby ustalić aktywność stanu, należy zbadać wyjścia przerzutników związanych ze wszystkimi stanami nadrzędnymi. Innymi słowy, stan jest aktywny wtedy i tylko wtedy, gdy aktywne są wszystkie stany nadrzędne (tak jak to wynika z definicji 6.18), co prowadzi do zdefiniowania następującego warunku aktywności stanu.

Definicja 7.4. Warunek aktywności stanu, *oznaczany jako activecond(s), obliczany jest następująco:*

$$activecond\,(s) = \prod_{s_i \in path(root_z, s)} s_i.$$

W dalszej części pracy nazwa stanu używana w definicjach i wyrażeniach odnosi się do wyjścia przerzutnika, związanego ze stanem. Stan s_7 jest przykładem stanu z atrybutem historii, a jego warunek aktywności jest następujący: $activecond\,(s_7) = s_1 * s_{11} * s_3 * s_7$.

Celem wprowadzenie do syntaktyki diagramów stanu końcowego, jest dostarczenie środka, dzięki któremu można opisywać procesy, które muszą zostać zrealizowane do końca (czyli do osiągnięcia stanu końcowego). Automatu ze stanem końcowym, modelującego taki proces, nie można pozbawić sterowania poprzez realizację tranzycji wyższych poziomów hierarchii. Inaczej mówiąc, osiągnięcie stanu końcowego ma wyższy priorytet, niż tranzycje wyższego poziomu hierarchii. Zatem opisując warunki realizacji tranzycji, należy między innymi wziąć pod uwagę stany końcowe w automatach podległych stanowi, z którego tranzycja bierze swój początek. Do reprezentowania opisanej sytuacji służy wewnętrzny warunek wywłaszczenia sterowania ze stanu (definicja 6.11).

Definicja 7.5. Wewnętrzny warunek wywłaszczenia sterowania *dla danego stanu s (*ang. *control pre-emptive internal condition), oznaczany jako intcond(s), obliczany jest następująco:*

1. $hrc_z(s) = \emptyset$ – *stan s jest stanem prostym:*

$$intcond(s) = \prod_{s_i \in hrc_z(s)} \left(\overline{activecond\,(s_i)} + activecond\,(s_i) * intcond\,(\mathrm{s_i}) \right),$$

2. $type_z\,(s) = OR \wedge hrc_z\,(s) \neq \emptyset \wedge endst_s \in hrc_z\,(s)$ – *stan s jest stanem nadrzędnym automatu sekwencyjnego bez stanu końcowego:*

$$intcond(s) = \prod_{s_i \in hrc_z(s)} \left(\overline{activecond\,(s_i)} + activecond\,(s_i) * intcond\,(\mathrm{s_i}) \right),$$

3. $type_z\,(s) = AND \wedge hrc_z\,(s) \neq \emptyset$ – *stan s jest stanem nadrzędnym automatu sekwencyjnego ze stanem końcowym:*

$$intcond\,(s) = activecond\,(endst_s)\,,$$

4. $type_z\,(s) = AND \wedge hrc_z\,(s) \neq \emptyset$ – *stan s jest stanem nadrzędnym automatu współbieżnego:*

$$intcond\,(\mathrm{s}) = \prod_{s_i \in hrc_z(s)} intcond\,(\mathrm{s_i}).$$

Punkt 1 z definicji 7.5 mówi, że stan prosty można zawsze wywłaszczyć, np. $intcon\,(\mathrm{s}_7) = 1$. Natomiast, aby wywłaszczyć automat sekwencyjny bez stanu końcowego podległy stanowi s (def. 7.5 punkt 2), wszystkie jego stany składowe ($s_i \in hrc_z\,(s)$) jednocześnie muszą spełniać wyrażenie:

$$\overline{activecond\,(s_i)} + activecond\,(s_i) * \mathrm{intcond}\,(s_i)\,.$$

Składowa $\overline{activecond\,(s_i)}$ oznacza, że jeżeli podległy stan s_i nie jest aktywny, to wyrażenie przyjmuje wartość **1** i automaty podległe stanowi s_i nie wpływają na wewnętrzny warunek wywłaszczenia sterowania ze stanu s. Jeżeli s_i jest stanem aktywnym, wówczas warunek wywłaszczenia stanu s jest zależny od wewnętrznego warunku wywłaszczenie sterowania stanu s_i. Dla stanu s_3 warunek ten wygląda następująco:

$$\begin{aligned} intcond\,(s_3) = \quad & \left(\overline{activecond\,(s_6)} + activecond\,(s_6) * intcond\,(s_6)\right) * \\ & * \left(\overline{activecond\,(s_7)} + activecond\,(s_7) * intcond\,(s_7)\right) = 1. \end{aligned}$$

Jeżeli podległym automatem jest automat ze stanem końcowym (def. 7.5 punkt 3), to wówczas wobec przyjętych założeń o stanie końcowym, wywłaszczenie jest zależne tylko od aktywności stanu końcowego. Przykładem takiego stanu jest stan s_{12} a jego $intcond\,(s_{12}) = activecond\,(endst_{s_{12}})$. Ostatnim możliwym przypadkiem jest sytuacja, gdy podległym automatem jest automat współbieżny (def. 7.5 punkt 4). W takiej sytuacji wywłaszczenie jest możliwe, jeżeli warunki wywłaszczenia są jednocześnie spełnione we wszystkich podległych automatach składowych. Przykładem takiego stanu jest s_1:

$$\begin{aligned} intcond\,(s_1) = \quad & intcond\,(s_{12}) * intcond\,(s_{11}) = \\ & \left(\overline{activecond\,(s_2)} + activecond\,(s_2) * intcond\,(s_2)\right) * \\ & * \left(\overline{activecond\,(s_3)} + activecond\,(s_3) * intcond\,(s_3)\right) * \\ & * (activecond\,(endst)) = s_1 * s_{12} * endst. \end{aligned}$$

Aby tranzycja mogła zostać zrealizowana (definicja 6.12) konieczne jest, aby jej stan źródłowy był stanem aktywnym (należał do konfiguracji bieżącej) oraz, aby wszystkie ewentualne stany końcowe, podległe stanowi źródłowemu, były aktywne. Dodatkowo tranzycja musi być wzbudzana poprzez wystąpienie odpowiednich zdarzeń, co w interpretowanej postaci diagramów odpowiada spełnieniu logicznego predykatu nałożonego na tranzycję. Te trzy wymienione warunki składają się na warunek realizacji tranzycji.

Definicja 7.6. Warunek realizacji tranzycji, *oznaczany jako encond(t), obliczany jest następująco:*

$$encond\,(t) = activecond\,(out_z\,(t)) * intcond\,(out_z\,(t)) * trigger_z\,(t)\,.$$

W realizacji układowej istotne jest nie tylko to, kiedy przerzutnik jest aktywowany, ale też kiedy stan, z którym jest on związany, pozbawiany jest aktywności. Z każdego stanu sterowanie może zostać wywłaszczone przez tranzycje wyjściowe, z równych mu i wyższych poziomów hierarchii. Następująca definicja określa zbiór takich tranzycji.

Definicja 7.7. Zbiór tranzycji wywłaszczających sterowanie ze stanu s *(*ang. *state inactivating transition set), oznaczany jako* $stinacttr(s)$*, określony jest następująco:*

$$stinacttr\,(s) = \bigcup_{s_i \in path_z(root_z, s)} s_i^{\bullet}.$$

Dla przykładowego stanu s_5 zbiór ten wynosi odpowiednio: $stinacttr\,(s_5) = \{t_1, t_3, t_5\}$.

Głównym środkiem wykorzystywanym do komunikacji w diagramie są zdarzenia. Zdarzenie mają charakter globalny, tzn., że wystąpienie zdarzenia w jednej części układu jest widoczne dla wszystkich innych jego części. Jedynym zasięgiem dostępności zdarzeń jest zasięg samego diagramu, który dzieli zdarzenia na lokalne (wewnętrzne) – generowane i widoczne tylko w diagramie i nielokalne, związane ze światem zewnętrznym. Te drugie stanowią wejście i wyjście z układu. W implementacji sprzętowej zdarzenia są reprezentowane przez sygnały i związane z nimi funkcje. Do celów tworzenia funkcji zdarzeń konieczne jest wprowadzenie czterech kolejnych pojęć odnoszących się do miejsc w diagramie, gdzie zdarzenia mogą być wygenerowane.

Definicja 7.8. Zbiór stanów ustawiających sygnał *związany ze zdarzeniem* e *(*ang. *signal activating state set), oznaczany jako* $sigactst(e)$*, określony jest następująco:*

$$sigactst\,(e) = \{s \in S_z | e \in do_z\,(s)\}\,.$$

Istnieje możliwość przypisania każdemu stanowi w diagramie zdarzenia, które jest generowane wraz z nadejściem taktów zegara, tak długo jak długo rozpatrywany stan jest aktywny. Taki typ zdarzeń czasami jest nazywany akcją statyczną, a na diagramie zbiór tych zdarzeń poprzedzony jest słowem kluczowym *do*. W nawiązaniu do przykładu (rys. 7.1) zdarzeniami generowanymi w taki sposób są zdarzenie e_3 i e_4, a stanami które je generują są, odpowiednio, $sigactst\,(e_3) = \{s_1\}$, $sigactst\,(e_4) = \{s_7\}$.

Fakt aktywacji stanu jak również jego deaktywację można skojarzyć ze zdarzeniami. Na diagramie zdarzenia związane z aktywacją poprzedzone są słowem kluczowym *entry*, zaś do specyfikacji zdarzeń oznaczających wywłaszczenie sterowania ze stanu służy słowo kluczowe *exit*. Zbiory stanów związanych z tak generowanymi zdarzeniami są przedmiotem kolejnych dwu definicji.

Definicja 7.9. Zbiór stanów ustawiających sygnał związany z wejściowym zdarzeniem *e do stanu (*ang. *signal entry activating state set), oznaczanym jako* $sigenactst(e)$*, określony jest następująco:*

$$sigenactst\,(e) = \{s \in S_z | e \in entry_z\,(s)\}\,.$$

Definicja 7.10. Zbiór stanów ustawiających sygnał związany z wyjściowym zdarzeniem *e ze stanu (*ang. *signal exit activating state set), oznaczanym jako* $sigexactst(e)$*, określony jest następująco:*

$$sigexactst\,(e) = \{s \in S_z | e \in exit_z\,(s)\}\,.$$

Stanami, którym w przykładzie (rys. 7.1) przypisano generowanie zdarzeń w opisywanych okolicznościach są: $sigenactst\,(e_2) = \{s_5\}$ – dla zdarzenia wejściowego, $sigexactst\,(e_1) = \{s_2\}$ – dla zdarzenia wyjściowego.

Pozostałym miejscem w układzie, gdzie może zostać wygenerowane zdarzenie jest realizacja tranzycji. Na diagramie zbiory generowanych zdarzeń umieszczane są w opisie przy tranzycji za znakiem ukośnika (rys. 7.1 oraz punkt 3.3.1).

Definicja 7.11. Zbiór tranzycji ustawiających sygnał *związany ze zdarzeniem e (*ang. *signal activating transition set), oznaczany jako sigacttr*(e)*, określony jest następująco:*

$$sigacttr\,(e) = \{t \in T_z | e \in taction_z\,(t)\}\,.$$

Tranzycją w przykładzie (rys. 7.1), której realizacja sygnalizowana jest zdarzeniem e_2 jest t_5, czyli: $sigacttr\,(e_2) = \{t_5\}$.

Przy tak przyjętych założeniach o przyczynach powstawania zdarzeń związanych z przekazaniem sterowania do stanu, łatwo jest zauważyć następującą regułę:

$$entry_z\,(s) = A \Rightarrow \underset{t \in {}^\bullet s}{\forall} A \subseteq taction_z\,(t)\,, \tag{7.1}$$

która mówi, że jeżeli stanowi przypisuje się akcje wejściowe, to jest to równoważne przypisaniu tychże akcji wszystkim tranzycjom wejściowym do rozpatrywanego stanu. Podobną regułę można sformułować dla zdarzeń wyjściowych i tranzycji wyjściowych:

$$exit_z\,(s) = A \Rightarrow \underset{t \in s^\bullet}{\forall} A \subseteq taction_z\,(t)\,, \tag{7.2}$$

która z kolei mówi, że przypisanie stanowi akcji związanej z opuszczeniem sterowania jest równoważne przypisaniu tych samych zdarzeń wszystkim tranzycjom wyjściowym z rozpatrywanego stanu.

7.2. Synteza logiczna sterowników opisanych diagramami statechart

Zagadnienie sprzętowej syntezy układów sterowania specyfikowanych diagramami statechart jest bardzo słabo obecne w literaturze. Pierwsze próby realizacyjne wywodzą się z Instytutu Weizmanna w Rehovot w Izraelu i są wspólnym dziełem ich twórcy Dawida Harela oraz Dorona Drusinsky'ego (Drusinsky i Harel, 1989*a*; Drusinsky i Harel, 1989*b*). Główna idea tej metody polega na reprezentowaniu każdego abstrakcyjnego stanu elementem zwanym procesorem lub maszyną, implementującymi zachowanie opisane wszystkimi stanami znajdującymi się na podległym poziomie hierarchii. Maszyny te następnie są łączone sygnałami aktywującymi i deaktywującymi w sposób opierający się na topografii układu — w zależności od tego jak na diagramie przebiegają tranzycje. W wyniku takiego postępowania uzyskuje się drzewo hierarchicznie połączonych ze sobą maszyn. Poszczególne maszyny mogą być implementowane jako klasyczne automaty cyfrowe *FSM*, w których stany z diagramu odpowiadają poszczególnym stanom maszyny *FSM*, wzbogaconej

o tzw. stan jałowy (*ang.* idle state), który oznacza brak aktywności nadrzędnego stanu abstrakcyjnego. Właściwość historii jest realizowana poprzez skojarzenie z każdym stanem maszyny *FSM*, dodatkowego stanu pamiętającego jego poprzednią aktywność. W rozwiązaniu Harela-Drusinsky'ego nie uwzględnia się sprzężeń zwrotnych, stąd zdarzenia generowane w tak zrealizowanym układzie, nie mogą wpływać na jego działanie.

Inne podejście do syntezy układowej zostało przedstawione w publikacji (Drusinsky-Yoresh, 1991), gdzie kluczowymi działaniami są odpowiednie kodowanie konfiguracji stanów oraz budowa bloku logiki kombinacyjnej, będącej funkcją wzbudzeń przerzutników reprezentujących kod konfiguracji. Kodowanie konfiguracji, w pierwszym kroku, polega na ustaleniu rozłącznych zbiorów stanów, z których w danym momencie, co najwyżej tylko jeden stan może być aktywny. Każda para stanów, należących do tak wyznaczonych zbiorów rozłącznych może być współbieżnie aktywna. Każdemu stanowi ze zbioru przypisuje się unikalny w obrębie zbioru kod. Kod konfiguracji jest konkatenacją (złożeniem) kodów poszczególnych stanów. Na jego podstawie możliwe jest ustalenie aktywności stanów składających się na rozpatrywaną konfigurację. W wyniku takiego postępowania w kodowaniu biorą udział wyłącznie stany podstawowe. Ponadto, zakłada się, że wszystkie tranzycje mogą występować tylko między stanami podstawowymi. W przypadku, gdy w diagramie występują tranzycje abstrakcyjne (związane ze stanami wyższych poziomów hierarchii) wymagane jest pewne kłopotliwe przekształcenie diagramu, polegające na wprowadzeniu dodatkowego stanu podstawowego, którego aktywność oznacza aktywność stanu abstrakcyjnego związanego z rozpatrywaną tranzycją abstrakcyjną. Zadaniem cyfrowego bloku logiki kombinacyjnej jest wyznaczenie kodu konfiguracji następnej, co jest realizowane na podstawie aktualnych sygnałów wejściowych i kodu aktualnej konfiguracji bieżącej, a same funkcje logiczne bloku zbudowane są z termów, które bezpośrednio odpowiadają tranzycjom w odpowiednio przekształconym diagramie (Drusinsky-Yoresh, 1991). O sposobie tworzenia funkcji wzbudzeń bloku logiki kombinacyjnej można powiedzieć, że jest zorientowany na tranzycje. Wadą opisywanej metody, podobnie jak metody przedstawionej jako pierwszej, jest brak wyraźnego wsparcia dla właściwości historii oraz to, że nie są uwzględnione w niej oddziaływania zwrotne. Ponadto, metoda ta jest objęta zgłoszeniem patentowym (Drusinsky i Harel, 1989*a*), a jej opis ma charakter szkicowy i występuje w nim szereg niedomówień i niejasności.

W roku 1999, bazując na koncepcji Drusinky'ego (Drusinsky-Yoresh, 1991), Ramesh (Ramesh, 1999) zaproponował nową, ulepszoną metodą kodowania konfiguracji nazwaną kodowaniem prefiksowym (ang. *prefix- encoding*), w której obok stanów podstawowych również koduje się stany abstrakcyjne, przez co nie jest już wymagane kłopotliwe przekształcanie diagramu. Jak to zostało przedstawione w publikacji (Ramesh, 1999), metoda ta daje lepsze rezultaty syntezy niż propozycja Drusinsky'ego, co zostało potwierdzone analitycznie i eksperymentalnie. Mimo wykazanych niewątpliwych zalet nadal nie są uwzględniane oddziaływania zwrotne oraz brak jest w niej wsparcia dla właściwości historii.

Przedstawione koncepcje implementacji bezpośrednich nie są jedynymi kierunkami realizacji sprzętowych. Istnieje również metoda wykorzystująca języki

HDL (*VHDL* i *Verilog*), ich behawioralny podzbiór, która została zaimplementowana w systemie *Statemate MAGNUM* (I-L, 2000*b*) (podrozdział 5.3). Warto w tym miejscu również wspomnieć o ciekawej koncepcji wykorzystującej procesor z dedykowaną listą rozkazów (Buchenrieder i in., 1996) – *ASIP* (ang. *Application Specific Instruction Processor*), która została zaimplementowana w strukturach *FPGA*.

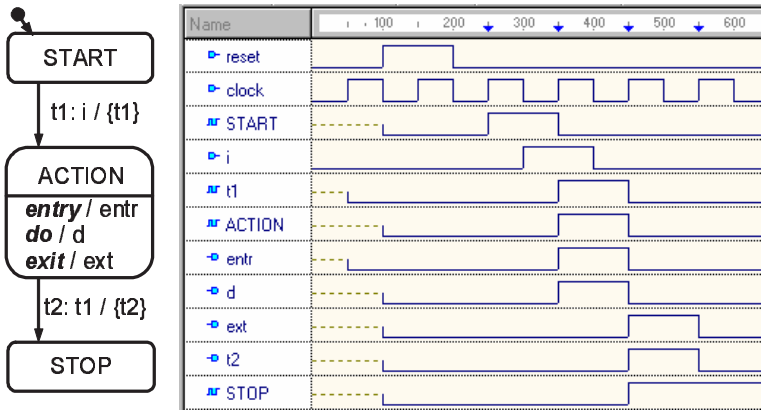
Niedostatki przedstawionych metod bezpośrednich stały się podstawą opracowania własnej metody syntezy. Prezentowany w niniejszym podrozdziale sposób opisu równaniami logicznymi opiera się na kodowaniu na zasadzie podobnej do metody *one-hot*. Każdemu stanowi z diagramu przypisuje się jeden przerzutnik, a funkcje wzbudzeń przerzutników kodu konfiguracji tworzone są według metody zorientowanej na miejsca. Dzięki takiemu podejściu uzyskano wsparcie dla właściwości historii oraz uwzględniono wpływ oddziaływań zwrotnych i wszelkie wynikającego z tego konsekwencje – nieformalnie opisane w rozdziale 4, a formalnie ujęte w podrozdziale 6.2.

7.2.1. Dynamika układu – założenia

W rozdziale 6.2 przedstawiono reguły funkcjonowania diagramów. Układ zrealizowany z wykorzystaniem specyfikacji diagramami statechart funkcjonuje w pewnym środowisku, które wytwarza zdarzenia stymulujące układ. Zakłada się, że wszelkie zdarzenia związane z układem, tzn.: wejściowe, wyjściowe i lokalne są powiązane z dyskretną dziedziną czasu. W danym momencie czasu układ reaguje na zbiór dostępnych zdarzeń poprzez realizację zbioru gotowych tranzycji zwanych mikrokrokiem (def. 6.13). Wykonanie mikrokroku na skutek generowanych zdarzeń, może pociągać za sobą przygotowanie do realizacji kolejnych tranzycji, czyli kolejnego mikrokroku. Wszystkie zdarzenia wygenerowane w czasie realizacji mikrokroku dostępne są dla układu w następnym momencie dyskretnego czasu. Dostępność wygenerowanych zdarzeń powoduje realizację kolejnego mikrokroku. Taka sekwencja mikrokroków, zwana krokiem (def. 6.16), trwa aż do momentu, gdy kolejne wykonania mikrokroków nie pociągają już za sobą pojawiania się w układzie gotowych do realizacji tranzycji, przy czym przyjmuje się że w trakcie kroku do układu nie dochodzą żadne zdarzenia zewnętrzne. Podsumowując, o dynamice realizacji układowej można powiedzieć:

- układ jest synchroniczny,
- układ reaguje na zbiór dostępnych zdarzeń poprzez realizacje poszczególnych tranzycji,
- wygenerowane zdarzenia są dostępne dla układu w następnym momencie dyskretnego czasu.

Za przykład niech posłuży diagram wraz z odpowiadającym mu przebiegiem czasowym z rysunku 7.2, otrzymanym z programu *HiCoS* i symulatora (środowisko *Active HDL* firmy *ALDEC*). Układ jest taktowany zboczem narastającym sygnału zegarowego o nazwie *clock*. Po wyzerowaniu ($t = 150$) w chwili $t = 250$,



Rys. 7.2. Przykład prostego diagramu wraz z przebiegami czasowymi

układ jest w stanie początkowym *START*. Zdarzeniem, które dociera do układu z otoczenia układu jest zdarzenie o nazwie i. W odpowiedzi na to zdarzenie, w chwili $t = 350$, układ wykonuje mikrokrok składający się z jednej tranzycji t_1 i przechodzi do stanu *ACTION*. Realizacja t_1 powoduje wygenerowanie zdarzenia t_1, które jest dostępne dla układu dopiero przy następnym zboczu narastającym sygnału taktującego ($t = 450$). Wystąpienie zdarzenia t_1 powoduje, że tranzycja t_2, związana z bieżącą konfiguracją, jest pobudzana i tym samym gotowa do realizacji. W chwili $t = 450$ tranzycja t_2, jednocześnie będąca kolejnym mikrokrokiem, jest realizowana, a jej efekt – zdarzenie t_2 – już nie wpływa na realizację kolejnych mikrokroków, co zgodnie z definicją kroku (def. 6.16) oznacza jego zakończenie. Układ znajduje się w stanie *STOP*. Dodatkowo, przedstawiony przykład ilustruje generowanie zdarzeń związanych z aktywacją stanu. Zdarzenie o nazwie *entr*, które jest typu *entry*, pojawia się w tym samym momencie czasu, gdy uaktywniany jest stan, któremu jest przypisane (w przykładzie $t = 350$ i stan *ACTION*). Wartość **1** sygnału związanego z takim zdarzeniem podtrzymywana jest do następnego zbocza narastającego sygnału taktującego ($t = 450$), kiedy to jest dostępne dla układu. Zdarzenie o nazwie *ext*, które jest typu *exit*, pojawia się w sytuacji gdy stan z którym jest związane przestaje być stanem aktywnym. W chwili $t = 450$ stan *ACTION* traci aktywność i jest generowana wartość **1** na sygnale reprezentującym rozpatrywane zdarzenie. Wartość **1** podtrzymywana jest do następnego sygnału taktującego. Zdarzenie d, typu *do*, jest tzw. akcją statyczną, a wartość **1** na sygnale związanym z tym zdarzeniem jest podtrzymywana tak długo, jak długo stan któremu jest przypisane jest aktywny, w przykładzie od chwili $t_1 = 350$ do chwili $t_2 = 450$, a aktywnym stanem jest *ACTION*.

7.2.2. Założenia realizacji sprzętowej

Formułując założenia realizacji sprzętowej należy przytoczyć fragment tezy pracy dotyczącej implementacji. Teza mówi o tym, że cyfrowe systemy sterujące, któ-

rych zachowanie modelowane jest diagramami statechart, mogą być bezpośrednio implementowane w strukturach programowalnych. Założenie o bezpośredniej implementacji oznacza, iż elementy diagramu są w sposób bezpośredni odwzorowane w struktury programowalne (głównie programowana logika kombinacyjna i przerzutniki). Na tej podstawie, oraz biorąc pod uwagę założenia dotyczące dynamiki implementowanego układu, zostały sformułowane następujące zasady sprzętowej implementacji:

- każdemu stanowi z diagramu jest przyporządkowany jeden przerzutnik – aktywność przerzutnika oznacza, że stan, z którym skojarzony jest przerzutnik, może być aktywny lub, w przypadku stanu z atrybutem historii, jest pamiętana jego przeszła aktywność,
- każdemu miejscu na diagramie, gdzie może być generowane zdarzenie, również jest przyporządkowany przerzutnik, którego zadaniem jest podtrzymanie informacji o zdarzeniu, do momentu nadejścia następnego taktu sygnału zegarowego, kiedy to zdarzenie będzie dostępne dla systemu,
- na podstawie topografii diagramu oraz reguł realizacji tranzycji, dla każdego przerzutnika w systemie tworzone sa funkcje wzbudzeń.

Jak widać z tak przyjętych założeń, aktywność przerzutnika stanu może oznaczać wystąpienie jednej z dwu sytuacji. Zatem, aby ustalić aktywność stanu skojarzonego z danym przerzutnikiem, należy wziąć pod uwagę wyjścia przerzutników, związanych ze wszystkimi stanami nadrzędnymi.

Dalszy opis zawarty w podrozdziale 7.2 głównie koncentruje się wokół tworzenia funkcji wzbudzeń przerzutników stanu i przerzutników zdarzeń oraz na realizacji funkcji sygnałów.

7.2.3. Funkcje wzbudzeń przerzutników związanych ze stanami

Podstawowym założeniem realizacji układowej jest skojarzenie z każdym stanem diagramu jednego przerzutnika. Przyjmuje się, że wartość **1** na jego wyjściu może oznaczać wystąpienie jednej z dwu sytuacji:

- aktywność stanu (z którym związano przerzutnik),
- pamiętanie że dany stan jest stanem ostatnio aktywnym — co ma miejsce w przypadku stanów z atrybutem historii.

Te dwie okoliczności są zasadniczo różne, zatem należy określić reguły pozwalające jednoznacznie ustalić aktywność stanu i drugą z opisywanych sytuacji. Definicja 6.18 określa stan aktywny w sytuacji, gdy wszystkie jego stany nadrzędne należą do konfiguracji bieżącej, czyli są aktywne. Zatem do ustalenia aktywności stanu służy spełnienie warunku aktywności stanu, określonego w definicji 7.4.

Mając określoną rolę jaką pełni przerzutnik stanu, można ustalić założenia co do budowy jego funkcji wzbudzeń. Funkcja ta przyjmuje wartość **1**, gdy stan związany z przerzutnikiem:

- nie jest aktywny i w następnej iteracji będzie stanem aktywnym,

- jest stanem aktywnym lub tzw. stanem ostatnio aktywnym (co ma miejsce w przypadku stanów z atrybutem historii) i w następnej iteracji również będzie stanem aktywnym lub stanem ostatnio aktywnym.

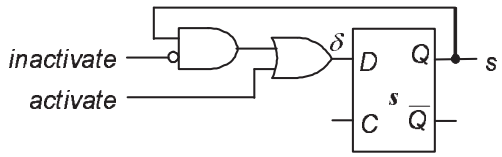
Cechą wspólną powyższych dwu założeń jest pewien związek, polegający na tym, że zanim stan będzie stanem ostatnio aktywnym, to musi być najpierw stanem aktywnym. Spostrzeżenie to prowadzi do równania funkcji wzbudzeń przerzutnika stanu o następującej postaci:

$$\delta(s) = \underbrace{activate(s)}_{a} + \underbrace{s * \overline{incativate(s)}}_{b}, \tag{7.3}$$

gdzie:

a) składowa zapalająca – *activate*: przyjmuje wartość **1**, gdy stan związany z przerzutnikiem nie jest stanem aktywnym i w następnym kroku stanie się stanem aktywnym; odpowiada to realizacji tranzycji wejściowych do stanu (czyli tranzycji należących do zbioru ${}^{\bullet}s$) oraz w przypadku stanów początkowych aktywacji ewentualnego stanu nadrzędnego,

b) składowa podtrzymująca: przyjmuje wartość **1**, gdy stan związany z przerzutnikiem jest stanem aktywnym (nastąpiło spełnienie składowej *activate*) i w następnej iteracji również będzie stanem aktywnym, lub gdy posiada atrybut historii i jest tzw. stanem ostatnio aktywnym; czynnik *inactivate* przyjmuje wartość **1**, gdy stan przestaje być stanem aktywnym i nie jest stanem ostatnio aktywnym, czyli zostaje zrealizowana jedna z tranzycji wywłaszczających sterowanie (np. należąca do zbioru $s^{\bullet}$ lub $stinacttr(s)$).

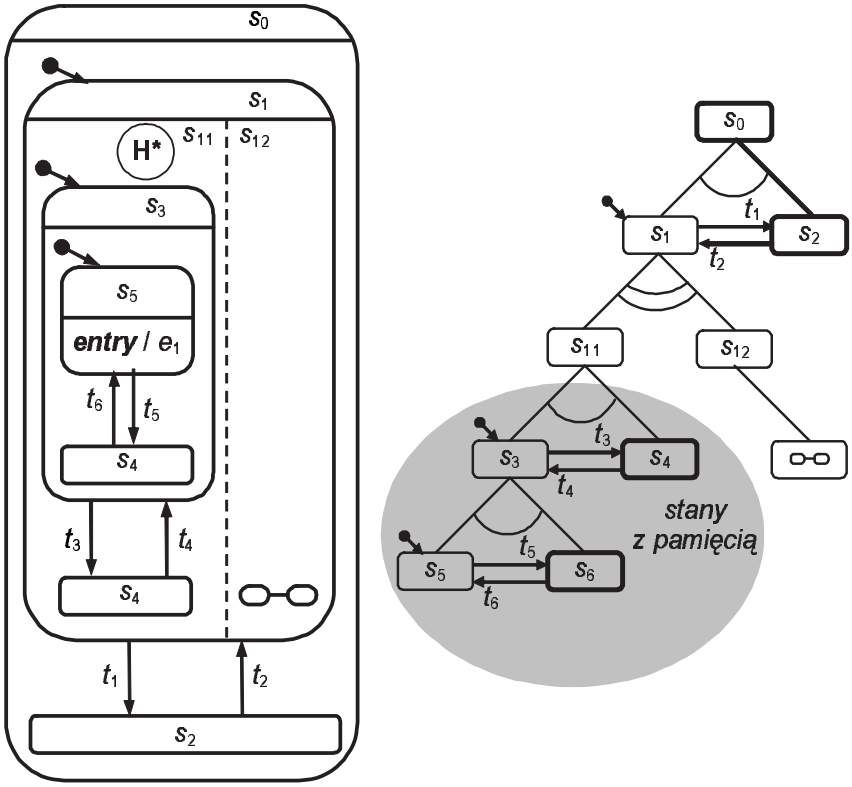


Rys. 7.3. Schemat logiczny funkcji wzbudzeń przerzutnika stanu

Zmienna s w równaniu, będąca sygnałem sprzężenia zwrotnego z wyjścia przerzutnika, podtrzymuje jego aktywność od momentu określonego poprzez składową *activate* do momentu określonego poprzez czynnik *inactivate*. Tak zdefiniowana funkcja wzbudzeń prowadzi wprost do realizacji logicznej przedstawionej na rysunku 7.3. Dalszy opis reguł tworzenia równań logicznych koncentruje się wokół budowy składowej zapalającej *activate* i czynnika *inactivate*.

7.2.3.1. Składowa zapalająca activate

Zadaniem składowej *activate* jest wzbudzanie przerzutnika w sytuacji, gdy stan z którym jest on skojarzony ani nie jest stanem aktywnym i ani nie jest stanem ostatnio aktywnym. Taka aktywacja przerzutnika związana jest z realizacją każdej bezpośredniej tranzycji wejściowej do stanu, zatem budowa składowej *activate* wynika wprost z topografii diagramu oraz z reguł jego działania. Tworząc wyrażenie na składową aktywującą należy rozpatrzyć w pierwszej kolejności te tranzycje, których realizacja zawsze powoduje uaktywnienie przerzutnika stanu. Takimi tranzycjami są bezpośrednie tranzycje wejściowe (czyli należące do zbioru ${}^\bullet s$). W przypadku stanów nie będących stanami początkowymi jest to wystarczające założenie.



Rys. 7.4. Diagram wraz z odpowiadającym mu drzewem hierarchii

W przypadku stanów początkowych aktywacja przerzutników jest uzależniona nie tylko od tranzycji należących do zbioru ${}^\bullet s$, ale też i od tranzycji wyższych poziomów hierarchii i od sytuacji związanych z obecnością atrybutu historii (zarówno w przypadku samego stanu początkowego, jak i automatów nadrzędnych). Na przykład na diagramie z rysunku 7.4, gdzie wszystkie stany poniżej s_{11} po-

siadają atrybut historii (nadany poprzez umieszczenie na diagramie symbolu tzw. historii głębokiej: **H***), stan początkowy s_5 zawsze będzie aktywowany poprzez realizację tranzycji wejściowej t_6, z kolei tranzycja t_4 będzie uaktywniać s_5 w sytuacji, gdy s_5 nie był jeszcze aktywowany lub jest stanem ostatnio aktywnym. Jeśli by rozpatrywać aktywacje stanu s_5, pochodzącą od innej pozostałej możliwej tranzycji t_2, to dodatkowo należałoby wziąć pod uwagę sytuację zaistniałą w automacie składającym się ze stanów s_3 i s_4. Wówczas to na przykład, tak jak to pokazuje rysunek 7.4, aktywnym byłby s_4, przez co s_5 nie mógłby być uaktywniony. Realizacja t_2 spowodowałaby wzbudzenie przerzutnika stanu s_4 a nie s_3, będącego bezpośrednim stanem nadrzędnym dla s_5. Dalsze rozważania prowadzone podobnym tokiem prowadzą do spostrzeżenia, że aktywacja przerzutnika związanego ze stanem początkowym jest wynikiem aktywacji bezpośredniego stanu nadrzędnego, pod warunkiem, że żaden z przerzutników związanych ze stanami należącymi do automatu, z którym jest on związany, nie jest wzbudzony (co oznacza że żaden ze stanów nie jest stanem ostatnio aktywnym). Całość sprowadza się do rozpatrzenia dwu przypadków i formalnie przedstawia się następująco:

- stan nie będący stanem początkowym:

$$activate\,(s) = \sum_{t_i \in {}^\bullet s} encond\,(t_i) \tag{7.4}$$

  przerzutnik aktywowany jest wyłącznie przez bezpośrednie tranzycje wejściowe,

- stan będący stanem początkowym

$$activate\,(s_d) = \underbrace{\sum_{t_i \in {}^\bullet s_d} encond\,(t_i)}_{a} + \underbrace{\prod_{s_i \in path(root_z, parent_z(s_d))} \delta\,(s_i) *}_{b} * \underbrace{\sum_{s_i \in hrc_z(parent_z(s_d))} s_i}_{c} \tag{7.5}$$

  przerzutnik aktywowany jest przez swoje bezpośrednie tranzycje wejściowe (a) oraz przez aktywację bezpośredniego stanu nadrzędnego w następnej iteracji (b), jednakże pod warunkiem, że żaden ze stanów automatu, z którym jest on związany nie jest stanem ostatnio aktywnym, czyli wszystkie przerzutniki automatu są wyzerowane (c).

W niniejszym rozdziale na rysunkach i w równaniach, tak jak to zostało wprowadzone przy okazji definicji 7.4, symbolem s_i oznacza się wyjście z przerzutnika związanego ze stanem s_i. Symbolem t_i oznacza się wyrażenie na warunek realizacji tranzycji t_i (7.4). Symbol δ_i natomiast oznacza wartość funkcji wzbudzeń stanu s_i i w realizacji układowej jest sygnałem stanowiącym wejście przerzutnika. Różnica

między sygnałami s_i a δ_i polega na tym, że wartość s_i wskazuje sytuację aktualną, natomiast wartość δ_i wskazuje sytuację, która wystąpi przy następnym takcie zegara. Czynnik b w równaniu 7.5 na aktywację przerzutnika stanu początkowego, oznacza sytuację w układzie, która pojawi się w następnym momencie dyskretnego czasu. Jeżeli przerzutniki wszystkich stanów nadrzędnych w następnej iteracji będą wzbudzone (wówczas wartość czynnika wynosi **1**), to zgodnie z definicją 7.4 oznacza to, że stan nadrzędny będzie stanem aktywnym. Jeżeli dodatkowo, aktualnie żaden ze stanów składowych rozpatrywanego automatu nie jest stanem ostatnio aktywnym, czyli przerzutniki z nimi związane na swoich wyjściach mają wartość **0** (czynnik c), to znaczy że w następnej iteracji dany stan początkowy ma być aktywny, a jego przerzutnik ma zostać ustawiony w stan **1**.

Ryunek 7.5 stanowi ilustrację schematyczną omawianego przypadku dla diagramu zamieszczonego na rysunku 7.4. Zamieszczono tam fragment układu stanowiącego realizację sprzętową stanów należących do $path(s_0, s_5)$ wraz ze zdarzeniem e_1 związanym ze stanem s_5, przy czym zostały pominięte stany s_0 i s_{11}, które w realizacji sprzętowej zazwyczaj są nadmiarowe. W ogólności stan $root_z$ (np. s_0) oraz stany dla których (np. s_{11}) czyli stany, które na diagramach również nie są zamieszczane, w realizacji sprzętowej są pomijane, gdyż wnoszą redundantną informację. Na rysunku widać, że sygnał typu δ_i znajduje się na wejściu przerzutnika, a sygnał typu s_i na jego wyjściu.

Po spełnieniu warunków określonych w równaniach 7.4 lub 7.5 składowa przyjmuje wartość **1**. Dodatkowo, w przypadku stanów początkowych należących do automatów bezpośrednio podległych stanowi $root_z$, do składowej *activate* dodawany jest następujący składnik ustawiający:

$$setup = \prod_{s_i \in hrc_z(root_z)} \bar{s}_i, \tag{7.6}$$

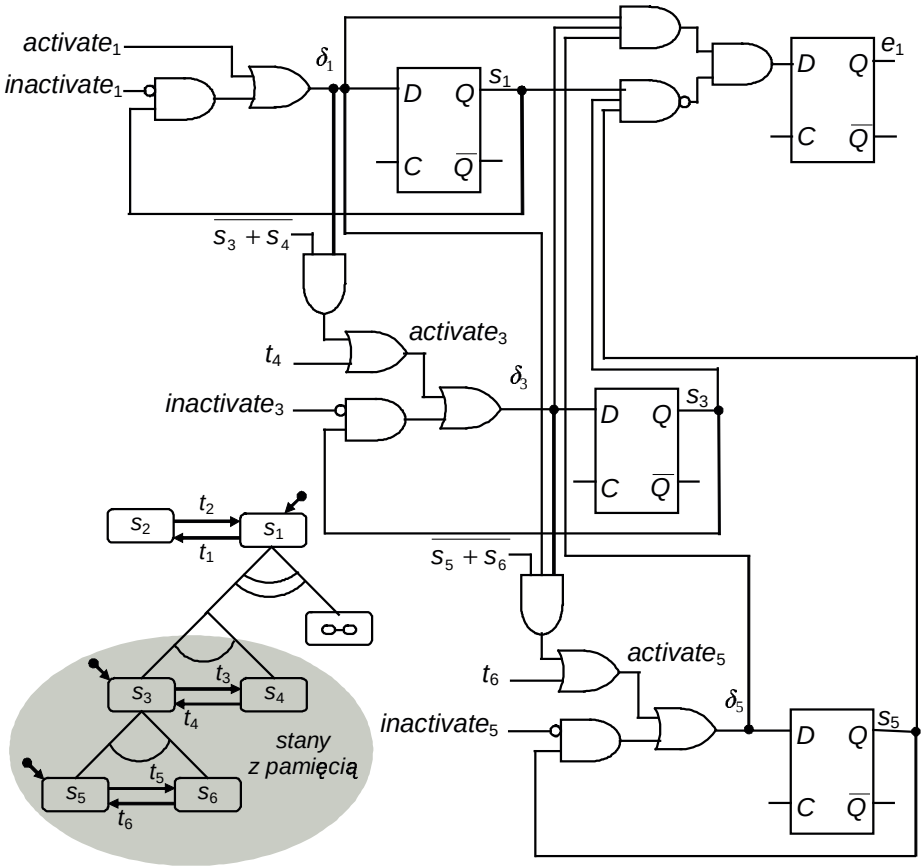
którego rolą jest aktywacja przerzutników po globalnym wyzerowaniu systemu, przy założeniu, że sygnał *reset* ustawia przerzutniki w stan **0**. W wyrażeniu na składnik *setup* wykorzystano fakt, że w trakcie funkcjonowania układu zawsze co najmniej jeden z przerzutników związanych ze stanami automatu na najwyższym poziomie hierarchii jest aktywny, tym samym stan zerowy wszystkich przerzutników oznacza, że układ nie funkcjonuje i taką sytuację przyjęto uważać za moment zerowania układu.

Jako przykład do prowadzonych rozważań niech posłużą równania dla przerzutników s_1, s_2 i s_3, dla realizacji diagramu przedstawionego na rysunku 7.4, pozbawionej stanów nadmiarowych (s_0, s_{11}, s_{12}). Stan nie będący stanem początkowym – przypadek opisany równaniem 7.4:

$$activate\,(s_2) = t_1, \qquad activate\,(s_4) = t_3, \qquad activate\,(s_6) = t_5.$$

Stan początkowy należący do konfiguracji początkowej – przypadek opisany równaniami 7.5 i 7.6:

$$activate\,(s_1) = t_2 + setup = t_2 + \bar{s}_1 * \bar{s}_2.$$



Rys. 7.5. Fragment realizacji układowej oraz zredukowane drzewo hierarchii dla diagramu z rysunku 7.4

Stany początkowe nie należący do konfiguracji początkowej – przypadek opisany równaniem 7.4:

$$activate\,(s_3) = t_4 + \delta_1 * \overline{(s_3 + s_4)}, \qquad activate\,(s_5) = t_6 + \delta_1 * \delta_3 * \overline{(s_5 + s_6)}.$$

Część równania 7.5 (a także równań 7.7, 7.11, 7.14 i 7.15) dotycząca tranzycji niebezpośrednich (w równaniu 7.5 składnik oznaczony jako b) mogłaby być przedstawiona w postaci bardziej naturalnej, zależnej od aktualnego stanu diagramu (reprezentowanej w równaniach przez symbol s_i), lecz próby stworzenia takiego opisu okazywały się zawiłe i autor z tego kierunku poszukiwań zrezygnował. Uzależnienie składnika aktywującego nie od aktualnej sytuacji w diagramie, lecz od informacji o sytuacji, która dopiero pojawi się w chwili następnej (reprezentowanej przez symbol δ_i) jest prawidłowa, ale tylko wtedy, gdy składnik aktywacyjny nie wpływa na funkcję wzbudzeń stanu od którego jest on uzależniony. Jak można to zauważyć w równaniu 7.5 (a także w równaniach 7.7, 7.11, 7.14 i 7.15) taka

sytuacja rzeczywiście ma miejsce. Czynnik *activate*, co prawda zależy od funkcji wzbudzeń δ_i (do obliczenia której stosuje się czynnik *activate*), lecz jest to funkcja wzbudzeń przerzutników stanów wyższych poziomów hierarchii, na które to rozpatrywany czynnik *activate* nie ma żadnego wpływu. Wynika to z prostego faktu, że w autorskim modularnym modelu diagramów statechart, realizacja tranzycji niebezpośrednich powoduje, że sterowanie może nadejść tylko z „góry" (w sensie drzewa hierarchii), czyli przerzutniki stanów wyższych poziomów hierarchii wzbudzają przerzutniki stanów niższych poziomów, ale nigdy odwrotnie.

Zdaniem autora, tworząc równania dla modelu diagramów z tranzycjami przekraczającymi granice stanów, wskazanym jest posługiwanie się predykcyjną formą równań (operującą informacją o sytuacji, która pojawi się w najbliższej przyszłej chwili następnej) z jednoczesnym uwzględnieniem „kierunku" nadejścia sterowania.

7.2.3.2. Czynnik inactivate

Zadaniem czynnika *inactivate* jest wyzerowanie przerzutnika (poprzez przyjęcie wartości **1** w funkcji wzbudzenia – równanie 7.3) w sytuacji, gdy stan z którym jest on skojarzony przestaje być stanem aktywnym i jednocześnie nie jest stanem ostatnio aktywnym. Sytuacja taka jest wynikiem realizacji odpowiednich tranzycji. Zbiory tranzycji których realizacje powodują zerowanie przerzutnika są zależne od obecności stanu końcowego w automacie, do którego należy stan związany z rozpatrywanym przerzutnikiem, oraz od obecności atrybutu historii. Podobnie jak to miało miejsce przy opisie czynnika *activate*, w poniższych równaniach tranzycje (zasadniczo wyższych poziomów hierarchii) reprezentowane są przez warunek aktywności stanu nadrzędnego.

W przypadku stanów nie będących stanami końcowymi prosta kombinacja tych dwu cech prowadzi do rozpatrzenia czterech sytuacji:

- stan bez atrybutu historii należący do automatu bez stanu końcowego:

$$inactivate\,(s) = \underbrace{\sum_{t_i \in s^\bullet} encond\,(t_i)}_{a} + \underbrace{\overline{\prod_{s_i \in path(root_z, parent_z(s))} \delta\,(s_i)}}_{b}. \quad (7.7)$$

  Przerzutnik jest zerowany poprzez realizacje bezpośrednich tranzycji wyjściowych ze stanu s (składnik a) oraz poprzez realizacje tranzycji wyjściowych stanów nadrzędnych, które w równaniu reprezentowane są przez warunek na aktywność stanu nadrzędnego w następnej iteracji (składnik b).

- Stan bez atrybutu historii należący do automatu ze stanem końcowym:

$$inactivate\,(s) = \sum_{t_i \in s^\bullet} encond\,(t_i). \quad (7.8)$$

  Sterowanie z automatu ze stanem końcowym może zostać wywłaszczone tylko po osiągnięciu przez ten automat stanu końcowego, zatem tranzycje wyższych poziomów hierarchii nie mogą wyzerować takiego przerzutnika.

- Stan z atrybutem historii należący do automatu bez stanu końcowego:

$$inactivate\,(s) = \sum_{t_i \in s^\bullet} encond\,(t_i). \tag{7.9}$$

  Realizacja tranzycji wyższych poziomów hierarchii oznacza sytuację, że dany stan jest stanem ostatnio aktywnym i nie zeruje przerzutnika, gdyż ten pamięta aktywność stanu.

- Stan z atrybutem historii należący do automatu ze stanem końcowym:

$$inactivate\,(s) = \sum_{t_i \in s^\bullet} encond\,(t_i). \tag{7.10}$$

  Obecność stanu końcowego w automacie z atrybutem historii sprowadza się do tego, że tak zadeklarowany automat pozbawiony jest właściwości pamięci (atrybutu historii), co odpowiada sytuacji opisanej równaniem 7.8.

Natomiast w przypadku stanu końcowego istotne jest uwzględnienie tylko atrybutu historii:

- Stan będący stanem końcowym bez historii:

$$inactivate\,(endst) = \overline{\underbrace{\prod_{s_i \in path(root_z, parent_z(s))} \delta\,(s_i)}_{b}}. \tag{7.11}$$

  Stan końcowy nie może posiadać bezpośrednich tranzycji wyjściowych (definicja 6.4 punkt 2), lecz jego aktywność może być usunięta poprzez realizacje tranzycji wyższych poziomów hierarchii (podobnie jak w sytuacji opisanej równaniem 7.7).

- Stan będący stanem końcowym z historią:

$$inactivate\,(endst) = 0. \tag{7.12}$$

  Nadanie stanowi końcowemu atrybutu historii oznacza, że po uaktywnieniu takiego stanu, zawsze będzie pamiętana jego aktywność i ponowne wznowienia automatu z nim związanego będą prowadziły do jego uaktywnienia, a tym samym taki przerzutnik już nigdy, w skutek normalnego działania układu, nie będzie wyzerowany, stąd w równaniu wartość **0**; w ogólności stosowanie stanu końcowego razem z atrybutem historii nie jest zalecane.

W nawiązaniu do diagramu z rysunek 7.1, stan s_3 jest stanem bez atrybutu historii należącym do automatu bez stanu końcowego (przypadek opisany równaniem 7.7, zatem:

$$inactivate\,(s_3) = \sum_{t_i \in s_3^\bullet} encond\,(t_i) + \overline{\delta\,(s_1)} = t_6 + 0 = t_6,$$

stan s_5 jest stanem bez atrybutu historii, należącym do automatu ze stanem końcowym (przypadek opisany równaniem 7.9):

$$inactivate\,(s_5) = \sum_{t_i \in s_5^\bullet} encond\,(t_i) = t_3,$$

stan s_7 jest stanem z historią, należącym do automatu bez stanu końcowego (przypadek 7.9):

$$iinactivate\,(s_7) = \sum_{t_i \in s_7^\bullet} encond\,(t_i) = t_5.$$

7.2.4. Funkcje wzbudzeń przerzutników zdarzeń tranzycji

Z realizacją każdej tranzycji można skojarzyć generowanie dowolnego zbioru zdarzeń. Z każdą taką tranzycją należy związać jeden przerzutnik. Rolą przerzutnika jest podtrzymanie informacji o wytwarzanym zdarzeniu do momentu nadejścia następnego taktu zegara. Dla każdego takiego przerzutnika należy określić funkcję wzbudzeń, która przyjmuje wartość **1** gdy tranzycja ma być zrealizowana. Funkcją wzbudzenia takiego przerzutnika jest warunek realizacji tranzycji (def. 7.6):

$$\delta\,(t) = encond\,(t) \tag{7.13}$$

Przykładem tranzycji, z realizacją której związano zdarzenie, jest tranzycja t_5 z rysunku 7.1. Generowane zdarzenie to zdarzenie e_2.

7.2.5. Funkcje wzbudzeń przerzutników zdarzeń wejściowych stanu

Każdemu stanowi w układzie (za wyjątkiem stanu końcowego) można przyporządkować akcję (czyli zdarzenia) związaną z jego aktywacją. Za każdym razem gdy stan jest uaktywniany, odpowiednie zdarzenia mogą być generowane. Na diagramach zbiory takich zdarzeń poprzedzone są słowem kluczowym *entry*. Zdarzenia związane z aktywacją stanu są dostępne dla układu w następnym (od momentu aktywacji stanu) takcie sygnału zegarowego – stąd w celu podtrzymania informacji o generowanych zdarzeniach wynika potrzeba związania z taką akcją wejściową przerzutnika. Funkcja wzbudzenia takiego przerzutnika (równanie 7.14) składa się z warunku na aktywność stanu w następnej iteracji (czynnik b), przy założeniu, że aktualnie stan nie jest aktywny (czynnik a). Z tak skonstruowanej funkcji wzbudzenia płynie wniosek, że realizacja tranzycji o początku i końcu w tym samym stanie, nie spowoduje wygenerowania zdarzenia wejściowego, gdyż przed realizacją takiej tranzycji rozpatrywany stan był stanem aktywnym (czyli *de facto* uaktywnienie nie ma miejsca). W dalszej części opisu takie przerzutniki są nazywane *en.*

$$\delta\,(e_s) = \underbrace{\overline{activecond\,(s)}}_{a} * \underbrace{\prod_{s_i \in path(root_z, s)} \delta\,(s_i)}_{b} \tag{7.14}$$

Ilustracją działania przerzutników związanych ze zdarzeniami są przebiegi czasowe zamieszczone na rysunku 7.2, natomiast realizacja układowa takiego przerzutnika (dla zdarzenia e_1 przyporządkowanego stanowi s_5 z diagramu na rysunku 7.4) przedstawiona jest na rysunku 7.5.

7.2.6. Funkcje wzbudzeń przerzutników zdarzeń wyjściowych stanu

Podobnie jak to ma miejsce w przypadku zdarzeń wejściowych, również każdemu stanowi można przyporządkować akcję związaną z sytuacją, gdy stan traci aktywność. Na diagramie zdarzenia takie poprzedzone są słowem kluczowym *exit*. Również i w tym przypadku, dla podtrzymania informacji o wystąpieniu takiej akcji, konieczna jest implementacja pomocniczego przerzutnika. Funkcja wzbudzenia takiego przerzutnika (równanie 7.15) składa się z warunku na aktywność stanu (czynnik a) i z warunku mówiącego, że w następnej iteracji stan nie będzie stanem aktywnym (czynnik b). Realizacja tranzycji o początku i końcu w tym samym stanie nie powoduje generowania opisywanej akcji. Przerzutniki pomocnicze związane z akcją wyjściową w dalszej części nazywane są *ex*.

$$\delta\left(e_s\right) = \underbrace{activecond(s)}_{a} * \underbrace{\overline{\prod_{s_i \in path(root_z, s)} \delta\left(s_i\right)}}_{b} \tag{7.15}$$

Przebieg czasowy zdarzeń wyjściowych przedstawia rysunek 7.2. Inny przykład zdarzenia wyjściowego znajduje się na rysunku 7.1 (zdarzenie e_1 przypisane do stanu s_2).

7.2.7. Funkcje sygnałów

W modelowaniu projektowanego sterownika diagramami statatechart, głównym środkiem do synchronizacji w układzie oraz komunikacji ze światem zewnętrznym, są rozgłaszane zdarzenia o zasięgu globalnym (ang. *broadcasting*). W definicji statecharta interpretowanego (def. 7.1) przyjęto, że w realizacji sprzętowej zdarzenia będą reprezentowane poprzez sygnały. Wartość logiczna **1** sygnału oznacza wystąpienie zdarzenia z nim związanego, wartość **0** – jego brak. Dodatkowo założono, że to samo zdarzenie może wystąpić w kilku różnych miejscach w układzie (np. jako wejściowe lub jako akcja statyczna) lub też może nadejść z otoczenia układu. Ponadto przyjęto, że na zdarzeniach można dokonywać dowolnych operacji logicznych, a wyniki tych działań mogą służyć definiowaniu innych zdarzeń w układzie. Z tak przyjętych założeń płynie konieczność określenie funkcji definiujących sygnały zdarzeń.

Sygnały można podzielić na wejściowe – przychodzące z zewnątrz, wyjściowe – widoczne dla otoczenia i lokalne (wewnętrzne) – istniejące tylko w układzie. Mając na uwadze punkt 5 z definicji 7.1, mówiący o tym, że zbiory zdarzeń przychodzących i wychodzących z układu są rozłączne, w realizacji układowej można dokonać podziału na sygnały związane z wejściem i sygnały nie związane z wejściem, zaliczając do drugiej grupy wyróżniony zbiór sygnałów wychodzących.

7.2.7.1. Funkcje sygnałów związanych z wejściem układu

Funkcja sygnału zdarzenia związanego z wejściem układu jest następująca:

$$\underset{e\in X}{\forall}\lambda(e) = \underbrace{input(e)}_{a} + \underbrace{\sum_{s_i\in sigactst(e)} activecond(s_i)}_{b} + \underbrace{\sum_{t_j\in sigacttr(e)} t_j}_{c} + \underbrace{\sum_{s_k\in sigenactst(e)} en_k}_{d} + \underbrace{\sum_{s_l\in sigexactst(e)} ex_l}_{e} + \underbrace{expr(g)}_{f} \quad (7.16)$$

Powyższe równanie wynika wprost z przyjętych założeń. Składnik a stanowi sygnał związany ze zdarzeniem przychodzącym z zewnątrz (def. 7.1 punkt 3). Składnik b reprezentuje akcje statyczne (def. 7.8, warunki aktywności tych stanów którym na diagramie przypisano zdarzenia poprzedzone słowem kluczowym *do*), składnik c zdarzenia związane z realizacją tranzycji (def. 7.11). Składniki d i e odpowiadają za zdarzenia, kolejno, wejściowe do i wyjściowe ze stanu (odpowiednio def. 7.9 i 7.10). Składnik f oznacza dowolne wyrażenia boolowskie na zdarzeniach (tak jak to określa gramatyka podana w punkcie 6 z definicji 7.1).

7.2.7.2. Funkcje sygnałów nie związanych z wejściem układu

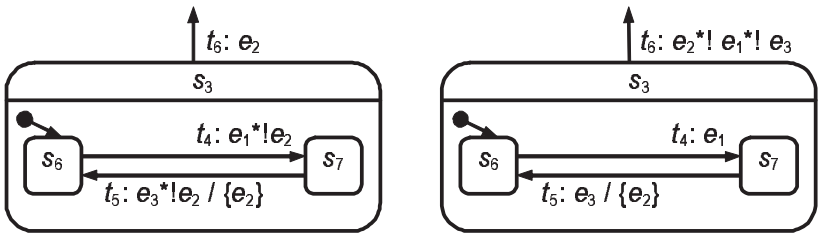
Funkcja sygnału zdarzenia nie związanego z wejściem układu jest następująca:

$$\underset{e\in X}{\forall}\lambda(e) = \underbrace{\sum_{s_i\in sigactst(e)} activecond(s_i)}_{b} + \underbrace{\sum_{t_j\in sigacttr(e)} t_j}_{c} + \underbrace{\sum_{s_k\in sigenactst(e)} en_k}_{d} + \underbrace{\sum_{s_l\in sigexactst(e)} ex_l}_{e} + \underbrace{expr(g)}_{f} \quad (7.17)$$

Znaczenie kolejnych składników są takie same jak w równaniu 7.16. Wyróżniony zbiór sygnałów nie związanych z wejściem i widoczny dla świata zewnętrznego stanowi wyjście z układu (punkt 4 z definicji 7.1).

7.3. Tranzycje w konflikcie, rozwiązywanie konfliktów

Składnia diagramów opisana w rozdziale 6 dopuszcza modelowanie zachowania układów w sposób niedeterministyczny. Układ zamodelowany diagramem niedeterministycznym cechuje się występowaniem tranzycji będących w konflikcie. O dwóch tranzycjach gotowych do realizacji można powiedzieć że są w konflikcie, jeżeli istnieje jeden taki stan, że realizacja dowolnej z nich, powoduje wywłaszczenie sterowania ze stanu (definicja zaczerpnięta z Harel i Naamad, 1996). W przykładzie (rys. 7.6) tranzycjami które mogłyby być w konflikcie są tranzycje t_6 i t_4 oraz t_6 i t_5. W pierwszym przypadku stanem pozbawionym sterowania byłby stan s_6, w drugim przypadku stan s_7.



Rys. 7.6. Dwa rodzaje priorytetów tranzycji

W realizacji diagramu w postaci układu cyfrowego ma tylko sens implementacja zachowania deterministycznego, zatem w modelowanym układzie należy wykryć konflikty, a następnie je usunąć. Do rozwiązywania konfliktów wykorzystuje się predykaty, które pozwalają na ustalenie priorytetów między tranzycjami. Predykaty powinny mieć charakter ortogonalny.

Rysunek 7.6 przedstawia dwa możliwe sposoby rozwiązywania konfliktów między tranzycjami. W przykładzie z lewej strony rysunku, tranzycja wyższego poziomu hierarchii posiada priorytet nad tranzycjami niższych poziomów hierarchii. Jednoczesne wystąpienie zdarzeń e_1, e_2 i e_3 przy aktywności stanu s_3 spowoduje, że zostanie zrealizowana tranzycja wzbudzana zdarzeniem e_2, czyli tranzycja t_6. Ustalenie priorytetów zostało osiągnięte dzięki odpowiedniemu dobraniu predykatów. W drugim przykładzie sytuacja jest odwrotna, tranzycje niższych poziomów hierarchii t_4 i t_5 posiadają priorytet nad tranzycją t_6. Jak widać, rozwiązywanie konfliktów między tranzycjami przy wykorzystaniu predykatów pozwala na ustalanie priorytetów między tranzycjami w sposób dowolny tak, że w prezentowanym przykładzie można sobie wyobrazić każde inne priorytetowe uszeregowanie tranzycji.

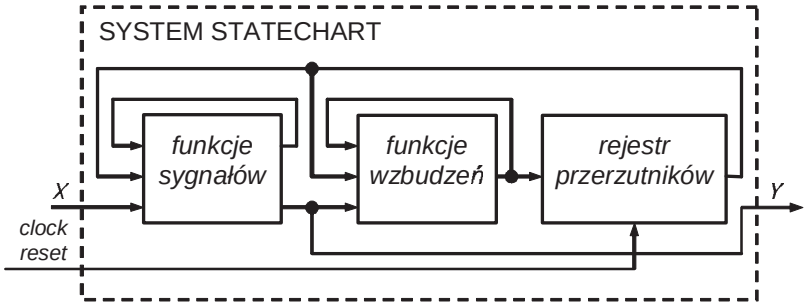
7.4. Model implementacyjny

Wprowadzone pojęcie interpretowanego diagramu statechart (def. 7.1) jest pomostem między modelem matematycznym a fizyczną realizacją oraz podaje sprzętową interpretację takich pojęć jak zdarzenia, zbiór zdarzeń czy etykieta. Stanowi to podstawę do stworzonego opisu równaniami logicznymi, dzięki czemu możliwa jest implementacja elektronicznego układu w takiej postaci jak na rysunku 7.7.

Realizacja układu cyfrowego sterownika binarnego, zrealizowanego z wykorzystaniem specyfikacji zachowania diagramami statechart, polega na przyporządkowaniu przerzutnika w następujący sposób:

- każdemu stanowi diagramu, każdej tranzycji, której realizacja implikuje generowanie zdarzeń,
- każdemu zbiorowi zdarzeń wejściowych przypisanych stanowi,
- każdemu zbiorowi zdarzeń wyjściowych przypisanych stanowi.

Z każdą tranzycją w układzie należy związać określony warunek realizacji tranzycji. Na podstawie przyjętych założeń (o dynamice układu i bezpośredniej implementacji) oraz zasad działania diagramów, tworzone są funkcje wzbudzeń przerzutników stanu i przerzutników pomocniczych (tranzycji i zdarzeń). Zdarzenia są reprezentowane jako sygnały, dla których są tworzone odpowiednie funkcje logiczne. Zbiory zdarzeń są reprezentowane przez wyrażenia logiczne. Wyróżnione zbiory sygnałów stanowią wejście i wyjście z układu. Do rozwiązywania konfliktów między tranzycjami służą predykaty.



Rys. 7.7. Diagram jako układ cyfrowy

7.5. Podsumowanie

O ile w światowej literaturze przedmiotu diagramy statechart, są stale obecne jako środek wizualizacji złożonego zachowania systemów przeróżnego rodzaju, o tyle zagadnienia syntezy diagramów w postaci układowej, w tym w układach reprogramowalnych, są tematyką prawie w ogóle nieobecną. Z ustaleń autora wynika, że najistotniejszymi publikacjami dotyczącymi sprzętowej implementacji diagramów są materiały firmowe związane z pakietem *Statemate MAGNUM* (I-L, 2000*b*) oraz prace (Drusinsky i Harel, 1989*a*; 1989*b*; Drusinsky-Yoresh, 1991; Buchenrieder i in., 1996; Ramesh, 1999), w których jednak zasadniczo pomija się właściwość historii oraz wpływ oddziaływań zwrotnych. Reprezentacja diagramów na poziomie *RTL* opisanych równaniami w formie predykcyjnej, zorientowanej na miejsca, jest więc koncepcją zdecydowanie odmienną nie tylko od metodyki *Statemate MAGNUM*, gdzie powstały model jest opisem w języku *HDL* w formie behawioralnej ale również i od metod pozostałych autorów. W tym względzie przedstawiona propozycja ma charakter nowatorski.

W zależności od tego czy sygnały wyjściowe bezpośrednio zależą od sygnałów wejściowych, układ zachowuje się jak układ typu Mealy'ego lub układ typu Moore'a. Postrzeganie powstałego układu w kategoriach klasycznych modeli, może stanowić podstawę badań nad wykorzystaniem w diagramach statechart metod analizy symbolicznej (podrozdział 8.6).

Rozdział 8

HICOS – SYSTEM AUTOMATYCZNEGO PROJEKTOWANIA HIERARCHICZNYCH STEROWNIKÓW

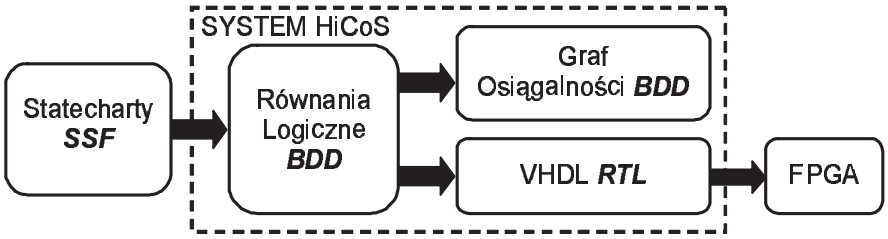
Rozdział ten, obok rozdziału szóstego i siódmego, zawiera dalszą prezentację uzyskanych rezultatów. Opisywany w rozdziale program *HiCoS* jest praktyczną implementacją opracowanych metod i technik. Kolejno są omawiane: specyfikacja danych wejściowych, budowa systemu, format danych wyjściowych oraz dodatkowo nowatorski algorytm generowania grafu osiągalności. Jako ostatnie zostały omówione testowanie systemu oraz wyniki sprzętowej implementacji wybranych modeli testowych.

8.1. Wprowadzenie

Dotychczas powstało bardzo niewiele systemów bezpośrednio wykorzystujących diagramy do projektowania układów cyfrowych implementowanych w strukturach programowalnych. Najbardziej popularnym jest system *Statemate MAGNUM* firmy *I-Logix* (podrozdział 5.3), gdzie zachowanie układu opisane jest w języku *HDL* (*VHDL* lub *Verilog*) z wykorzystaniem m.in. instrukcji *case* i instrukcji sekwencyjnych (*process* czy *always*) (I-L, 2001; I-L, 2000*b*). Autor w swych pracach (m.in.: Łabiak, 2000*b*; Łabiak, 2002*a*), skupił się na takim podzbiorze graficznego języka statechart, który zapewnia, że modelowane układy są binarne (operują wartościami binarnymi) oraz są w pełni modularne (brak możliwości realizacji tranzycji przekraczających granice stanów). W odróżnieniu od programu *Statemate MAGNUM*, w proponowanym systemie przyjęto metodę bezpośredniej implementacji sprzętowej modelowanego układu. Powstały układ cyfrowy jest opisany na poziomie rejestrów i przesłań między nimi (*ang.* Register Transfer Level). Takie podejście charakteryzuje się nie tylko potencjalnie lepszym wykorzystaniem zasobów układu programowalnego, ale też daje łatwą możliwość analizy formalnej (Łabiak, 2001*c*) projektowanego układu, np. z wykorzystaniem grafu osiągalności.

Rysunek 8.1 przedstawia schemat blokowy powstałego systemu realizującego opisane przejście. Wejściem do programu jest plik tekstowy w formacie *SSF*, który jest równoważny postaci graficznej. Program dokonuje transformacji na równania logiczne (podrozdział 7.2), które przekształcone na język *VHDL* (poziom *RTL*) mogą być implementowane w strukturach *FPGA*. Dodatkowo system *HiCoS* oferuje możliwość obejrzenia funkcji charakterystycznej przestrzeni stanów globalnych

i konfiguracji, co w przypadku niedużych przykładów może stanowić dodatkowy środek weryfikacji zachowania modelu.



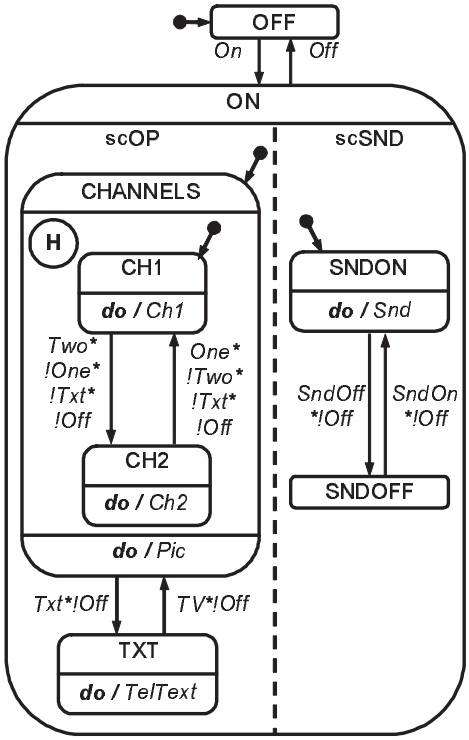
Rys. 8.1. Schemat ideowy sytemu *HiCoS*

8.2. Wejściowy język opisu SSF

Wejście do systemu stanowi plik w autorskim formacie *SSF* (Łabiak, 2000*a*) (*ang.* Statecharts Specification Format), który jest równoważną tekstową reprezentacją postaci graficznej (rys. 8.2 oraz dodatek A). Dla inżyniera projektanta postać tekstowa jest zdecydowanie mniej poglądowa niż diagramy, lecz dla celów badawczych zdecydowano się na mniej atrakcyjną pośrednią formę wprowadzania danych, ale za to tańszą w realizacji i łatwiejszą do szybkiej modyfikacji. W opinii autora, realizacja prostego programu kompilatora jest znacznie mniej pracochłonna niż realizacja złożonego edytora graficznego.

Język *SSF* składa się z 16 słów kluczowych (na rysunku zaznaczone drukiem wytłuszczonym) oraz z 27 produkcji, a jego gramatyka jest gramatyką typu *LL(1)* (Aho i in., 1990; Hopcroft i Ullman, 2003). Projektując język pewne koncepcje słów kluczowych i definicje automatów zostały zaczerpnięte z publikacji (Nazareth i in., 1996), jednak powstała gramatyka jest czymś zupełnie odmiennym niż generalna idea przedstawiona we wzmiankowanym raporcie.

Jako przykład wykorzystania języka niech posłuży diagram modelu zachowania prostego pilota telewizyjnego, posiadającego funkcje włączania telegazety, wyłączania dźwięku i przełączania się między dwoma kanałami telewizyjnymi. Rysunek 8.2 przedstawia dwie równoważne postaci opisu: graficzną i tekstową. Po słowie kluczowym *port* ma miejsce blok deklaracji sygnałów przychodzących do i wychodzących ze sterownika. Podstawowym elementem języka jest specyfikacja automatu sekwencyjnego (np. *scSND*), która składa się z jawnego wymienienia stanów automatu (na rysunku krągłokąty), ustalenia stanu startowego (na rysunku stan ze strzałką dochodzącą, np. *SNDON*) oraz zdefiniowania odwzorowań funkcji etykietowanie tranzycji (def. 6.4 punkt 11) i funkcji etykietowania stanu (def. 6.4 punkt 12). Omawiane funkcje w przykładowym opisie (rys. 8.2) w przypadku automatu *scSND*, nazwano kolejno *tf* i *ef*. Do ustalenia związków współbieżności i hierarchii pomiędzy automatami służą odpowiednio słowa kluczowe *and* oraz *dec by* (na rysunku automaty w relacji współbieżności są przedzielona linią przerywaną). Na przykład diagram statechart *scRm* jest automatem składającym się ze




```
controller TVRm;
port(Onn: in; Off: in; SndOff: in;
  SndOn: in; One: in; Two: in; Txt:
  in; TV: in; Pic: out; TeleTxt:
  out; Snd: out; Ch1: out; Ch2:out);
scCHANNELS=({CH1,CH2},CH1,tf,ef){
  ...
};
scOP=({CHANNELS,TXT},CHANNELS,tf,ef)
{
  ...
};
dec scOP by df {
  df(CHANNELS) = scCHANNELS:h;
  df(TXT) = nodec;
};
scSND=({SNDON,SNDOFF},SNDON,tf,ef) {
  tf(SNDON, SndOff*!Off) => SNDOFF;
  tf(SNDOFF, SndOn*!Off) => SNDON;
  ef(SNDON) = do /{Snd};
};
scRm=({OFF,ON},OFF,tf) {
  tf(OFF, Onn) => ON;
  tf(ON, Off) => OFF;
};
dec scRm by df {
  df(OFF) = nodec;
  df(ON) = and(scOP, scSND);
};
```

Rys. 8.2. Diagram wraz z równoważną postacią tekstową formacie *SSF*

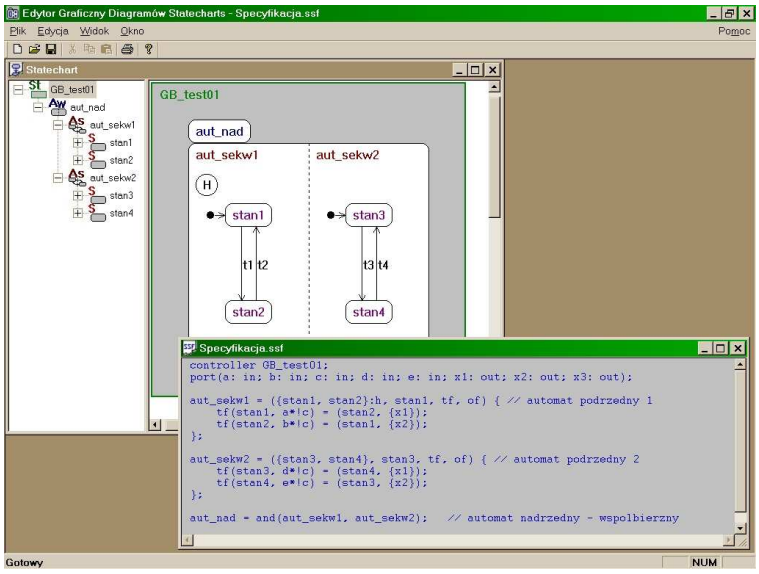
stanów *OFF* i *ON*, któremu przypisano dwa automaty *scOP* i *scSND*. Ustalenie związków hierarchii polega na przyporządkowaniu stanom automatu sekwencyjnego innych podautomatów. Dokonuje się tego poprzez jawne wyspecyfikowanie odwzorowań funkcji dekompozycji (w przypadku automatu *scRm* funkcja dekompozycji nazywa się *df* i jej nazwa jest ustalana po słowie kluczowym *by*). Stanom, które nie są stanami złożonymi, przypisuje się słowo kluczowe *nodec* (*ang.* no decomposition). Ponadto składnia języka *SSF* dopuszcza definiowanie predykatów jako równania logiczne.

8.3. Wizualizacja SSF

W ramach prac nad systemem *HiCoS* prowadzone były również badania nad wizualizacją tekstowej reprezentacji diagramów statechart. W wyniku tych prac powstało oprogramowanie (Bazydło, 2001), które dokonuje zamiany tekstowego opisu w języku *SSF* na postać graficzną oraz umożliwia jego modyfikację w sposób wizualny.

Rysunek 8.3 przedstawia działanie programu. Opis diagramów w języku *SSF* jest poddawany procesowi translacji na wewnętrzną strukturę danych. Diagramy

wyświetlane są w głównym oknie aplikacji, podzielonym na dwie części. W części lewej znajduje się przedstawienie drzewa hierarchii, a w części prawej klasyczna postać graficzna. Nawigowanie drzewem hierarchii, zwijanie i rozwijanie drzewa, edycja, dodawanie i usuwanie węzłów, zsynchronizowane jest z prezentacją postaci graficznej. W programie istnieje również możliwość wygenerowania i obejrzenia pliku *SSF*.



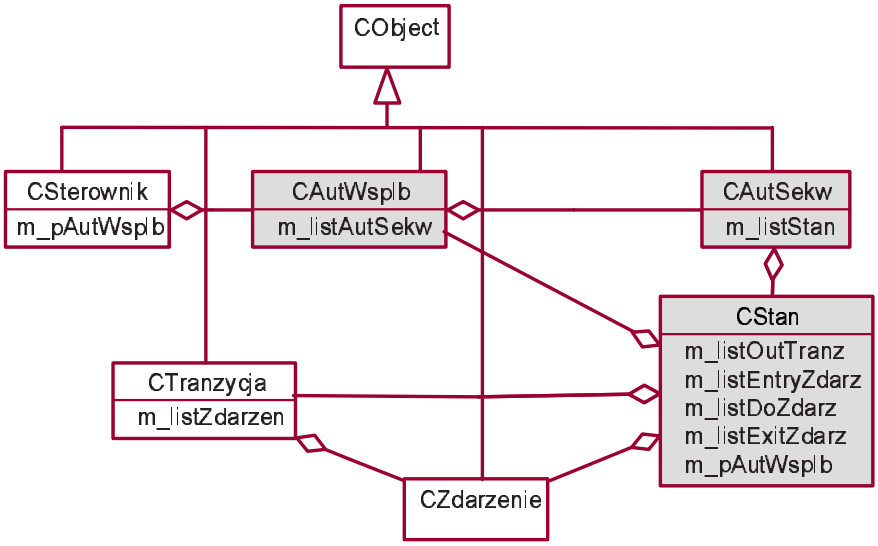
Rys. 8.3. Graficzny edytor diagramów statechart (Bazydło, 2001)

W pracach nad graficznym edytorem wykorzystano parser oraz strukturę danych z systemu *HiCoS* oraz algorytm przedstawiony w (Łabiak i Ludwicki, 2000), a całość również powstała w środowisku *MS VisualC++* z wykorzystaniem biblioteki *MFC* (Toth, 1997).

8.4. Budowa i działanie systemu

System *HiCoS* jest programem komputerowym napisanym w języku *C++* (Stroustrup, 2002) (około 7 tysięci linii własnego kodu), w środowisku Microsoft *Visual C++* z wykorzystaniem biblioteki *MFC*. Plik wejściowy w formacie *SSF* jest poddawany translacji kierowanej składnią (Aho i in., 1990), której celem jest stworzenie wewnętrznej reprezentacji danych. Uproszczony schemat modelu danych programu przedstawiony jest na rysunku 8.4. Rysunek ten przedstawia diagram klas zgodny ze standardem *UML* i został wykonany w programie *Rational Rose* (Rat, 2004; Rat, 2000). Główną klasą jest klasa *CSterownik* zawierająca wskaźnik do obiektu klasy *CAutWsplb*, a ta z kolei posiada listę wskaźników od obiektów reprezentujących podległe automaty sekwencyjne. Każdy obiekt automatu sekwencyjnego posiada listę obiektów odpowiadających stanom, które to posiadają listy

obiektów tranzycji wyjściowych. Zarówno obiekty tranzycji jak i stanów posiadają listy wskaźników na obiekty związane ze zdarzeniami. Zdarzenia generowane są przy realizacji tranzycji lub przy realizacji akcji statycznej, wejściowej lub wyjściowej ze stanu (zdarzenia typu *entry*, *do* i *exit*). Wszystkie klasy w programie są klasami pochodnymi od klasy *CObject*, co ułatwiło testowanie programu i dzięki czemu można było skorzystać z użytecznych operacji wejścia-wyjścia oferowanych przez bibliotekę *MFC*. Taka statyczna reprezentacja danych sprawia, że obiekty klas na rysunku oznaczone szarym tłem, tworzą trójdzielny acykliczny graf odpowiadający drzewu hierarchii modelowanego diagramu. W tak utworzonym grafie węzły są obiektami, a wskaźniki reprezentują relacje typu automat sekwencyjny stany podległe lub stan podległy automat współbieżny.



Rys. 8.4. Uproszczony model danych programu

Powstałe drzewo jest następnie przeglądane w celu utworzenie funkcji wzbudzeń i funkcji sygnałów. Tworzone funkcje w programie reprezentowane są przez binarne diagramy decyzyjne. W pracy autor wykorzystał ich obiektową odmianę z pakietu *CUDD* (Somenzi, 2004), dzięki czemu można było skorzystać z techniki przeciążania operatorów, co diametralnie uprościło kodowanie w programie operacji logicznych. W systemie nie czyniono żadnych założeń o uporządkowaniu zmiennych w diagramie, co oczywiście nie jest bez wpływu na zajętość pamięci komputera i zużycie zasobów w układzie programowalnym. Nie mniej jednak, jak to się okazało na drodze eksperymentalnej, brak takiego założenia nie miał większego wpływu na potwierdzenia tezy pracy.

8.5. Specyfikacja danych wyjściowych z systemu

Główne zadanie jakie przyświecało realizacji systemu było stworzenie możliwości implementacji w układach cyfrowych sterowników specyfikowanych diagramami statechart. Docelowo założono, że projektowane układy będą bezpośrednio implementowane w strukturach programowalnych (np. *FPGA*). Dostępne na rynku komercyjne systemy realizujące syntezę i implementację układów programowalnych (np. *Xilinx Foundation*, *FPGA Express* czy *Leonardo Spectrum*), jako swój główny format danych wejściowych wykorzystują języki *HDL*. Stąd, aby była możliwa współpraca systemu *HiCoS* z oprogramowaniem komercyjnym zdecydowano się, że formatem danych wyjściowych będzie plik zapisany w języku *VHDL*.

Pilot.vhd

```
-- część przerzutnikowa
library IEEE;
use IEEE.std_logic_1164.all;
entity FDD is
  ...
end entity;

architecture FDD of FDD is
begin
  process (CLK, CLR)
    ...
end architecture;

-- część sterownikowa
library IEEE;
use IEEE.std_logic_1164.all;
entity Pilot is
  port(
     reset : in STD_LOGIC;
     clock : in STD_LOGIC;
     Onn: in STD_LOGIC;
     Snd: out STD_LOGIC;
    ...
end Pilot;

architecture Pilot of Pilot is
-- sygnały wyjśc. z przerzutników
signal scSND_SNDON, ...;
-- sygnały wejść przerzutników
signal f_scSND_SNDON, ...;
-- funkcje sygnałów
signal fsx_SndOn, fsx_One, ...;
  component FDD port (...);
    ...
  end componnent;
begin
-- instancjacje komponentu
-- przerzutnika FDD
  FF_SNDON: ffd port map (...);
  ...
-- przypisania funkcji wzbudzeń
-- przerzutników zmiennych stanu
  f_scSND_SNDON <= ...;
  ...
-- przypisania funkcji sygnałów
  fsy_Snd <= ...;
  ...
-- przypisanie sygnałów wyjściowych
  Snd <= fsy_Snd;
  ...
end Pilot;
```

Rys. 8.5. Budowa pliku wyjściowego

Rysunek 8.5 schematycznie przedstawia strukturę pliku wyjściowego. Jak to widać z rysunku, do budowy pliku został wykorzystany tylko taki podzbiór języka, który jest niezbędny do deklaracji przerzutników i zdefiniowania przypisań wyrażeń logicznych do sygnałów. W pliku można wyróżnić dwie części: przerzutnikową i sterownikową. W części przerzutnikowej znajduje się deklaracja jednostki przerzutnika wraz z definicją jego architektury. W części sterownikowej zamieszczona jest deklaracja jednostki sterownika, po czym następuje definicja jego architektury, w której znajdują się instancjacje przerzutników wraz z definicjami funkcji wzbu-

dzeń i sygnałów. Prostota budowy pliku jest konsekwencją przyjętego założenia o bezpośredniej implementacji.

Inną korzyścią płynącą z opracowania modelu układu na poziomie *RTL* jest łatwość realizacji metod analizy symbolicznej. System *HiCoS* posiada możliwość wygenerowania i wyświetlenia funkcji charakterystycznej przestrzeni stanów globalnych układu. Dysponowanie przestrzenią stanów układu w postaci symbolicznej, w której zbiór stanów jest symbolicznie reprezentowany przez funkcję charakterystyczną zmiennych stanu (czyli przerzutników), stwarza możliwości dokładniejszej analizy działania układu (np. wykrywanie tranzycji w konflikcie).

8.6. Algorytm generowania przestrzeni stanów

Istnieje bardzo wiele metod formalnych wykorzystywanych do analizy właściwości modelowanego układu. W grupie tej można wyróżnić zbiór metod, których cechą wspólną jest operowanie grafem osiągalności. Stąd powstała koncepcja opracowania algorytmu generującego symboliczny graf osiągalności. Działanie algorytmu opiera się na poruszaniu po grafie przejść automatu, którego to stany reprezentują stany globalne diagramu statechart.

Techniki symboliczne realizowane przy udziale *BDD* nie tylko z powodzeniem mogą być stosowane do układów sekwencyjnych, ale też okazały się skuteczne w analizie układów cyfrowych opisanych sieciami Petriego (Biliński, 1996). Sieć Petriego stanowi rozwinięcie klasycznego modelu automatu cyfrowego i w literaturze jest nazywana cyfrowym automatem współbieżnym (Adamski, 1990; 1998; Andrzejewski i Łabiak, 1999; Biliński, 1996; Karatkevich i Andrzejewski, 2002; Węgrzyn, 1998*b*; 2001; Wolański, 1998*a*). Sukces technik symbolicznych stosowanych dla układów współbieżnych, zrodził koncepcję zastosowania tychże technik do modelowania układów, których zachowanie opisywane jest diagramami statechart. W wyniku prowadzonych przez autora badań, opracowano metodę generowania funkcji charakterystycznej przestrzeni stanów globalnych układu modelowanego diagramami statechart, składającą się z następujących punktów (Łabiak, 2001*c*):

- określenie funkcji wzbudzeń przerzutników związanych ze stanami,
- określenie funkcji wzbudzeń pomocniczych przerzutników zdarzeń (tranzycji, wejściowych do stanu i wyjściowych ze stanu),
- określenie funkcji sygnałów w modelu,
- reprezentowanie funkcji boolowskich poprzez diagramy *BDD*,
- reprezentowanie zbiorów stanów w postaci funkcji charakterystycznych,
- obliczenie zbioru aktywnych przerzutników w następnych stanach globalnych jako obrazu zbioru aktywnych przerzutników bieżących stanów globalnych dla wektora funkcji wzbudzeń przerzutników.

Rysunek 8.6 w sposób formalny przedstawia omawiany algorytm. Poczynając od stanu globalnego konfiguracji początkowej oraz zbioru wszystkich sygnałów wejściowych, w jednym kroku algorytmu obliczane są zbiory wszystkich możliwych następnych stanów globalnych. Pierwszymi, którzy niezależnie zaproponowali metody obliczania obrazu funkcji, byli Burch (Burch i in., 1990) oraz Coudert (Coudert i in., 1989). Proponują oni obliczanie obrazu funkcji, stosując relację tranzycji i funkcję tranzycji. Algorytm wykorzystujący funkcję tranzycji, obliczający przestrzeń stanów statecharta Z jest następujący (Łabiak, 2001*c*):

```
symbolic_traversal_of_Statechart(Z, initial_marking) {
  X_[G0⟩ = current_marking = initial_marking;
  while (current_marking != ∅) {
    next_marking = image_computation(Z, current_marking);
    current_marking = next_marking * ¬X_[G0⟩ ;
    X_[G0⟩ = current_marking + X_[G0⟩ ;
  }
}
```

Rys. 8.6. Algorytm generowania przestrzeni stanów

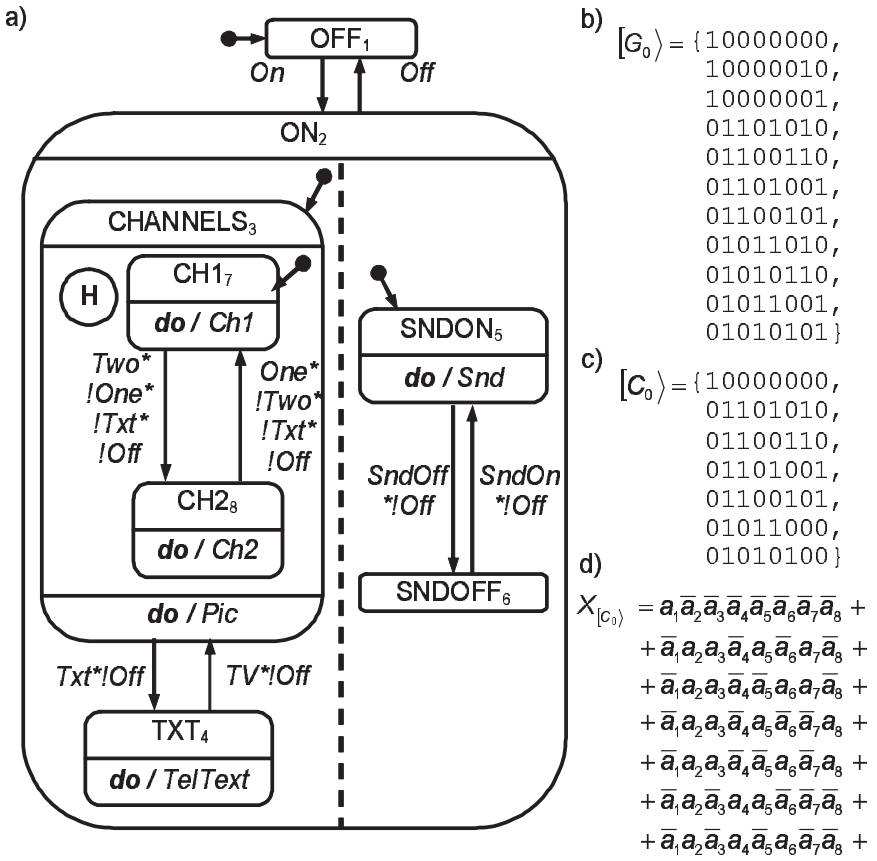
Zmienne w algorytmie zapisane kursywą reprezentują funkcje charakterystyczne zbiorów stanów globalnych. Rysunek 8.7d przedstawia przykładową funkcje charakterystyczną zbioru wszystkich możliwych konfiguracji diagramu z rysunku. Wszystkie zmienne logiczne są reprezentowane przez diagramy *BDD*. Zbiory następnych stanów globalnych (*next_marking*) są obliczane w oparciu o funkcję charakterystyczną zbioru bieżących stanów globalnych (*current_marking*) i wektora funkcji przejść δ_i. Obliczenia te są wykonywane przez funkcję *image_computation* implementującą równania 8.1 i 8.2:

$$next\_marking = \exists_s \exists_x \left(current\_marking * \right.$$
$$\left. * \prod_{i=1}^{n} \left[s_i' \odot \left(current\_marking * \delta_i\left(s, x\right)\right)\right]\right) \tag{8.1}$$

$$next\_marking = next\_marking \left\langle s' \leftarrow s \right\rangle \tag{8.2}$$

gdzie s, s', x oznaczają odpowiednio stan bieżący, stan następny oraz sygnały wejściowe, a n jest liczbą wszystkich przerzutników w układzie. Działania $\exists_s$ i $\exists_x$ reprezentują operacje wygładzania funkcji ze względu na zmienne związane z bieżącymi stanami globalnymi i z sygnałami wejściowymi (Ghosh i in., 1992; Biliński, 1996; de Micheli, 1998), co nieformalnie odpowiada usunięciu zmiennych z zapisu. Symbole $\odot$ i $*$ oznaczają odpowiednio logiczne operatory *XNOR* oraz *AND*, a samo równanie 8.2 realizuje podstawienie, czyli zamianę zmiennych w wyrażeniu.

Mając funkcję charakterystyczną wszystkich możliwych stanów globalnych systemu, możliwe jest na jej podstawie obliczenie zbioru wszystkich możliwych kon-



Rys. 8.7. Przykład analizy symbolicznej dla pilota telewizyjnego: a) diagram statchart, b) zbiór wszystkich stanów globalnych, c) zbiór wszystkich możliwych konfiguracji, d) funkcja charakterystyczna zbioru konfiguracji

figuracji. W podrozdziale 6.1 (def. 6.15) określono konfigurację jako zbiór stanów aktywnych, a sposób obliczania aktywności stanu podano w podrozdziale 7.1 (def. 7.4). Zatem $activecond_i$ jest boolowską funkcję aktywności zależną od sygnałów wejściowych i przerzutników stanów, określoną następująco: $activecond_i : S_z \rightarrow \{0,1\}$ taką, że funkcja ta zwraca wartość **1**, gdy s_i jest aktywny. Wówczas obliczenie funkcji charakterystycznej zbioru wszystkich możliwych konfiguracji systemu polega na obliczeniu obrazu funkcji charakterystycznej zbioru wszystkich stanów globalnych wyznaczonym przez wektor funkcji aktywności stanu ($activecond_i$). Działanie to jest realizowane przez równania 8.3 i 8.4 (Łabiak, 2001*c*), które są modyfikacją równań 8.1 i 8.2.

$$X_{[C_0\rangle} = \exists_s \exists_x \left(X_{[G_0\rangle} * \prod\nolimits_{i=1}^{n} \left[s'_i \odot \left(X_{[G_0\rangle} * activecond_i\,(s,x) \right) \right] \right) \tag{8.3}$$

$$X_{[C_0\rangle} = X_{[C_0\rangle}\,\langle s' \leftarrow s \rangle \tag{8.4}$$

Rysunek 8.7 opisuje zachowanie pilota telewizyjnego. Modelowany system może znaleźć się w 11 stanach globalnych (rys. 8.7b), co odpowiada zbiorowi 7 konfiguracji (rys. 8.7c).

8.7. Testowanie systemu

Zrealizowany system *HiCoS* jest praktyczną implementacją opracowanych metod i technik, zaś wyniki uzyskiwane z systemu mają stanowić empiryczne potwierdzenie przyjętej tezy badań. Zatem, jeśli analiza wyników eksperymentalnych z programu ma stanowić o przyjęciu bądź odrzuceniu tezy badań, w pierwszej kolejności należy dołożyć wszelkich starań, aby zapewnić, że opracowany program komputerowy jest napisany w sposób poprawny. Mając pewność, że program został napisany prawidłowo, można przystąpić do analizy i oceny wyników generowanych przez program i do ostatecznej weryfikacji przyjętej tezy.

Prace testowe zostały podzielona na trzy etapy:

- testowanie czasu realizacji projektu (tryb „debug"),
- testowanie poprawności zaimplementowania opracowanych zasad opisu równaniami logicznymi (dla niedużych przykładów),
- testowanie poprawności opracowanego przekształcenia.

W zakresie testowania czasu realizacji projektu (w środowisku *MS VisualC++* tryb „debug" (Toth, 1997)) m.in. prowadzono następujące czynności:

- krokowe śledzenie działania programu,
- zastosowanie makr *VERIFY* i *ASSERT* (z pakietu *MFC* (Toth, 1997)),
- opracowanie dodatkowych pomocniczych procedur testowych wyświetlających m.in. wewnętrzną strukturę danych (drzewo hierarchii) oraz diagramy *BDD* funkcji logicznych z programu.

W celu stwierdzenie poprawności zaimplementowania opracowanych algorytmów przygotowany został pomocniczy program testowy, w którym „ręcznie" budowano wyrażenia logiczne reprezentujące funkcje wzbudzeń lub sygnałów dla wybranych diagramów testowych i je wyświetlano. Następnie ten sam diagram testowy został poddany automatycznemu przetworzeniu przez program testowany (program *HiCoS*), w którym to w podobny sposób wyświetlono reprezentacje tych samych funkcji wzbudzeń i sygnałów. Następnie dokonano porównania wyświetlonych reprezentacji graficznych z programu testowego i testowanego. Przyjęto, że otrzymanie zgodności wyników oznacza poprawność zaimplementowania opracowanego algorytmu. Porównania były prowadzone przy tych samych uporządkowaniach w obu programach zmiennych w binarnych diagramach decyzyjnych.

Mając pewność co do poprawności zakodowania w języku *C++* realizowanego programu, a zwłaszcza co do implementacji algorytmów, można było przystąpić do

czynności związanych z najistotniejszym z punktu widzenia badań działań testowych, a mianowicie do testowaniem poprawności opracowanego przekształcenia. Na tym etapie były przeprowadzone następujące czynności:

- symulacji modeli w języku *VHDL*,
- analiza symbolicznej przestrzeni stanów globalnych.

Symulacje modeli w języku *VHDL* polegały na prześledzeniu zachowania modeli generowanych przez program *HiCoS*. Przeprowadzono badania dla grupy około 100 przykładów testowych, porównując otrzymane przebiegi czasowe z oczekiwanymi. We wszystkich przypadkach otrzymane wyniki były zgodne z oczekiwanymi. Sesje symulacyjne były prowadzone z wykorzystaniem oprogramowania *Active HDL* firmy *Aldec*.

Dodatkową sposobem potwierdzenia poprawności opracowanych i zaimplementowanych algorytmów jest „ręczna" analiza symbolicznej przestrzeni stanów globalnych modelowanych zachowań (podrozdział 8.6). Ponieważ do wygenerowania funkcji charakterystycznych przestrzeni stanów globalnych i konfiguracji wykorzystywane są funkcje wzbudzeń przerzutników w systemie oraz funkcje sygnałów, zatem można przyjąć, że poprawnie wygenerowane funkcje charakterystyczne symbolicznych przestrzeni stanów globalnych i konfiguracji świadczą o poprawności wszystkich działań na to się składających, w tym samych funkcji wzbudzeń i funkcji sygnałów. Ze względu na oczywiste trudności związane z „ręczną" analizą przeprowadzaną „na piechotę", działania te były prowadzone dla wybranych modeli zachowań, charakteryzujących się niewielką złożonością. Również i w tym przypadku wszystkie otrzymane wyniki okazały się zgodne z oczekiwaniami.

8.8. Przeprowadzone eksperymenty

Przeprowadzone eksperymenty polegały na symulacji i implementacji wybranych układów sterowania w strukturach *FPGA*. Dokładny opis zachowania badanych modeli (słownie, diagram stanów oraz postać w *SSF*) jest zamieszczony w dodatku (dodatek C), a ich wybrane właściwości zestawiono w tabeli 8.1. Tak przygotowane przykłady zostały przetworzone przez program *HiCoS*, który wygenerował kod w języku *VHDL*. Otrzymane modele zostały poddane symulacji w środowisku *Active-HDL* wersja 5.1, celem sprawdzenia poprawności reguł tworzenia równań logicznych. Przykładowe przebiegi czasowe znajdują się w rozdziale 6 (rys. 6.8) i rozdziale 7 (rys. 7.2). We wszystkich przypadkach otrzymane zachowania modeli w języku *VHDL* były całkowicie zgodne z ich graficzną specyfikacją diagramami statechart. Dodatkowo o poprawności opracowanych reguł świadczyły analizy funkcji charakterystycznej przestrzeni stanów globalnych i konfiguracji rozpatrywanych przykładów. Funkcje te są tworzone na podstawie opracowanych równań logicznych, zatem na podstawie poprawności funkcji charakterystycznych można uznać, że równania użyte do ich wygenerowania również są tworzone w sposób prawidłowy.

Tab. 8.1. Wybrane właściwości modeli testowych

Układ	#Stanów	#Tran.	#Aut. sekw.	#Aut. z hist.	#Głęb. hier.	#We.	#Wy.
Garaż	8	7	3	0	2	6	3
Pilot	8	8	4	1	3	8	5
Reaktor	20	19	8	3	3	10	15
OSDW	19	20	4	0	2	16	11

Aby ocenić praktyczną przydatność proponowanej metody projektowej otrzymane modele w języku *VHDL* zostały następnie poddane syntezie z wykorzystaniem licencjonowanego oprogramowania *Leonardo Spectrum* firmy *Exemplar* (Level 3, v2001_1a.32). Implementacja w strukturach programowalnych (*FPGA*) została zrealizowana przy wykorzystaniu oprogramowania *ISE 3.1i* firmy *Xilinx*. Wyniki implementacji – zużycie zasobów oraz maksymalne opóźnienie w układzie – są zestawione w tabelach (tab. 8.2 i tab. 8.3). Proces syntezy i implementacji zostały przeprowadzone przy domyślnych ustawieniach konfiguracyjnych oprogramowania *CAD*.

Tab. 8.2. Wyniki implementacji dla układu XCS05PC84 z rodziny SPARTAN

Układ	#IOB	#CLB	#FF CLB	#LUT 4we	#LUT 2we	%CLB	Max pin delay [ns]
Garaż	11	11	10	22	8	11	2,6
Pilot	15	15	8	25	7	15	2,5
Reaktor	27	64	20	122	30	64	5
OSDW	29	20	19	37	15	20	3,3

Tab. 8.3. Wyniki implementacji dla układu XCV50BG256 z rodziny *VIRTEX*

Układ	#IOB	#SLICEs	%SLICEs	Max pin delay [ns]
Garaż	10	13	1	2.1
Pilot	14	11	1	2.2
Reaktor	27	50	6.5	3.9
OSdW	28	27	3	3.7

Oceniając zużycie zasobów w układach *FPGA* należy wyjaśnić, w jaki sposób kolejne elementy proponowanej ścieżki projektowej wpływają na konsumpcję

przerzutników i konfigurowalnych bloków logicznych. Ogólna liczba wykorzystanych przerzutników jest sumą liczby wszystkich stanów lokalnych modelowanego zachowania, liczby tranzycji generujących zdarzenia oraz akcji wejściowych i wyjściowych przypisanych stanom. O wiele bardziej złożone są zależności dla zużytych bloków logicznych. Podstawowym czynnikiem jest oczywiście złożoność modelowanego zachowania, reprezentowana głównie przez liczbę stanów, tranzycji i sygnałów w modelu. Kolejnym czynnikiem jest sposób opisu modelowanego zachowania równaniami logicznymi. Reguły przedstawione w rozdziale 7 nie są jedyną możliwą propozycją i być może istnieje jakiś inny sposób, który umożliwia tworzenie w ogólności lepszych równań (np.: Drusinsky i Harel, 1989*a*; 1989*b*; Drusinsky-Yoresh, 1991; Ramesh, 1999). Ponadto w testowym systemie *HiCoS* nie rozpatrywano zagadnienia usuwania z równań tzw. „martwej" logiki ani żadnych innych technik optymalizacji. Swoje znaczenie ma również sposób reprezentowania funkcji logicznych w pamięci komputera. W programie nie zastosowano żadnych algorytmów minimalizacji wyrażeń logicznych. Jedynie ze względu na stosunkową łatwość pisania kodu programu, zdecydowano się na binarne diagramy decyzyjne, nie czyniąc żadnych założeń co do uporządkowania zmiennych w diagramie. Ten ostatni element, jak się wydaje, w sposób istotny może wpływać na jakość zużycia zasobów programowalnych.

Podsumowując, otrzymane wyniki implementacji, za wyjątkiem układu reaktor, są porównywalne, aczkolwiek nieco gorsze, z wynikami otrzymanymi dla podobnych układów specyfikowanych sieciami Petriego (Węgrzyn, 1998*b*; Wolański, 1998*b*). Przypadek ten stanowi przyczynek do dalszych prac (podrozdział 9.3). Wziąwszy jednak pod uwagę silne wsparcie dla pojęcia hierarchii oraz możliwość generowania funkcji charakterystycznej przestrzeni stanów globalnych i konfiguracji autor uważa, że proponowana metodologia może stanowić ciekawe narzędzie inżynierskie.

8.9. Podsumowanie

W rozdziale omówiono system automatycznego projektowania hierarchicznych sterowników, będący praktyczną implementacją metod i technik opracowanych w ramach prowadzonych badań. Wejście do systemu stanowi własny format danych — plik tekstowy w formacie *SSF*, równoważny reprezentacji graficznej. Dodatkowo przedstawiono również możliwości wizualizacyjne postaci tekstowej. Plik w języku *SSF*, zawierający modelowane zachowanie, jest wczytywany przez program, który dokonuje analizy leksykalnej i składniowej celem stworzenia wewnętrznej struktury danych. Kolejnym krokiem działania systemu jest zamiana modelowanego zachowania na równania logiczne, reprezentowane w pamięci komputera poprzez binarne diagramy decyzyjne. Wynikiem działania systemu jest wygenerowanie pliku w języku *VHDL*, przeznaczonego do implementacji w strukturach programowalnych lub wyświetlenie funkcji charakterystycznej przestrzeni stanów globalnych i konfiguracji.

Poprawność działania programu została potwierdzona poprzez przeprowadzenie licznych sesji symulacyjnych dla kilkudziesięciu przykładów o z góry narzuco-

nych własnościach. Ponadto program został wykorzystany w procesie dydaktycznym w ramach zajęć laboratoryjnych z przedmiotu Reaktywne Systemy Cyfrowe, prowadzonego na Uniwersytecie Zielonogórskim. W ramach zajęć, w grupie ok. 60 studentów, zadano do zrealizowania to samo proste zadanie, celem zapoznania się z systemem i zasadami jego używania. Wszyscy studenci po upływie około jednej godziny laboratoryjnej otrzymali model w języku *VHDL*, który został przetestowany w symulatorze. Praca z systemem został oceniona bardzo pozytywnie, zwłaszcza w kontekście konkurencyjnych metod projektowych, jak język *VHDL*, czy komercyjny edytor grafu przejść automatu skończonego z pakietu *Active VHDL*. Kolejnym zadaniem był projekt bardziej złożonego reaktywnego systemu sterującego, zadanego przez prowadzącego lub zaproponowanego przez studentów. Również i w tym przypadku uzyskano modele działających układów, co również potwierdza edukacyjną przydatność systemu *HiCoS*.

Zebrane doświadczenia, jak i uzyskane wyniki praktyczne, pozwalają ocenić powstały system jako przydatne narzędzie zarówno inżynierskie jak i dydaktyczne.

Rozdział 9

PODSUMOWANIE I KIERUNKI DALSZYCH PRAC

9.1. Potwierdzenie tezy badań

Niniejsza dysertacja jest wynikiem prowadzonych badań nad zastosowaniem hierarchicznego modelu automatu współbieżnego w projektowaniu cyfrowych układów sterowania. Prowadzone badania miały charakter teoretyczny, praktyczny i eksperymentalny. Potwierdzenie przyjętej tezy głównej zostało dokonane poprzez realizację systemu *CAD*, który umożliwia projektowanie założoną metodą. Powstały system *HiCoS* realizuje przejście od specyfikacji zachowania diagramami statechart (tekstowy format *SSF*) do postaci możliwej do implementacji w strukturach programowalnych (opis w języku *VHDL* na poziomie *RTL*). O jakości pracy z systemem może świadczyć fakt wykorzystania programu w procesie dydaktycznym w ramach przedmiotu Reaktywne Systemy Cyfrowe prowadzonego na Uniwersytecie Zielonogórskim. Praca z systemem zostało oceniona przez studentów bardzo pozytywnie.

Poprawność opracowanych koncepcji i otrzymanych wyników została zweryfikowana licznymi sesjami symulacyjnymi (około stu różnych przykładów), między innymi przykładami opisywanymi w dodatku C.

Najważniejsze częściowe wyniki pracy doktorskiej zostały przedstawione w referatach wygłoszonych na pięciu konferencjach o zasięgu międzynarodowym (Łabiak, 1999; 2001*a*; 2001*b*; 2001*c*; 2003), w dziesięciu wystąpieniach na krajowych konferencjach naukowych (Łabiak, 1998; 2000*a*; 2000*b*; 2001*d*; 2002*a*; 2002*b*; Andrzejewski i Łabiak, 1999; Łabiak i Andrzejewski, 1999; Łabiak i Ludwicki, 2000; Puczyńska i in., 2000) oraz w pracy (Bazydło, 2001).

Praca została zrealizowana w ramach grantu *KBN* nr 4 T11C 006 24.

9.2. Elementy nowatorskie i autorskie

Potwierdzenie tezy głównej, dokonane poprzez realizację celów cząstkowych, zaowocowało opracowaniem nowatorskich metod i koncepcji. W zakresie zadań teoretycznych uzyskano następujące wyniki:

- zaproponowano konkretny sposób wykorzystania technologii *UML* w procesie projektowania reaktywnych układów cyfrowych,
- ustalono te elementy składni i semantyki diagramów, które są istotne w specyfikacji zachowania cyfrowych układów sterowania binarnego,

- opracowano model matematyczny, który dla potrzeb syntezy w strukturach programowalnych w sposób formalny definiuje zarówno składnię jak i zachowanie diagramów,
- opracowano założenia dynamiki oraz zasady realizacji sprzętowej — opis równaniami logicznymi na poziomie *RTL*,
- opracowano algorytm generowanie grafu osiągalności dla hierarchicznego modelu automatu współbieżnego,
- opisano podstawowe reguły przejścia z diagramów statechart na sieć Petriego.

W zakresie realizacji zadań praktycznych autorskimi osiągnięciami są:

- opracowanie gramatyki języka *SSF*, będącego tekstową postacią równoważną postaci graficznej,
- zrealizowanie program kompilatora wykonującego opisywane przejście – od opisu w języku *SSF* do poziomu *RTL* w języku *VHDL*.

Ponadto opracowano i zaimplementowano sprzętowo szereg przykładów o charakterze praktycznym oraz przeprowadzono liczne sesje symulacyjne, a opracowany system *HiCoS*, który znalazł zastosowanie w procesie dydaktycznym, również z powodzeniem może być stosowany jak narzędzie inżyniera projektanta.

9.3. Kierunki dalszych prac

Kierunki dalszych prac można podzielić na działania związane z trzema obszarami badawczymi: poprawa właściwości modelowania, polepszenie wyników syntezy, formalne metody analizy symbolicznej. Za najistotniejszy kierunek dalszych prac uważa się polepszenie możliwości modelowania. W tym obszarze badawczym na pierwszy plan wysuwają się dwa kierunki prac:

- dodanie możliwości operowania tranzycjami przekraczającymi granice stanów,
- wprowadzenie dodatkowych środków synchronizujących takich jak tranzycje złożone i stany synchronizujące.

Przyjęcie proponowanych zmian nie tylko poprawi jakość modelowania, ale zapewni większą zgodność z coraz bardziej popularną technologią *UML*. Pewne prace już zostały rozpoczęte, czego wyrazem jest proponowane rozszerzenie języka *SSF*, przedstawione w dodatku B.

Poprawę jakości implementacji, która jest głównie rozumiana jako ilość zużytych zasobów, proponuje się w pierwszym etapie realizować, między innymi, poprzez przebadanie wpływu uporządkowania zmiennych w binarnych diagramach decyzyjnych na wyniki syntezy. Ten aspekt w prowadzonych badaniach w ogóle nie był rozpatrywany, a jak wiadomo kolejność zmiennych w diagramie *BDD* ma znaczący wpływ na jego wielkość i tym samym na wyniki syntezy. Wydaje

się że interesującym byłoby dodatkowe zaimplementowanie dedykowanych algorytmów minimalizacji, a zwłaszcza algorytmów wykorzystujących dekompozycję funkcji logicznych (Majewski i in., 1992; Łuba, 2001; 2002). Osobnym tematem jest usprawnienie zasad opisu równaniami logicznymi i ich odwzorowania w taki sposób, aby pominąć logikę nadmiarową np. związaną z przerzutnikami przyporządkowanymi stanom abstrakcyjnym bez historii i bez zdarzeń, które często do układu wnoszą jedynie redundantną informację.

Ostatnim proponowanym obszarem badań jest opracowanie algorytmów operujących metodami symbolicznymi. Opisany w pracy algorytm generowania grafu osiągalności może tu stanowić punkt startowy dla dalszych prac nad weryfikacją poprawności. Przykładowym wykorzystaniem algorytmu może być jego modyfikacja polegająca na przeglądaniu przestrzeni stanów układów z jednoczesnym dokładnym wyszukiwaniem tranzycji będących w konflikcie. Innym zastosowaniem jest sprawdzanie właściwości żywotności i bezpieczeństwa. Ponadto zdaniem autora interesującym wydaje się wykorzystanie algorytmu hierarchicznego grafu znakowań (Miczulski, 2002*a*), który już znalazł zastosowanie w przypadku pewnych hierarchicznych sieci Petriego. Przesłankę stanowią tu istniejące podobieństwa hierarchicznych sieci Petriego i diagramów statechart. Dalszym nurtem poszukiwań, podobnie jak czyni to autor pracy (Miczulski, 2002*b*), jest rozpatrzenie stosowania innych odmian diagramów decyzyjnych, takich jak np. *KFDD* (ang. *Kronecker Functional Decision Diagram*) czy *ZBDD* (ang. *Zero Suppressed Binary Decision Diagram*).

Dodatek A

GRAMATYKA JĘZYKA SSF

Gramatyka języka *Statecharts Specification Format* (Łabiak, 2000*a*) jest gramatyką typu *LL(1)* (Aho i in., 1990; Hopcroft i Ullman, 2003) (za wyjątkiem produkcji *wyr-bool*), składającą się z 16 słów kluczowych i 27 produkcji. Czcionką pochyloną niewytłuszczoną oznaczono symbole nieterminalne. Czcionką wytłuszczoną symbole terminalne i słowa kluczowe. Czcionką nie pochyloną oznaczono metasymbole symbole pomocnicze ([,], |): ciąg ujęty w nawiasy [] oznacza zero lub więcej powtórzeń, symbol | oznacza alternatywę. Indeks dolny oznacza interpretacje semantyczną i tak na przykład $nazwa_{fn-tr}$ oznacza nazwę funkcji tranzycji. Dodatkowo wymaga się, aby nazwy automatów (sekwencyjnego i współbieżnego), nazwy stanów, nazwy funkcji (tranzycji i dekompozycji hierarchii) i sygnałów zostały przed użyciem zdefiniowane. Ponadto, dopuszczalne jest stosowanie komentarzy w stylu języka *C++* (// – komentarz jednoliniowy, /*...*/ – komentarz blokowy).

program:
controller $nazwa_{ctrl}$ **;** *dekl-port blok-dekl*

nazwa:
litera | *nazwa* [*cyfra*] | *nazwa* [*nazwa*] | *nazwa* [_]

dekl-port:
port ([$nazwa_{sygn}$ **:** *rodz-sygn* [**;** $nazwa_{sygn}$ **:** *rodz-sygn*]]**) ;**

rodz-sygn:
in | **in comb** | **out** | **out reg** | **out comb**

blok-dekl:
def-aut **;** [*blok-dekl*]
dekomp-hier **;** [*blok-dekl*]
dekl-sygn **;** [*blok-dekl*]
równ-bool **;** [*blok-dekl*]

def-aut:
$nazwa_{automatu}$ **=** *spec-aut*

spec-aut:
spec-aut-sekw | *spec-aut-współb*

spec-aut-sekw:
(*lista-stanów***,** $nazwa_{st-pocz}$**,** $nazwa_{fn-tr}$**)** *def-fn-aut*

(*lista-stanów*, $nazwa_{st-pocz}$, $nazwa_{fn-tr}$, $nazwa_{fn-wy}$) *def-fn-aut*

lista-stanów:
{[*stan* [, *stan*]]**}** | **{**[*stan* [, *stan*]]**} : h** | **{**[*stan* [, *stan*]]**} : h***

stan:
$nazwa_{stanu}$ | $nazwa_{stanu}$ **: h** | $nazwa_{stanu}$ **: h***

def-fn-aut:
{[*odwzr-fn-aut* **;**]**}**

odwzr-fn-aut:
odwzr-fn-tr | *odwzr-fn-wy*

odwzr-fn-tr:
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$, *wyr-bool*) => $nazwa_{st-konc}$
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$, *wyr-bool*) => **endst**
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$, *wyr-bool*) => ($nazwa_{st-konc}$, *lista-sygn-wy*)
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$, *wyr-bool*) => (**endst**, *lista-sygn-wy*)
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$) => $nazwa_{st-konc}$
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$) => **endst**
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$) => ($nazwa_{st-konc}$, *lista-sygn-wy*)
$nazwa_{fn-tr}$(*nazwa*$_{st-pocz}$) => (**endst**, *lista-sygn-wy*)

wyr-bool:
wyr-bool **+** *składnik* | *składnik*

składnik:
składnik ***** *czynnik* | *czynnik*

czynnik:
$nazwa_{sygn}$ | **0** | **1** | **!** | *czynnik* | (*wyr-bool*)

lista-sygn-wy:
{[*sygn-wy* [, *sygn-wy*]]**}**

sygn-wy:
$nazwa_{sygn-wy}$ | **!** $nazwa_{sygn-wy}$

odwzr-fn-wy:
$nazwa_{fn-wy}$(*nazwa*$_{stanu}$) **= entry** / *lista-sygn-wy*
$nazwa_{fn-wy}$(*nazwa*$_{stanu}$) **= do** / *lista-sygn-wy*
$nazwa_{fn-wy}$(*nazwa*$_{stanu}$) **= exit** / *lista-sygn-wy*

spec-aut-wspłb:
and (*aut-sekw*, *aut-sekw* [, *aut-sekw*])

aut-sekw:
$nazwa_{aut-sekw}$
$nazwa_{aut-sekw}$ **: h**
$nazwa_{aut-sekw}$ **: h***
spec-aut-sekw

spec-aut-sekw **: h**
spec-aut-sekw **: h***

dekomp-hier:
dec *aut-sekw* **by** $nazwa_{fn-hier}$ *def-fn-hier*

def-fn-hier:
{[*odwzr-fn-hier* **;**]**}**

odwzr-fn-hier:
$nazwa_{fn-hier}(nazwa_{stanu})$ = *aut-sekw*
$nazwa_{fn-hier}(nazwa_{stanu})$ = *aut-współb*
$nazwa_{fn-hier}(nazwa_{stanu})$ = **nodec**

aut-współb:
$nazwa_{aut-wsplb}$
$nazwa_{aut-wsplb}$ **: h**
$nazwa_{aut-wsplb}$ **: h***
spec-aut-współb
spec-aut-współb **: h**
spec-aut-współb **: h***

dekl-sygn:
signal $nazwa_{sygn}$ [**,** $nazwa_{sygn}$]

równ-bool:
$nazwa_{sygn}$ = *wyr-bool*

Dodatek B

PROPONOWANE ROZSZERZENIA JĘZYKA SSF

Język *SSF* w swej rozszerzonej wersji powinien wspierać opis tych elementów diagramów zgodnych z normą *UML* (UML, 2003), które z punktu widzenia modelowania sterowania są najistotniejsze. Zdaniem autora do tych elementów należą: tranzycje przekraczające granice stanów, tranzycje złożone i stany synchronizujące. Diagramami, które ilustrują te elementy opisu są diagramy przedstawione na rysunkach 3.6 i 3.7.

Kod B.1. Tranzycje przekraczające granice stanu – rozszerzona wersja języka *SSF* dla diagramu z rysunku 3.6

```
controller przyklad;
port();
scA = ({A1, A2}, A1, tf) {
    tf(A1) => A2;
};
scB = ({B1, B2}, B1, tf) {
    tf(B1) => B2;
};

scDzialanie = ({UstawieniaPoczatkowe, Przetwarzanie,
KoncowePorzadki}, UstawieniaPoczatkowe, tf)
    // deklaracja tranzycji złożonych
    transition T1, T2;

    tf(UstawieniaPoczatkowe) => T1;
    // odwołania do stanów "obcych"
    tf(T1) => scA::A1;
    tf(T1) => scB::B1;
    tf(scA::A2) => T2;     // pseudotranzycja prosta
    tf(scB::B2) => T2;     // pseudotranzycja prosta
    tf(T2) => KoncowePorzadki;
};

dec scDzialanie by df {
    df(UstawieniaPoczaktowe) = nodec;
    df(Przetwarzanie) = and(scA, scB);
    df(KoncowePorzadki) = nodec;
};
```

Opisywanie tranzycji przekraczających granice stanu polega na tym, że definiując automat sekwencyjny trzeba się odwołać do stanu zdefiniowanego w innym automacie. Aby tego dokonać należy nazwę stanu „obcego" poprzedzić operatorem *::* oraz nazwą automatu gdzie stan został zadeklarowany (kod B.1).

W celu zdefiniowania tranzycji złożonej należy posłużyć się słowem kluczowym *transition*, po którym podaje się nazwę tranzycji. W definicji automatu nazwą tranzycji operuje się tak samo jak nazwą stanu, tworząc przy jej użyciu pseudotranzycje proste. Zbiór takich pseudotranzycji prostych składa się na tranzycję złożoną. Predykat tak definiowanej tranzycji złożonej jest iloczynem predykatów pseudotranzycji prostych (kod B.1 i B.2).

Kod B.2. Stany synchronizujące – rozszerzona wersji języka *SSF* dla diagramu z rysunku 3.7

```
controller Budowa;
port();
// definicja automatu współbieżnego
// wraz ze stanami synchronizującymi SYNC1 i SYNC2
scPraceNaziemne = and(scElektryka, scStanSurowy)
    {SYNC1, SYNC2};

// definicja składowego automatu sekwenycjnego
scElektryka = ({ElektrykaWFundamentach, ElektrykaWSzkielecie,
    ElektrykaNaZewnatrz), ElektrykaWFundamentach, tf) {
    // deklaracje tranzycji złożonych
    transition T1, T2;

    // odwołanie do stanu "obcego" SYNC1
    tf(scPraceNaziemne::SYNC1) => T1;
    tf(ElektrykaWFundamentach) => T1;
    tf(T1) => ElektrykaWSzkielecie;
    tf(ElektrykaWSzkielecie) => T2;
    // odwołanie do stanu "obcego" SYNC2
    tf(T2) => scPraceNaziemne::SYNC2;
    tf(T2) => ElektrykaNaZewnatrz;
    tf(ElektrykaNaZewnatrz) => endst;
};

// definicja składowego automatu sekwenycjnego
scStanSurowy = ({Szkielet, Dach, Sciany}, Szkielet, tf) {
    // deklaracje tranzycji złożonych
    transition T1, T2;

    tf(Szkielet) => T1;
    // odwołanie do stanu "obcego" SYNC1
    tf(T1) => scPraceNaziemne::SYNC1;
    tf(T1) => Dach;
    tf(Dach) => T2;
    // odwołanie do stanu "obcego" SYNC2
```

```
    tf(scPraceNaziemne::SYNC2) => T2;
    tf(T2) => Sciany;
    tf(Sciany) => endst;
}

Budowa = ({Fundamenty, PraceNaziemne, Odbiór},
    Fundamenty, tf) {
    tf(Fundamenty) => PraceNaziemne;
    tf(PraceNaziemne) => Odbiór;
    tf(Fundamenty) => endst;
};

dec Budowa by df {
    df(Fundamenty) = nodec;
    df(PraceNaziemne) = scPraceNaziemne;
    df(Odbiór) = nodec;
};
```

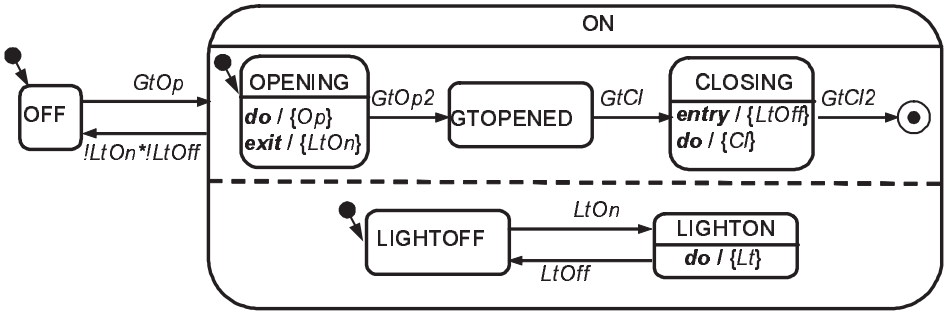
Definiowanie automatu współbieżnego ze stanami synchronizującymi wymaga operowania tym samym stanem we wszystkich jego składowych automatach sekwencyjnych. W związku z tym definicje automatów składowych muszą być poprzedzone definicją automatu współbieżnego, jawnie wymieniającą wszystkie stany synchronizujące (kod B.2).

Dodatek C

PRZYKŁADY

C.1. Brama garażowa

Zadaniem układu jest sterowanie prostą bramą garażową. Użytkownik ma do dyspozycji możliwość otwarcia bramy (na diagramie jest to reprezentowane przez zdarzenie *GtOp*), zamknięcia bramy (zdarzenie *GtCl*) oraz zapalenia i zgaszenia światła w garażu (odpowiednio zdarzenia *LtOn* i *LtOff*). W stanie początkowym *OFF* układ oczekuje na żądanie otwarcia. Po wystąpieniu zdarzenia *GtOp* układ przechodzi do stanów *OPENING* i *LIGHTOFF* (obydwa w stanie *ON*) w którym brama jest otwierana (układ ustawia sygnał *Op*). W trakcie otwierania bramy światło w garażu jest zgaszone. Następnie do układu z czujnika otwarcia bramy kierowane jest zdarzenie *GtOp2*, w wyniku którego sterowanie opuszcza stan OPENING i generowane jest zdarzenie *LtOn* (realizacja akcji wyjściowej). Wystąpienie tego zdarzenia powoduje w sąsiednim automacie aktywację stanu *LIGHTON* i ustawienie sygnału *Lt*. Światło zostaje zapalone. Pojawienie się zdarzenia *GtCl* powoduje rozpoczęcie sekwencji zamykania bramy. Aktywowany jest stan *CLOSING* oraz, na skutek realizacji akcji wejściowej (wygenerowanie zdarzenia *LtOff*), zostaje zgaszone światło (w sąsiednim automacie sterowanie przechodzi ze stanu *LIGHTON* do stanu *LIGHTOFF*). Zamknięcie bramy sygnalizowane jest pojawieniem się zdarzenia *GtCl2*, w wyniku czego sterowanie przechodzi do stanu końcowego i następnie do stanu *OFF*, stanu początkowego układu.



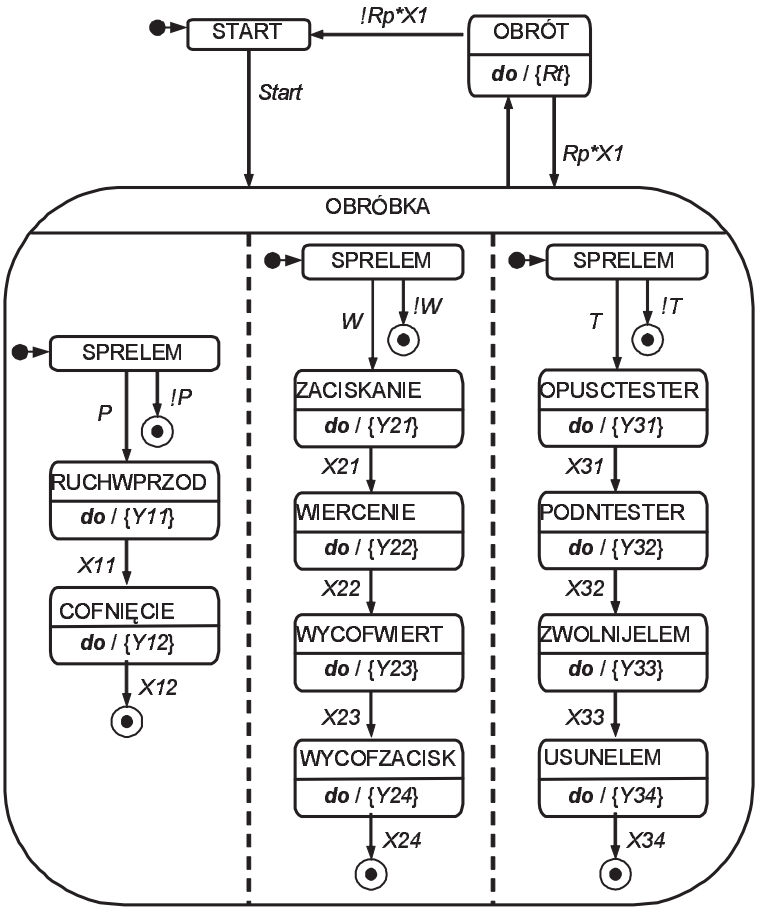
Rys. C.1. Brama garażowa – diagram statechart

Kod C.1. Brama garażowa – postać tekstowa w formacie *SSF*

```
controller Garage;
port(LtOn: in; LtOff: in; GtOp: in; GtCl: in; GtOp2: in;
     GtCl2: in; Lt: out; Op: out; Cl: out);

scLIGHT = ({LIGHTOFF, LIGHTON}, LIGHTOFF, tf, ef) {
    // zapalenie swiatła
    tf(LIGHTOFF, LtOn) => LIGHTON;
    ef(LIGHTON) = do / {Lt};
    // zgaszenie swiatła
    tf(LIGHTON, LtOff) => LIGHTOFF;
};

scGATE = ({OPENING, GTOPENED, CLOSING}, OPENING, tf, ef) {
    // otwieranie bramy
    tf(OPENING, GtOp2) => GTOPENED;
    ef(OPENING) = do / {Op};
    ef(OPENING) = exit / {LtOn};
    // brama otwarta
    tf(GTOPENED, GtCl) => CLOSING;
    // zamykanie bramy
    tf(CLOSING, GtCl2) => endst;
    ef(CLOSING) = entry / {LtOff};
    ef(CLOSING) = do / {Cl};
};

scSYSTEM = ({OFF, ON}, OFF, tf) {
    tf(OFF, GtOp) => ON;
    tf(ON, !LtOn*!LtOff) => OFF;
};

dec scSYSTEM by df {
    df(OFF) = nodec;
    df(ON) = and(scLIGHT, scGATE);
};
```

C.2. Obrotowe stanowisko do wiercenia

Obrotowe stanowisko do wiercenia składa się z obrotowego stołu z trzema gniazdami, w których umieszczane są obrabiane elementy (Węgrzyn, 1998*a*). Poprzez obrót stołu, za każdym razem o 120°, obrabiane elementy trafiają kolejno do trzech stanowisk obróbczych: mocującego, wiertniczego, testującego. Zadaniem układu sterującego jest kierowanie obróbczym cyklem technologicznym. Tabela C.1 zawiera szczegółowy wykaz sygnałów układu.



Rys. C.2. Obrotowe stanowisko do wiercenia – diagram statechart

Tab. C.1. Opis sygnałów wejściowych sterownika stanowisko do wiercenia

Nazwa sygnału	Opis sygnału
START	sygnał zezwalający na rozpoczęcie operacji
RP	sygnał zezwalający na powtórzenie operacji
X1	obrót zakończony
P, W, T	sygnały zezwalające na operację (podaw., wierc., test.)
X11	element załadowany
X12	podajnik cofnięty
X21	element zaciśnięty
X22	koniec wiercenia
X23	wiertarka w położeniu początkowym

Tab. C.1. Opis sygnałów wejściowych sterownika stanowisko do wiercenia – ciąg dalszy

Nazwa sygnału	Opis sygnału
X24	koniec ściskania
X31	tester opuszczony
X32	tester podniesiony
X33	element zwolniony
X34	element usunięty
RT	obrót taśmy
Y11	ruch podajnika do przodu
Y12	cofnięcie podajnika
Y21	zaciskanie elementu
Y22	wiercenie
Y23	wycofywanie wiertarki
Y24	wycofywanie urządzenia ściskającego
Y31	opuszczanie testera
Y32	podnoszenie testera
Y33	zwolnienie elementu
Y34	usunięcie elementu

Kod C.2. Obrotowe stanowisko do wiercenia — postać tekstowa w formacie *SSF*

```
controller ObrStdWier;
port(Rt: out; Y11: out; Y12: out; Y21: out; Y22: out;
     Y23: out; Y24: out; Y31: out; Y32: out; Y33: out;
     Y34: out; Start: in; Rp: in; X1: in; P: in; W: in;
     T: in; X11: in; X12: in; X21: in; X22: in; X23: in;
     X24: in; X31: in; X32: in; X33: in; X34: in);

// proces 1: mocowanie elementu
sc_1=({SPRELEM, RUCHWPRZOD, COFNIECIE}, SPRELEM, tf_1, of_1) {
     tf_1(SPRELEM, !P) => endst;
     tf_1(SPRELEM, P) => RUCHWPRZOD;
     of_1(RUCHWPRZOD) = do / {Y11};
     tf_1(RUCHWPRZOD, X11) => COFNIECIE;
     of_1(COFNIECIE) = do / {Y12};
     tf_1(COFNIECIE, X12) => endst;
};

// proces 2: wiercenie elementu
sc_2 = ({SPRELEM, ZACISKANIE, WIERCENIE, WYCOFWIERT,
      WYCOFZACISK}, SPRELEM, tf_2, of_2) {
     tf_2(SPRELEM, W) => ZACISKANIE;
     tf_2(SPRELEM, !W) => endst;
     of_2(ZACISKANIE) = do / {Y21};
     tf_2(ZACISKANIE, X21) => WIERCENIE;
```

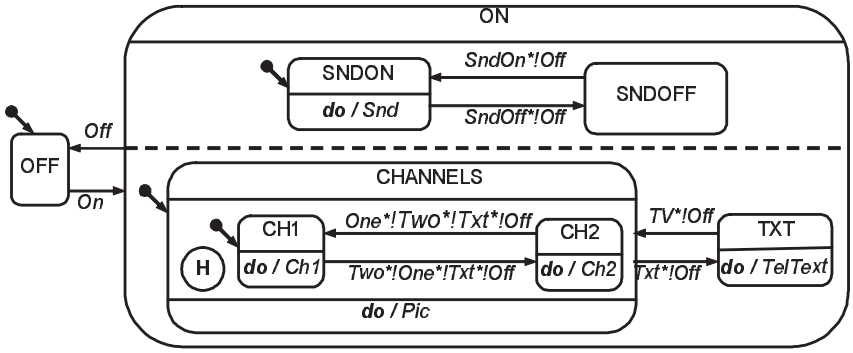
```
    of_2(WIERCENIE) = do / {Y22};
    tf_2(WIERCENIE, X22) => WYCOFWIERT;
    of_2(WYCOFWIERT) = do / {Y23};
    tf_2(WYCOFWIERT, X23) => WYCOFZACISK;
    of_2(WYCOFZACISK) = do / {Y24};
    tf_2(WYCOFZACISK, X24) => endst;
};

// proces 3: testowanie elementu
sc_3 = ({SPRELEM, OPUSCTESTER, PODNTESTER, ZWOLNIJELEM,
    USUNELEM}, SPRELEM, tf_3, of_3) {
    tf_3(SPRELEM, T) => OPUSCTESTER;
    tf_3(SPRELEM, !T) => endst;
    of_3(OPUSCTESTER) = do / {Y31};
    tf_3(OPUSCTESTER, X31) => PODNTESTER;
    of_3(PODNTESTER) = do / {Y32};
    tf_3(PODNTESTER, X32) => ZWOLNIJELEM;
    of_3(ZWOLNIJELEM) = do / {Y33};
    tf_3(ZWOLNIJELEM, X33) => USUNELEM;
    of_3(USUNELEM) = do / {Y34};
    tf_3(USUNELEM, X34) => endst;
};

sc_Obrobka= and(sc_1, sc_2, sc_3);

sc_OSdW=({START, OBROT, OBROBKA}, START, tf_OSdW, of_OSdW) {
    tf_OSdW(START, Start) => OBROBKA;
    tf_OSdW(OBROT, !Rp*X1) => START;
    tf_OSdW(OBROT, Rp*X1) => OBROBKA;
    tf_OSdW(OBROBKA) => OBROT;
    of_OSdW(OBROT) = do / {Rt};
};

dec sc_OSdW by df_OSdW {
    df_OSdW(START) = nodec;
    df_OSdW(OBROT) = nodec;
    df_OSdW(OBROBKA) = sc_Obrobka;
};
```

C.3. Pilot telewizyjny

Zadaniem pilota telewizyjnego (przykład opracowany na podstawie (Nazareth i in., 1996)) jest zdalne sterowanie bardzo prostym odbiornikiem telewizyjnym. Odbiornik ten jest zdolny do odbierania dwóch kanałów telewizyjnych oraz oferuje możliwość przeglądania telegazety. Atrybut historii nadany stanom *CH1* i *CH2* powoduje, że po powrocie sterowania ze stanu *TXT* do stanu *CHANNELS* aktywnym staje się stan związany z ostatnio oglądanym kanałem. Równocześnie użytkownik

ma możliwość włączania i wyłączania dźwięku.

Warto tu zwrócić uwagę na rolę atrybutu historii. W sytuacji gdy jest oglądany obraz jednego z kanałów, użytkownik ma możliwość przełączenia się w tryb teletekstu (stan *TXT*). Po czym, co jest oczywiste, sterowanie ma wrócić do stanu związanego z ostatnio oglądanym kanałem. Takie pożądane zachowanie zostało uzyskane właśnie dzięki atrybutowi historii.



Rys. C.3. Pilot telewizyjny – diagram statechart

Kod C.3. Pilot telewizyjny — postać tekstowa w formacie *SSF*

```
controller TVRm;
port(Onn: in; Off: in; SndOff: in; SndOn: in; One: in;
     Two: in; Txt: in; TV: in; Pic: out; TeleTxt: out;
     Snd: out; Ch1: out; Ch2: out);

scCHANNELS = ({CH1, CH2}, CH1, tf, ef) {
     tf(CH1, Two*!Txt*!Off) => CH2;
     tf(CH2, One*!Txt*!Off) => CH1;
     ef(CH1) = do / {Ch1};
     ef(CH2) = do / {Ch2};
};

scOP = ({CHANNELS, TXT}, CHANNELS, tf, ef) {
     tf(CHANNELS, Txt*!Off) => TXT;
     tf(TXT, TV*!Off) => CHANNELS;
     ef(CHANNELS) = do / {Pic};
     ef(TXT) = do / {TeleTxt};
};

dec scOP by df {
     df(CHANNELS) = scCHANNELS:h;
     df(TXT) = nodec;
};

scSND = ({SNDON, SNDOFF}, SNDON, tf, ef) {
```

```
    tf(SNDON, SndOff*!Off) => SNDOFF;
    tf(SNDOFF, SndOn*!Off) => SNDON;
    ef(SNDON) = do / {Snd};
};

scRm = ({OFF, ONN}, OFF, tf) {
    tf(OFF, Onn) => ONN;
    tf(ONN, Off) => OFF;
};

dec scRm by df {
    df(OFF) = nodec;
    df(ONN) = and(scOP, scSND);
};
```

C.4. Reaktor

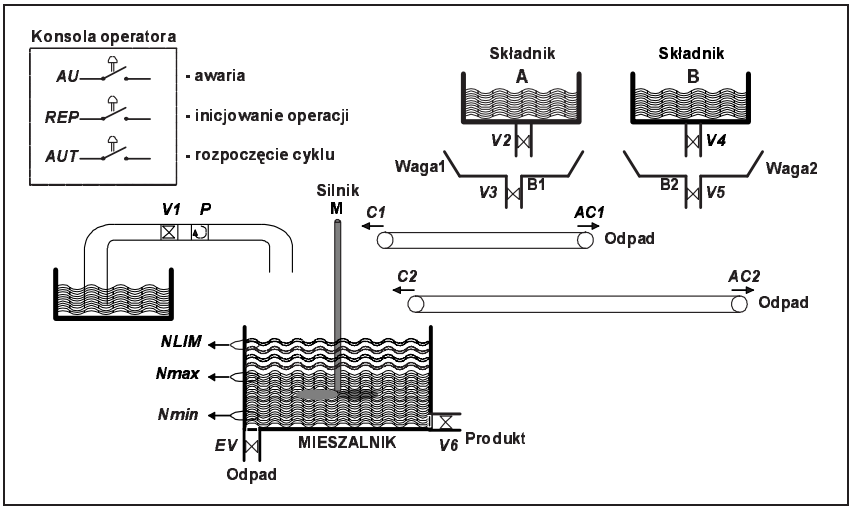
Reaktor jest przykładem technologicznego procesu mieszania dwu składników o ściśle odmierzanych ilościach w środowisku wodnym (Węgrzyn, 1998*a*). Proces ten składa się z trzech etapów:

- przygotowywanie wody i substratów o zadanej masie – stan *NAPELNIANIE*,
- mieszanie składników w mieszalniku w zadanym okresie czasu – stan *PROCES*,
- wstępnego przygotowania procesu polegającego na usunięciu pozostałości starych substratów w wagach i materiału w pojemniku mieszalnika – stan *INICJOWANIE*.

Osoba nadzorująca przebieg procesu ma do dyspozycji konsolę operatorską umożliwiającą: sygnalizowanie awarii (sygnał AU), żądanie inicjowania procesu (sygnał REP), rozpoczęcie procesu (sygnał AUT). Jak widać to ze schematu procesu (rys. C.4) operator ma możliwość zgłoszenia awarii w trakcie napełniania zbiorników oraz w trakcie realizacji procesu chemicznego. Do urządzenia sterującego dochodzą sygnały z czujników poziomu i ciężaru (*B1*, *B2*, *NLIM*, *Nmax*, *Nmin*), rozmieszczonych w instalacji oraz sygnały z zegarów odmierzających zadane interwały czasowe (*FT1*, *FT2*). Ze sterownika wysyłane są żądania dla zaworów pomp, taśmociągów, silnika mieszalnika i zegarów (*V1*, *V2*, *V3*, *V4*, *V5*, *V6*, *EV*, *C1*, *AC1*, *C2*, *AC2*, *M*, *TM1*, *TM2*). Rysunek C.5 przedstawia schemat blokowy sterownika.

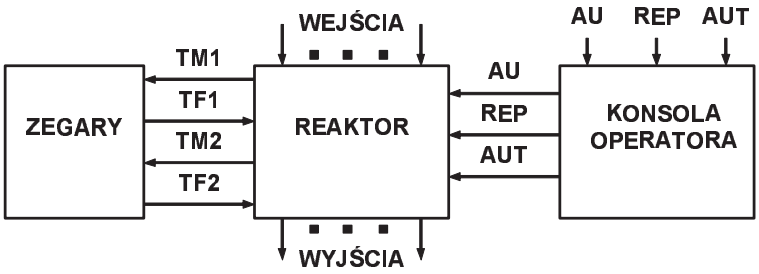
Układ rozpoczyna swoje działanie od stanu *START* (rys. C.6) przechodząc, jeżeli nie ma sygnału o awarii, do stanu *INICJOWANIE* gdzie następuje opróżnienie zbiornika głównego oraz uprzątnięcie taśmociągów z resztek pozostałych po poprzednim cyklu procesu. Następnie sygnałem AUT rozpoczynane jest przygotowywanie składników reakcji – stan *NAPEŁNIANIE*. W tym stanie zgłoszenie

awarii (zdarzenie *AU*) powoduje przejście do stanu *RESTART*, a po jej usunięciu powrót do stanów ostatnio aktywnych stanu *INICJOWANIE*. Taki powrót sterowania możliwy jest dzięki obecności atrybutu historii. Po odpowiednim napełnieniu wszystkich zbiorników rozpoczynany jest proces chemiczny, w którym czas trwania reakcji (stan *REAKCJA*) odmierzany jest przez zewnętrzne zegary. Ponowne rozpoczęcie cyklu odbywa się po wprowadzeniu sygnału *AUT* z konsoli operatora, pod warunkiem, że zbiornik główny został opróżniony do żądanego poziomu (sygnał *Nmin*). Układ ponownie przechodzi do stanu *NAPELNIANIE*.

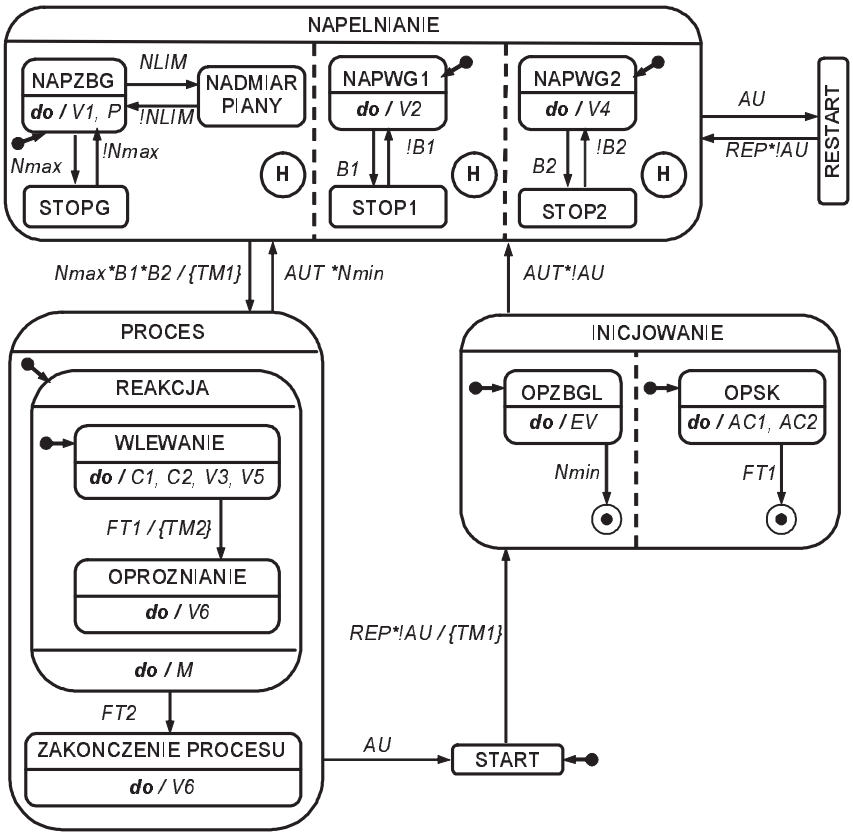


Rys. C.4. Reaktor – schemat procesu technologicznego (Węgrzyn, 1998*a*)

Diagram z rysunku C.6 charakteryzuje się niederminizmem, tranzycje na nim zawarte mogą znajdować się w konflikce. Ze względu na czytelności jako predykaty tranzycji na rysunku (rys. C.6) zaznaczone tylko te sygnały, które posiadają istotne znaczenie dla zrozumienia istoty działanie układu. Pełna deterministyczna wersja sterownika, opisana jest w tekstowym formacie *SSF* i przedstawiona jest na rysunku C.4.



Rys. C.5. Schemat blokowy sterownika reaktora (Węgrzyn, 1998*a*)



Rys. C.6. Reaktor – diagram statechart (M. Adamski)

Kod C.4. Reaktor – postać tekstowa w formacie *SSF*

```
controller reaktor;
port(AU: in; REP: in; AUT: in; B1: in; B2: in; NLIM: in;
     Nmax: in; Nmin: in; M: out; V1: out; P: out; V2: out;
     V4: out; V3: out; V5: out; C1: out; C2: out; AC1: out;
     AC2: out; EV: out; V6: out; TM1: out; TM2: out; FT1: in;
     FT2: in);

// napełnianie zbiornika głównego
sc_NAPZBG = ({NAPZBG, NADMIAR_PIANY, STOPG}, NAPZBG, tf, of) {
     of(NAPZBG) = do / {V1, P};
     tf(NAPZBG, Nmax*!AU) => STOPG;
     tf(STOPG, !Nmax*!AU) => NAPZBG;
     tf(NAPZBG, NLIM*!Nmax*!AU) => NADMIAR_PIANY;
     tf(NADMIAR_PIANY, !NLIM*!AU) => NAPZBG;
};
sc_NAPZB1=({NAPWG1, STOP1}, NAPWG1, tf, of) {
```

```
    of(NAPWG1) = do / {V2};
    tf(NAPWG1, B1*!AU) => STOP1;
    tf(STOP1, !B1*!AU) => NAPWG1;
};
sc_NAPZB2=({NAPWG2, STOP2}, NAPWG2, tf, of) {
    of(NAPWG2)= do / {V4};
    tf(NAPWG2, B2*!AU) => STOP2;
    tf(STOP2, !B2*!AU) => NAPWG2;
};
sc_NAPELNIANIE=and(sc_NAPZBG:h, sc_NAPZB1:h, sc_NAPZB2:h);

// proces chemiczny
sc_REAKCJA=({WLEWANIE, OPROZNIANIE}, WLEWANIE, tf, of) {
    of(WLEWANIE) = do / {C1, C2, V3, V5};
    tf(WLEWANIE, FT1*!AU) => (OPROZNIANIE, {TM2});
    of(OPROZNIANIE) = do / {V6};
};
sc_PROCES=({REAKCJA, ZAKONCZENIE_PROCESU}, REAKCJA, tf, of) {
    of(REAKCJA) = do / {M};
    tf(REAKCJA, FT2*!FT1*!AU) => ZAKONCZENIE_PROCESU;
    of(ZAKONCZENIE_PROCESU) = do / {V6};
};
dec sc_PROCES by df {
    df(REAKCJA) = sc_REAKCJA;
    df(ZAKONCZENIE_PROCESU) = nodec;
};

// inicjowanie procesu chemicznego
sc_OPZBGL=({OPZBGL}, OPZBGL, tf, of) {
    of(OPZBGL) = do / {EV};
    tf(OPZBGL, Nmin) => endst;
};
sc_OPSK=({OPSK}, OPSK, tf, of) {
    of(OPSK) = do / {AC1, AC2};
    tf(OPSK, FT1) => endst;
};
sc_INICJOWANIE=and(sc_OPZBGL, sc_OPSK);

// reaktor - proces główny
sc_REAKTOR = ({START, RESTART, NAPELNIANIE, PROCES,
    INICJOWANIE}, START, tf, of) {
    tf(START, REP*!AU) => (INICJOWANIE, {TM1});
    tf(INICJOWANIE, AUT*!AU) => NAPELNIANIE;
    tf(NAPELNIANIE, AU) => RESTART;
    tf(RESTART, REP*!AU) => NAPELNIANIE;
    tf(NAPELNIANIE, Nmax*B1*B2*!AU) => (PROCES, {TM1});
    tf(PROCES, AUT*Nmin*!FT1*!FT2*!AU) => NAPELNIANIE;
    tf(PROCES, AU) => START;
};
```

```
dec sc_REAKTOR by df {
    df(START) = nodec;
    df(INICJOWANIE) = sc_INICJOWANIE;
    df(NAPELNIANIE) = sc_NAPELNIANIE;
    df(RESTART) = nodec;
    df(PROCES) = sc_PROCES;
};
```

BIBLIOGRAFIA

Adamski M. (1990): *Projektowanie układów cyfrowych systematyczną metodą strukturalną.* — Seria Monografie, nr 49, Wydawnictwo Wyższej Szkoły Inżynierskiej w Zielonej Górze.

Adamski M. (1991): *Parallel Controller Implementation using Standard PLD Software.* — [In:] W. Moore i W. Luk, ed.ed., FPGAs, Abingdon EE&CS Books, pp. 296–304.

Adamski M. (1998): *SFC, Petri Nets and Application Specific Logic Controllers.* — [In:] Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics, USA, San Diego, pp. 728–733.

Aho A. V., Sethi R. i Ullman J. D. (1990): *Compilers Principles, Techniques, and Tools.* — Addison-Wesley Publishing Company.

Andrzejewski G. (2001): *Program model of petri net.* — [In:] Proc. of the 4th Int. Conf. on Computer-Aided Design of Discrete Devices – CAD DD'2001, tom 1, Mińsk, Białoruś, pp. 87–92.

Andrzejewski G. (2002*a*): *Programowy model interpretowanej sieci Petriego dla potrzeb projektowania mikrosystemów cyfrowych.* — Rozprawa doktorska, Politechnika Szczecińska, Wydział Informatyki, Szczecin.

Andrzejewski G. (2002*b*): *Synchronizacja procesów sterujących zdekomponowanych w hybrydowych stukturach mikrosystemów cyfrowych.* — [In:] Mat. V Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'02, Szczecin, pp. 137–143.

Andrzejewski G. i Łabiak G. (1999): *Eksperymentalny system do syntezy kontrolerów współbieżnych.* — [In:] Mat. II Krajowej Konferencji Naukowej Reprogramowalne Uklady Cyfrowe – RUC'99, Szczecin, pp. 43–50.

Årzén K.-E. (1996): *Grafcet: a graphical language for sequential supervisory control application.* — [In:] Proc. ot the IFAC World Congress, San Francisco.

Banaszak Z., Kuś J. i Adamski M. (1993): *Sieci Petriego. Modelowanie, sterowanie i synteza systemów dyskretnych.* — Wydawnictwo Wyższej Szkoły Inżynierskiej, Zielona Góra.

Bazydło G. (2001): *Edytor graficzny diagramów statecharts.* — Praca inżynierska, Politechnika Zielonogórska, Wydział Elektryczny, Zielona Góra.

Berry G. (1993): *Preemtion in concurrent systems.* — [In:] Proc. of Annual Conference on Foundations of Software Technology and Theoretical Computer Science – FSTTCS'93, tom 761, LNCS, Springer, pp. 72–93.

Berry G. i Gonthier G. (1992): *The ESTEREL synchronous programming language: design, semantics, implementation.* — Science of Computer Programming **19**, 87–152.

Bet (2004): *WindRiver – BetterState.* — WWW. *http://www.windriver.com/products/betterstate/index.html

Biliński K. (1996): *Application of Petri Nets in parallel controllers design.* — Rozprawa doktorska, University of Bristol, Electrical and Electronic Engineering Department, Bristol.

Booch G., Rumbaugh J. i Jacobson I. (2001): *UML przewodnik użytkownika.* — Wydawnictwa Naukowo–Techniczne, Warszawa.

Buchenrieder K., Pyttel A. i Veith C. (1996): *Mapping statechart models onto an FPGA-based ASIP architecture.* — [In:] Proc. of European Design Automation Conference –EURO-DAC'96, pp. 184–189.

Burch J. R., Clarke E. M., McMillan K. L., i Dill D. (1990): *Sequential Circuit Verification Using Symbolic Model Checking.* — [In:] Proc. of the 27th Design Automation Conference – DAC'90, pp. 46–51.

Buttazzo G. (2001): *Artificial Consciousness: Utopia or Real Possibility?.* — COMPUTER **34**(7), 24–30.

Classen A. (1993): *Modulare Statecharts: Ein formaler Rahmen zur hierarchischen Processspezifikation.* — Praca magisterska, Lehrstuhl fur Informatik II, Aachen University of Technology, Germany.

Cortadella J., Yakovlev A. i Rozenberg G. (2002): *Concurrency and Hardware Design.* — [In:] G. Rozenberg, ed., Advances in Petri Nets, tom 2549, LNCS, Springer–Verlag.

COS (2004): *Project COSMA.* — WWW. *http://www.ii.pw.edu.pl/cosma/

Coudert O., Berthet C., i Madre J. C. (1989): *Verification of Sequential Machines Using Boolean Functional Vectors.* — [In:] Proc. of IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, pp. 111–128.

Daszczuk W. B., Grabski W., Mieścicki J. i Wytrębowicz J. (2001): *System Modeling in the COSMA Environment.* — [In:] Proc. of Euromicro Symposium on Digital Systems Design, pp. 152–157.

Day N. (1993): *A Model Checker for Statecharts (Linking CASE tools with Formal Methods).* — Technical report, University of British Columbia, Vancouver, Canada.

de Micheli G. (1998): *Synteza i optymalizacja układów cyfrowych.* — Wydawnictwa Naukowo-Techniczne, Warszawa.

Drusinsky D. (1997): *BetterState Pro Tutorial: An Introduction to Design with StateCharts.* — Integrated Systems, Inc., 201 Moffett Park Drive Sunnyvale, California 94089.

Drusinsky D. i Harel D. (1989*a*): *Electronic Controller Based on the Use of Statecharts as an Abstract Model.* — USA Patent nr 4 799 141. przyznany w styczniu 1989.

Drusinsky D. i Harel D. (1989*b*): *Using Statecharts for Hardware Description and Synthesis.* — IEEE Transaction on Coputer-Aided Design **8**(7), 798–807.

Drusinsky-Yoresh D. (1991): *A State Assignment Procedure for Single-Block Implementation of State Chart.* — IEEE Transaction on Coputer-Aided Design **10**(12), 1569–1576.

Esser R. (1996): *Application of Petri Nets in parallel controllers design.* — Rozprawa doktorska, Swiss Federal Institute of Technology, Zurich, Szwajcaria.

Fernandes J. M., Adamski M. i Proenca A. J. (1997): *VHDL Generation from Hierarchical Petri Net Specifications of Parallel Controllers.* — IEE Proceedings-E: Computers and Digital Techniques **2**(144), 127–137.

Gajski D. D., Vahid F. i Narayan S. (1993): *SpecCharts: A VHDL Front-End for Embedded Systems.* — Technical Report ICS-TR-93-31, UC Irvine.

Gajski D. D., Vahid F., Narayan S. i Gong J. (1994): *Specification and Design of Embedded Systems.* — Prentice Hall, Englewood Cliffs, New Jersey.

Gajski D. D., Zhu J., Döner R., Gerstlauer A. i Zhao S. (2001): *SpecC: Specification Language and Methodology.* — Kluwer Academic Publishers, Norwell, Massechusetts.

Ghosh A., Devadas S. i Newton A. R. (1992): *Sequential logic testing and verification.* — Kluwer Academic Publishers, Boston.

Girault C. i Valk R. (2003): *Petri Nets for Systems Engineering. A Guid to Medeling, Verification, and Application.* — Springer-Verlag, Berlin Heidelberg.

Harel D. (1987): *Statecharts: A Visual Formalism for Complex Systems.* — Science of Computer Programming **8**, 231–274.

Harel D. (2001): *Rzecz o istocie informatyki. Algorytmika.* — Klasyka Informatyki, 3 wydanie, Wydawnictwa Naukowo–Techniczne, Warszawa.

Harel D. i Naamad A. (1996): *The STATEMATE Semantics of Statecharts.* — ACM Trans. Soft. Eng. Method **5**(4).

Harel D. i Politi M. (1998): *Modeling Reactive Systems With Statecharts: The Statemate Approach.* — McGraw Hill.

Holvoet T. i Verbaeten P. (1995): *Petri Charts: an Alternative Technique For Hierarchical Net Construction.* — [In:] Proc. of IEEE Conference on Systems, Man and Cybernetics.

Hong J. E. i Bae D. H. (1998): *HOONets: Hierarchical Object-Oriented Petri Nets for System Modeling and Analysis.* — Technical Report CS/TR-98-132, KAIST.

Hopcroft J. E. i Ullman J. D. (2003): *Wprowadzenie do teorii automatów, języków i obliczeń.* — Wydawnictwo Naukowe PWN, Warszawa.

Huizing C. i Gerth R. (1992): *Semantics of Reactive Systems in Abstract Time.* — tom 600, LNCS, Springer, pp. 291–314.

I-L (2000*a*): *Rhapsody Reference Guide.*

I-L (2000*b*): *STATEMATE HDL Code Generator Reference Manual.*

I-L (2001): *STATEMATE Magnum Code Generation Guide.*

IAR (1999): *IAR visualSTATE Concept Guide.*

IAR (2004): *IAR visualSTATE.* — WWW. IAR Systems. *http://www.iar.com/Products/VS/

Jensen K. (1997): *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* — Monographs in Theoretical Computer Science, Springer-Verlag.

Kalisz J. (2002): *Podstawy elektroniki cyfrowej.* — Wydawnictwa Komunikacji i łączności, Warszawa.

Kamionka-Mikuła H., Małysiak H. i Pochopień B. (2004): *Układy cyfrowe – teoria i przykłady.* — 6 wydanie, Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice.

Karatkevich A. i Andrzejewski G. (2002): *Analiza wybranych własności interpretowanej sieci Petriego metodą optymalnej symulacji.* — [In:] Mat. I Krajowej Konferencji Elektroniki – KKE'02, tom 2, Kołobrzeg-Dźwirzyno, pp. 685–690.

Kesten Y. i Pnueli A. (1992): *Timed and Hybrid Statecharts and their Textual Representation.* — tom 571, LNCS, Springer, pp. 591–620.

Kozłowski T. (1993): *Petri-Net-Based CAD Tools for Parallel Controller Synthesis.* — Praca magisterska, University of Bristol, Electrical and Electronic Engineering Department, Bristol.

Kozłowski T., Dagless E. L., Saul J. M., Adamski M. i Szajna J. (1995): *Parallel controller synthesis using Petri nets.* — IEE Proceedings-E, Computers and Digital Techniquess **142**(4), 263–267.

Kyeyune Y. (2000): *Developing Concepts and Methods for Module and Integration Test for Models of Reactive Systems.* — Rozprawa doktorska, Fachbereich Informatik am Universität Dortmund, Dortmund.

Łabiak G. (1998): *Implementacja siei statechart w reprogramowalnej strukturze FPGA.* — [In:] Mat. I Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'98, Szczecin, pp. 169–176.

Łabiak G. (1999): *Modelling Statecharts Diagram by Means of Petri Nets.* — [In:] Proc. of the VI Interantional Conference on Advanced Computer Systems – ACS'99, Szczecin, pp. 253–259.

Łabiak G. (2000*a*): *Statecharts Specification Format (SSF) – tekstowa postać opisu diagramów.* — [In:] Mat. III Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'00, Szczecin, pp. 147–154.

Łabiak G. (2000*b*): *Zastosowanie hierarchicznego modelu współbieżnego automatu w projektowaniu układów cyfrowych.* — [In:] Mat. Konf. II Ogólnopolskich Warsztatów Doktoranckich – OWD'00, Istebna-Zaolzie, pp. 152–157.

Łabiak G. (2001*a*): *Application of BDDs in FPGA Synthesis of Statechart-based Controllers.* — [In:] Proc. ot the International Workshop Control and Information Technology, IWCIT'01, Ostrawa, Czechy, pp. 45–60.

Łabiak G. (2001*b*): *Statecharts Diagram as Linked Hierarchical Sequential Automata.* — [In:] Proc. of the 4th Int. Conf. on Computer-Aided Design of Discrete Devices – CAD DD'2001, tom 1, Mińsk, Białoruś, pp. 209–214.

Łabiak G. (2001*c*): *Symbolic States Exploration of Controllers Specified by Means of Statecharts.* — [In:] Proc. of the Intl. Workshop on Discrete-Event System Design – DESDes'01, Przytok, pp. 209–214.

Łabiak G. (2001*d*): *Zastosowanie Diagramów Statecharts w Specyfikacji Funkcjonalnej Układów Sterowania Binarnego.* — [In:] Mat. IV Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'01, Szczecin, pp. 55–60.

Łabiak G. (2002*a*): *HiCoS - akademicki system do projektowania hierarchicznych współbieżnych cyfrowych ukladów sterowania.* — [In:] Mat. I Krajowej Konferencji Elektroniki – KKE'02, pp. 397–402.

Łabiak G. (2002*b*): *Koncepcja syntezy diagramow Statechart w cyfrowych ukladach reprogramowalnych.* — [In:] Mat. V Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'02, Szczecin, pp. 113–120.

Łabiak G. (2003): *From UML statecharts to FPGA - the HiCoS approach.* — [In:] Proc. of Forum on Specification & Design Languages – FDL'03, Frankfurt am Main, pp. 354–363.

Łabiak G. i Andrzejewski G. (1999): *Edytor graficzny sieci Petriego.* — [In:] Mat. II Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'99, Szczecin, pp. 57–64.

Łabiak G. i Ludwicki M. (2000): *Wizualizacja tekstowej reprezentacji diagramów Statecharts.* — [In:] Mat. Konf. 21. Międzynarodowego Sympozjum Naukowego Studentów i Młodych Pracowników Nauki – MSN'21, tom Informatyka, Zielona Góra, pp. 130–137.

Lavagno L., Sangiovanni-Vincentelli A. i Sentovich E. (1998): *Models of Computation for Embedded System Design.* — [In:] System-Level Synthesis, tom 357, NATO Science, Kluwer Academic Publishers, Dordrecht/Boston/London, pp. 45–102.

Licht T. i Fengler W. (2003): *Timing analysis of UML-RT diagrams using timed automata.* — [In:] M. ColnariČ, M. Adamski i M. Węgrzyn, ed.ed., REAL-TIME PRGORAMMING 2003 (WRTP 2003), ELSEVIER LTD, Łagów, Poland, pp. 105–110. A proceedings volume from 26th IFAC/IFIP/IEEE Workshop Łagów, Poland, 14 – 17 May 2003.

Lu S., Halang W. A. i Pöschmann A. (2003): *Extending UML with PEARL features for the design of embedded real-time systems.* — [In:] M. ColnariČ, M. Adamski i M. Węgrzyn, ed.ed., REAL-TIME PRGORAMMING 2003 (WRTP 2003), ELSEVIER LTD, Łagów, Poland, pp. 75–80. A proceedings volume from 26th IFAC/IFIP/IEEE Workshop Łagów, Poland, 14 – 17 May 2003.

Łuba T. (2001): *Synteza układów logicznych.* — 2 wydanie, Wyższa Szkoła Inforamtyki Stosowanej i Zarządzania, Warszawa.

Łuba T. (2002): *Rola i znaczenie syntezy logicznej w technice cyfrowej układów programowalnych.* — [In:] Mat. I Krajowej Konferencji Elektroniki – KKE'02, Kołobrzeg–Dźwirzyno, pp. 35–42. Invited paper.

Łuba T. i Zbierzchowski B. (2000): *Komputerowe projektowanie układów cyfrowych.* — Wydawnictwa Komunikacji Łączności, Warszawa.

Maggiolo-Schettini A. i Merro M. (1997): *Priorities in Statecharts.* — tom 1192, LNCS, Springer–Verlag, pp. 404–429.

Maggiolo-Schettini A. i Peron A. (1993): *Semantics of Full Statecharts Based on Graph Rewriting.* — Italy.

Majewski W. (1998): *Układy Logiczne. Wybrane zagadnienia.* — Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa.

Majewski W., Łuba T., Jasiński K. i Zbierzchowski B. (1992): *Programowalne moduły logiczne w syntezie układów cyfrowych.* — Wydawnictwa Komunikacji i Łączności, Warszawa.

Maraninchi F. (1991): *Argos: a Graphical Synchronous Lanaguage for the Description of Reactive Systems.* — Grenoble, France.

Maraninchi F. (1992): *Operational and Compositional Semantics of Synchronous Automaton Composition.* — tom 630, LNCS, Springer, pp. 550–564.

Miczulski P. (2002*a*): *Algorytm konstruowania hierarchicznego grafu znakowań z wykorzystaniem przekształceń na funkcjach logicznych.* — [In:] Mat. V Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'02, Szczecin, pp. 95–100.

Miczulski P. (2002*b*): *Analiza diagramów decyzyjnych pod kątem reprezentowania hierarchicznej przestrzeni stanów współbieżnych kontrolerów cyfrowych.* — [In:] Mat. I Krajowej Konferencji Elektroniki – KKE'02, Kołobrzeg–Dźwirzyno, pp. 385–390.

Minato S.-I. (1996): *Binary Decision Diagrams and Applications for VLSI CAD.* — Kluwer Academic Publishers, Boston.

Mirkowski J. i Skowroński Z. (1998): *Translation of C and VHDL Specifications into Interpreted Petri Nets for Hardware/Software Codesign.* — [In:] A. Napieralski, ed., Mixed Design of Integrated Circuits and Systems, Kluwer Academic Publishers, Dordrecht.

Misiurewicz P. (1987): *Układy automatyki cyfrowej.* — Wydawnictwa Szkolne i Pedagogiczne, Warszawa.

Mrozek Z. (2001): *UML as integration tool for design of the mechatronic system.* — [In:] Proc. of the Second Interational Workshop on Robot Motion and Control, Bukowy Dworek, pp. 189–194.

Murata T. (1989): *Petri Nets: Properties, Analysis and Application.* — Proc. of the IEEE **77**(4), 263–267.

Nazareth D., Regensburger F. i Sholz P. (1996): *Mini-Statecharts, A Lean Version of Statecharts.* — Technical Report TUM–I9610, Technische Universitat Munchen.

Pasierbiński J. i Zbysiński P. (2001): *Układy programowalne w praktyce.* — Wydawnictwa Komunikacji i Łączności, Warszawa.

Pastor E., Roig O., Cortadella J. i Badia R. (1994): *Petri Net Analysis Using Boolean Manipulation.* — tom 815, LNCS, Springer–Verlag, pp. 416–435.

Peron A. (1993): *Synchronous and Asynchronous Models for Statecharts.* — Technical Report TD-29/93, Dipartimento di Informatica, Universatita di Pisa, Italy.

Perry D. L. (1993): *VHDL.* — Computer Engineering, McGraw-Hill, Eglewood Cliffs, New Jersey.

Peterson J. L. (1981): *Petri net theory and the modeling of systems.* — Prentice-Hall, Eglewood Cliffs, New Jersey.

Pnueli A. i Shalev M. (1991): *What is in a step: On the semantics of Statecharts.* — tom 526, LNCS, Springer, pp. 244–264.

Puczyńska M., Łabiak G. i Wolański P. (2000): *Programowa implementacja konwersji sieci Petriego na język VHDL.* — [In:] Mat. III Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'00, Szczecin, pp. 285–291.

Ramesh S. (1999): *Efficient Translation of Statecharts to Hardware Circuits.* — [In:] Proc. of Twelfth International Conference On VLSI Design, pp. 384–389.

Rasiowa H. (1998): *Wstęp do matematyki współczesnej.* — Wydawnictwo Naukowe PWN, Warszawa.

Rat (2000): *Using Rose.* PART NUMBER: 800-024462-000.

Rat (2004): *Rational software corporation.* — WWW. *http://www.rational.com

Rausch M. i Krogh B. H. (1998): *Symbolic Verification of Stateflow Logic.* — [In:] Proc. of the 4th Workshop on Discrete Event System, Cagliari, Italy, pp. 489–494.

Rea (2001): *Realtimes – i-logix newsletter.* — Internet edtion. Issue 2, Volume 2.

Reisig W. (1988): *Sieci Petriego. Wprowadzenie.* — Wydawnictwa Naukowo-Techniczne, Warszawa.

Rha (2004): *I-Logix – Rhapsody.* — WWW. *http://www.ilogix.com/products/rhapsody/index.cfm

Rushton A. (1998): *VHDL for Logic Synthesis.* — John Wiley and Sons Ltd, Chichester.

Skahill K. (2001): *Język VHDL Projektowanie programowalnych układów logicznych.* — Wydawnictwa Naukowo–Techniczne, Warszawa.

Skowroński Z. (1999): *Translacja specyfikacji funkcjonalnej układów cyfrowych na sieć petirego dla potrzeb syntezy wysokiego poziomu.* — [In:] Mat. Konf. I Ogólnopolskich Warsztatów Doktoranckich – OWD'99, Istebna-Zaolzie, pp. 185–190.

Skowroński Z. (2000): *Translacja specyfikacji funkcjonalnej układów cyfrowych na sieć Petirego dla potrzeb syntezy systemowej.* — Rozprawa doktorska, Politechnika Szczecińska, Wydział Informatyki, Szczecin.

Somenzi F. (2004): *CUDD: CU Decision Diagram Package Release 2.4.0.* — WWW. Department of Electrical and Computer Engineering University of Colorado at Boulder.
*http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html

STA (2004): *I-Logix – Statemate.* — WWW.
*http://www.ilogix.com/products/magnum/index.cfm

Stroustrup B. (2002): *Język C++.* — Klasyka Informatyki, 6 wydanie, Wydawnictwa Naukowo–Techniczne, Warszawa.

Subieta K. (1999*a*): *Język UML.* — [In:] Mat. V Konferencji PLOUG, Zakopane, pp. 235–252.

Subieta K. (1999*b*): *Słownik terminów z zakresu obiektowości.* — Akademicka Oficyna Wydawnicza PLJ, Warszawa.

Subieta K. (1999*c*): Wprowadzenie do obiektowych metodyk projektowania i notacji UML. Jedenasta Górska Szkoła PTI.

Toth V. (1997): *Visual C++.* — UNLEASHED, Sams Publishing, Indianapolis.

Traczyk W. (1986): *Układy cyforwe. Podstawy teoretyczne i metody syntezy.* — Wydawnictwa Naukowo-Techniczne, Warszawa.

UML (2003): *OMG Unified Modeling Languag Specification Version 1.5.*
*http://www.omg.org/technology/documents/formal/uml.htm

Vojnar T. (1999): *Hierarchical and Time Extensions of Pure Object Oriented Petri Nets.* — [In:] Proc. of ASIS'97, Ostrawa.

von der Beeck M. (1994): *A Comparison of Statecharts Variants.* — [In:] Formal Techniques in real-Time and Fault-Tolerant Systems, Third International Symposium, LNCS, Springer-Verlag, pp. 128–148.

Węgrzyn A. (2001): *Symbolic Verification of Concurrent Logic Controllers by Means of Petri Nets.* — [In:] Proc. ot the International Workshop Control and Information Technology, IWCIT'01, Ostrawa, Czechy, pp. 198–203.

Węgrzyn A. i Węgrzyn M. (2000): *Petri Net-Based Specification, Analysis and Synthesis of Logic Controllers.* — [In:] Proc. of the IEEE International Symposium on Industrial Electronics – ISIE'00, tom 1, Udla, Meksyk, pp. 20–26.

Węgrzyn M. (1998*a*): *Hierarchiczna implementacja współbieżnych kontrolerów cyfrowych z wykorzystaniem FPGA.* — Rozprawa doktorska, Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych, Warszawa.

Węgrzyn M. (1998*b*): *Hierarchiczna synteza kontrolerów logicznych z wykorzystaniem FPGA.* — [In:] Mat. I Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'98, Szczecin, pp. 33–40.

Węgrzyn M. i Adamski M. (1999): *Hierarchical Approach for Design of Application Specific Logic Controller.* — [In:] Proc. of the IEEE International Symposium on Industrial Electronics – ISIE'99, tom 3, Bled, Słowenia, pp. 1389–1394.

Wolański P. (1998*a*): *Modelowanie układów cyfrowych na poziomie RTL z wykorzystaniem sieci Petriego i podzbioru języka VHDL.* — Rozprawa doktorska, Politechnika Warszawska, Wydział Elektroniki i Technik Informacyjnych, Warszawa.

Wolański P. (1998*b*): *Modelowanie współbieżnych układów cyfrowych z wykorzystaniem syntezowalnego podzbioru języka VHDL.* — [In:] Mat. I Krajowej Konferencji Naukowej Reprogramowalne Układy Cyfrowe – RUC'98, Szczecin, pp. 91–98.

Zwoliński M. (2002): *Projektowanie układów cyfrowych z wykorzystaniem języka VHDL.* — Wydawnictwa Komunikacji i Łączności, Warszawa.

SPIS RYSUNKÓW

SPIS TABEL

SPIS KODÓW ŹRÓDŁOWYCH

The use of hierarchical model of concurrent automaton in digital controller design

Abstract

The permanent and exponential growth of capabilities of programmable devices, which is not accompanied by a similar increase in design productivity, and has caused a kind of productivity gap. Electronic digital circuits, designed with traditional methods, have almost reached their upper complexity limits. Further improvements in digital circuitry functionality can be gained through development related to high level modelling according to the rule: the better model is, the better final device can be produced from it.

The main goal of the book is to present a new high level modelling method, which can be applied to the design of digital binary controllers whose behaviour may be specified with the statechart diagrams and consequently, the controllers are directly implemented in programmable logic devices.

Statechart diagrams Statechart diagrams are considered hierarchical concurrent finite state machines. The language was developed as a visual formalism for complex systems by David Harel. It is a state-based graphical notation which can be perceived as an extension of the state transition graph of the traditional finite state machine. In comparison with *FSM* statecharts are enhanced with concurrency, hierarchy and broadcasting mechanism. At present, statecharts are mainly used in the *UML* technology, where they are exploited in behaviour modelling of program objects (in the sense of *C++* or *Java* languages). Recently, *UML* is gaining in popularity as a tool for hardware design and *HiCoS* (***Hi****erarchical* ***Co****ncurrent* ***S****ystem*) system is the author's proposition of a *CAD* program, which directly transforms behaviour specified with statecharts into Register Transfer Level *VHDL*.

Syntax of statecharts The big problem about statecharts is syntax and semantics. A variety of applications domain caused that many authors proposed their own syntax and semantics, which sometimes differ significantly. Syntax and semantics presented in this paper are intended for specifying the behaviour of binary digital controllers which would comply with the *UML* standard as much as possible. Not every element of *UML* statechart syntax is supported in *HiCoS* approach. The selection of language characteristics were based on application domain and the technical constrains of programmable logic devices. As a result of those considerations, it was assumed that the syntax of the author's system, *HiCoS*, is to be intended for untimed control systems which operate on binary values. Moreover,

the research was divided into two stages, delimited with a modular paradigm. Hence, *HiCoS* statecharts are characterized by hierarchy and concurrency, simple state, composite state, end state, discrete events, actions assigned to states (*entry*, *do*, *exit*), simple transitions, history attribute and logic predicates imposed on transitions. Another essential issue is to allow the use of feedbacks, it means that events generated in a circuit can affect its behaviour. The role of an end state is to prevent an activity from taking away a sequential automaton before the end state becomes active. Such elements as factored transition paths and time were rejected, whereas others like cross-level and composite transitions as well as synch states were moved to the future stage of the research.

Semantics of statecharts A digital controller specified with a Statechart and realized as an electronic circuit is meant to work in an environment which prompts the controller by means of events. It is assumed that every event (incoming, outgoing and internal) is bound with a discrete time domain. The controller reacts to the set of accessible events in the system through firing a set of enabled transitions called a microstep. Because of feedback, execution of a microstep entails generating further events and causes firing subsequent microsteps. Events triggered during a current microstep do not influence transitions that are being realized, but they are only allowed to affect behaviour of a controller in the next tick of discrete time, that is, in the next microstep. A sequence of subsequently generated microsteps is called a step and additionally it is assumed that during a step no events can come from the outside world. A step is said to be finished when there are no enabled transitions. Summarizing, dynamic characteristics of hardware implementation are as follows:

- system is synchronous,
- system reacts to the set of available events through transition executions,
- generated events are accessible to the system during the next tick of the clock.

Hardware mapping The main assumption of a hardware implementation behaviour described with statechart diagrams is that the systems specified in this way can directly be mapped into the programmable logic devices. This means that elements from a diagram (for example states or events) are to be in direct correspondence with resources available in a programmable device – mainly flip-flops and programmable combinatorial logic. Basing on that assumption and taking into account the assumed dynamic characteristics, the following principles of hardware implementation have been formulated:

- each state is assigned one flip-flop – its activity means that the state associated with the flip-flop can be active or, in the case of a state with a history attribute, its past activity is being remembered; the activity of a state is established on the basis of activity of flip-flops assigned to superordinate states (in the sense of hierarchy tree),

- each event is also assigned one flip-flop – its activity means the occurrence of an associated event and is sustained to the next tick of discrete time when the event becomes available to the system,
- excitation functions are created for each flip-flop in a circuit, based on the diagram topography and rules of transition executions,

The HiCoS system Up till now there have not been many *CAD* programs which employ statechart diagrams in digital circuit design implemented in programmable devices. The most prominent is *Statemate MAGNUM* by *I-Logix*, where the modelled behaviour is ultimately described in *HDL* language (*VHDL* or *Verilog*) with the use of case and sequential instructions like *process* or *always*. The *HiCoS* system automatically converts the behaviour described with statecharts into a register transfer level. The transformation is realized as one-hot mapping, which means that one state corresponds to one flip-flop. The system implemented in programmable device reacts to the set of incoming events through transitions executions. The input model is described in its own textual representation called *SSF* (***S**tatechart **S**pecification **F**ormat*) which is equivalent to graphical form, and is later transformed into Boolean equations. The Boolean equations are internally represented by means of *BDD*s. Next, a reachability graph can be built or *RTL-VHDL*-designed model can be implemented in programmable logic devices. As opposed to the commercial solution, which transforms statechart diagrams into behavioural model of *VHDL* language, the author's proposition yields better to formal analysis. This advantage allows sophisticated optimization and model-checking algorithms to be implemented. In the dissertation, the algorithm that generates reachability graph is used as an introductory example. In this case, the symbolic state space is represented with a characteristic function and *ROBDD*.

Within the scope of the conducted research, a subset of statecharts language has been established which conforms to the *UML* standard and which is the most suitable for modelling binary controller behaviour. Many syntactic and semantic issues have been analyzed with reference to classic state-of-the-art papers. The proposed mathematical model in the thesis that formally defines syntax and semantics, forms a basis for description of behaviour with Boolean equations. The results of theoretical considerations are the foundation for the author's *CAD* program, called *HiCoS*, which can be downloaded from the web page: *www.uz.zgora.pl/~glabiak*.

Practical results show that the developed *HiCoS* system can be a valuable tool for the modern designer and allows the behaviour of modelled systems to be far more complex than it is in the case of traditional methods like *FSM* or Petri nets.

Prace Naukowe z Automatyki i Informatyki

Przewodniczący: Józef KORBICZ

Tom 6: Grzegorz Łabiak
Wykorzystanie hierarchicznego modelu współbieżnego automatu w projektowaniu sterowników cyfrowych
168 s. 2005 [83-89712-42-3]

Tom 5: Maciej Patan
Optimal Observation Strategies for Parameter Estimation of Distributed Systems
220 s. 2004 [83-89712-03-2]

Tom 4: Przemysław Jacewicz
Model Analysis and Synthesis of Complex Physical Systems Using Cellular Automata
134 s. 2003 [83-89321-67-X]

Tom 3: Agnieszka Węgrzyn
Symboliczna analiza układów sterowania binarnego z wykorzystaniem wybranych metod analizy sieci Petriego
125 s. 2003 [83-89321-54-8]

Tom 2: Grzegorz Andrzejewski
Programowy model interpretowanej sieci Petriego dla potrzeb projektowania mikrosystemów cyfrowych
109 s. 2003 [83-89321-53-X]

Tom 1: Marcin Witczak
Identification and Fault Detection of Non-Linear Dynamic Systems
124 s. 2003 [83-88317-65-2]