SPIS TREŚCI

Autor pragnie bardzo serdecznie podziękować wszystkim,
którzy okazali pomoc w trakcie tworzenia pracy,
a w szczególności Panu Profesorowi Marianowi Adamskiemu
za poświęcony czas, wsparcie i cenne uwagi merytoryczne.

mojej żonie

Wykaz najważniejszych symboli i oznaczeń

x	element zbioru, litera
X	zbiór elementów, alfabet
$\mathbf{x}$	wektor utworzony na alfabecie X
$\mathbf{X}$	zbiór wszystkich wektorów, przestrzeń
$\mathbb{T}$	dyskretna skala czasu
N	zbiór liczb naturalnych
$\forall$	kwantyfikator ogólny
$\exists$	kwantyfikator egzystencjalny
$\rightarrow$	operator odwzorowania
$\Rightarrow$	operator implikacji
$\in$	operator przynależności do zbioru
$\notin$	operator wyłączenia ze zbioru
$\subset$	operator inkluzji
$\cup$	operator sumy zbiorów
$\cap$	operator iloczynu zbiorów
$\times$	operator iloczynu kartezjańskiego
$\varnothing$	zbiór pusty
P	zbiór miejsc sieci Petriego
T	zbiór tranzycji sieci Petriego
N	zbiór węzłów sieci $N=P\cup T$
F	zbiór łuków sieci Petriego
F_o	zbiór łuków zwykłych
F_e	zbiór łuków zezwalających
F_i	zbiór łuków zabraniających
$P_t^{in(o)}$	zbiór miejsc wejściowych tranzycji t
$P_t^{in(e)}$	zbiór miejsc zezwalających tranzycji
$P_t^{in(i)}$	zbiór miejsc zabraniających tranzycji
P_t^{out}	zbiór miejsc wyjściowych tranzycji t
P_p^{end}	zbiór miejsc końcowych podsieci skojarzonej z makromiejscem p
δ	funkcja przejść
κ	funkcja pojemności miejsc
λ	funkcja wyjść
ω	funkcja wagi łuków
$\alpha(p)$	funkcja znakowania początkowego
$\chi(p)$	funkcja hierarchii

$\varepsilon(p)$	funkcja znakowania końcowego
$\lambda(n)$	funkcja etykietująca
$\tau(n)$	funkcja czasu
$\psi(p)$	funkcja historii
ac(p)	funkcja aktywności miejsca
cond(t)	warunek słaby zezwolenia tranzycji
abort(t)	warunek wywłaszczający tranzycji
action(n)	akcja związana z węzłem n
la(n)	najbliższy przodek węzła n
S	zbiór sygnałów
H	operator historii
H*	operator głębokiej historii
Z	zbiór wszystkich podsieci
Z^k	podsieć indeksowana przez k

Rozwinięcia najważniejszych skrótów

ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
CAD	Computer Aided Design
CPLD	Complex Programmable Logic Device
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HPN	Hierarchical Petri Net
HW/SW	Hardware / Software Co-Design
IP	Intellectual Property
PHPN	Program Hierarchical Petri Net
RTOS	Real Time Operating System
SOC	System on Chip
UML	Unified Modelling Language
VDS	Virtual Decision System
VHDL	Very high speed integrated circuit Hardware Description Language

Wprowadzenie

Tematyka projektowania układów sterowania cyfrowego jest szeroko rozwijana już od wielu lat. Opracowano wiele metod realizacji praktycznej takich układów, wskazując jako platformy docelowe zarówno struktury sprzętowe, jak i systemy procesorowe implementacji programowej. Realizacja sprzętowa pozwala na uzyskanie bardzo dobrych parametrów czasowych, ale jest stosunkowo kosztowna zarówno na etapie projektowania jak i produkcji. Realizacja programowa jest uważana wciąż jeszcze jako tańsza, ale parametry czasowe są gorsze od poprzedniej. Alternatywą może być realizacja sprzętowo-programowa, która pozwala w rozsądny sposób łączyć zalety obu metod i niwelować częściowo ich wady. Dzięki temu można uzyskać rozwiązania cechujące się zadowalającymi parametrami przy stosunkowo niewielkich kosztach wytwarzania.

Wiele ośrodków akademickich zajmuje się problematyką zintegrowanego projektowania sprzętu i oprogramowania (ang. *Hardware/Software Co-Design*), tworząc komputerowe systemy wspomagające projektowanie. Do najpopularniejszych pakietów należą: POLIS, opracowany na uniwersytecie w Berkeley, COSYMA, LYCOS, czy COSMOS. Wejściem do takich systemów są najczęściej projekty opisane z wykorzystaniem języków opisu sprzętu (np. VHDL, VERILOG) lub zapisu tekstowego modelu specyfikacji formalnej, wykorzystywanego w danym pakiecie (np. CFSM, CDFG). Modele te albo nie wspierają współbieżności, albo współbieżność osiągana jest na drodze łączenia wielu niezależnych modułów. Alternatywą może być zastosowanie interpretowanych sieci Petriego, które przy dobrze wykształconym aparacie matematycznym – w naturalny sposób wspierają współbieżność.

Dotychczasowe rozwiązania praktyczne realizacji niejednorodnych bazują na systemach złożonych z osobnych modułów sprzętowych i mikroprocesorowych komunikujących się pomiędzy sobą specjalnymi interfejsami (na poziomie synchronizacji zadań). W ostatnim czasie zauważa się dynamiczny rozwój cyfrowych technologii hybrydowych, integrujących w strukturach monolitycznych zarówno część sprzętową (np. FPGA), jak i rdzeń mikroprocesora lub mikrokontrolera (np. Intel 8051). Układy takie, często doposażone w programowalny blok analogowy, noszą nazwę mikrosystemów cyfrowych. Implementacja w takich strukturach sterowników cyfrowych wnosi nową jakość w dotychczasowe rozwiązania praktyczne, szczególnie w dziedzinie rozwiązań sprzętowo-programowych. Pozwalają one bowiem na szereg uproszczeń w projekcie, dotyczących szczególnie części interfejsu łączącego moduły sprzętowe z programowymi, a co za tym idzie – na zwiększenie prędkości wykonywania działań związanych z obsługą układu sterowanego.

Głównym problemem badawczym, jaki został podjęty w pracy jest sformułowanie ogólnych, przydatnych w praktyce zasad syntezy programowej

interpretowanej sieci Petriego, dla potrzeb projektowania układów sterowania binarnego, z wykorzystaniem mikrosystemów cyfrowych. Rozwiązania programowe obarczone są stosunkowo dużymi czasami reakcji na zmiany warunków wejściowych, dlatego też ogranicza się zakres stosowalności przyjętych rozwiązań do systemów, w których czas odpowiedzi nie jest parametrem wyjątkowo krytycznym. Dodatkowym kryterium wyznaczającym kierunek i zakres prac jest ograniczona pojemność pamięci programu w mikrosystemach cyfrowych, wynosząca dla najnowszych struktur do 64kB. Rozwiązanie praktyczne powinno więc traktować ograniczenie zużycia pamięci jako warunek naczelny.

W pracy zdecydowano się na zintegrowanie najistotniejszych rozszerzeń interpretowanych sieci Petriego w jeden model – HPN, dodatkowo doposażony w znaczące elementy opisu popularnych modeli standardu UML (np. historia, wywłaszczenie). Dzięki temu uzyskano wydajne i zarazem elastyczne narzędzie formalnego opisu zachowania systemów reaktywnych.

Proponuje się wprowadzenie nowego, programowego modelu interpretowanych sieci Petriego, będącego teoriomnogościową reprezentacją modelu sieci hierarchicznych HPN w ujęciu klasycznym. Model ten zdefiniowany został w abstrakcyjnym systemie decyzyjnym, rozumianym jako pewien zespół funkcji programowych, który operując na informacjach zawartych w odpowiednich strukturach danych podejmuje decyzje o przeprowadzaniu sieci do stanów kolejnych. Jest to nowatorskie, niespotykane dotychczas dla sieci Petriego rozwiązanie, umożliwiające prostą i efektywną implementację (pod względem zajętości pamięci) układów sterowania cyfrowego w sposób programowy, z wykorzystaniem języków wysokiego poziomu.

Wyniki prac mogą wskazywać na dużą użyteczność proponowanej metody nie tylko dla projektowania mikrosystemów cyfrowych, ale również w całym obszarze projektowania zintegrowanego sprzętu i oprogramowania. Opracowany moduł syntezy programowej może stanowić bowiem wygodną nakładkę narzędziową dla pakietów wspomagających projektowanie, dla których przewidziano heterogeniczny interfejs wejściowy (patrz rozdz. 6).

Bezpośrednią przyczyną podjęcia tematyki badań była potrzeba opracowania narzędzia wspomagającego automatyczną syntezę programową sieci w tworzonym na Uniwersytecie Zielonogórskim pakiecie oprogramowania wspomagającym projektowanie zintegrowane sprzętu i oprogramowania (Andrzejewski i Łabiak 1999, Biliński 1996, Kozłowski 1996, Łabiak i Andrzejewski 1999, Skowroński 2000).

Praca podzielona została na sześć rozdziałów. Pierwszy rozdział zawiera definicje niezbędnych pojęć, przedstawiono charakterystyki najpopularniejszych modeli specyfikacji formalnej układów sterowania cyfrowego. Na tym tle zaproponowano oryginalne rozszerzenie sieci interpretowanych do sieci hierarchicznych HPN. Ukazano także nowe możliwości implementacji praktycznej systemów sterowania na różnych platformach realizacyjnych, ze szczególnym wskazaniem na nowoczesną grupę struktur hybrydowych, tzw. mikrosystemów cyfrowych, łączących w sobie m.in. reprogramowalną matrycę FPGA wraz z rdzeniem mikroprocesora bądź mikrokontrolera. Rozdział ten kończy opracowanie przeglądu narzędzi wspomagających projektowanie zintegrowane części sprzętowej i programowej, wraz

ze wskazaniem na potencjalne możliwości wykorzystania wyników pracy jako nakładki narzędziowej w większości omawianych pakietów.

Rozdział drugi zawiera przegląd istniejących metod programowej realizacji interpretowanych sieci Petriego oraz najistotniejsze (z punktu widzenia pracy) metody realizacji modeli innych od sieci Petriego.

Rozdział trzeci stanowi najistotniejszą część pracy, w której wprowadzono szczegółowy, programowy model teoretyczny interpretowanej, hierarchicznej sieci Petriego. Potraktowano go jako system dyskretny, opisany uporządkowaną piątką, ze ściśle zdefiniowaną interpretacją sygnałów wejściowych, wyjściowych oraz elementów, takich jak miejsce, oznakowanie miejsca, tranzycja, realizacja tranzycji, itp.

W kolejnym rozdziale przedstawiono ogólne zasady syntezy proponowanego modelu programowego w mikrosystemach cyfrowych. Wskazano standard języka ANSI C, jako jeden z najpopularniejszych języków wysokiego poziomu programowania mikroprocesorów i mikrokomputerów przemysłowych, podejmując próbę opisania w nim wzmiankowanego modelu. Rozdział czwarty zawiera także autorskie opracowanie algorytmu dekompozycji systemowej sieci Petriego, dla potrzeb zastosowania w strukturach hybrydowych sprzętowo-programowych, szczególnie w kontekście współpracy międzymodułowej wykorzystującej możliwości istniejących układów mikrosystemów cyfrowych. Rozdział ten opisuje również szczegóły proponowanej metody projektowania oraz przedstawia zarys sposobu implementacji programowej uprzednio wprowadzonego modelu teoretycznego.

Rozdział piąty zawiera wyniki testów przeprowadzonych dla porównania proponowanej metody realizacyjnej z wybranymi istniejącymi metodami, stosowanymi w pakietach zarówno akademickich (POLIS), jak i profesjonalnych (ESTEREL). Pokazano także wpływ topologii sieci na rozmiar kodu wynikowego, a także szybkość wykonywania cyklu decyzyjnego.

W rozdziale szóstym przedstawiono ogólną koncepcję nowego środowiska programowego, wspomagającego projektowanie układów sterowania w strukturach sprzętowo-programowych, precyzyjnie wskazując miejsce autorskiego modułu syntezy programowej wśród innych modułów systemu.

Rozdział siódmy stanowi podsumowanie pracy, wskazuje się w nim osiągnięte cele oraz możliwości dalszej kontynuacji prowadzenia badań.

W dodatkach do pracy zamieszczono gramatykę języka *hpn*, wprowadzonego jako tekstową, pośrednią reprezentację sieci hierarchicznych HPN, oraz zestawienie skryptów testowych, używanych podczas prowadzonych badań.

Praca została przygotowana na podstawie pracy doktorskiej autora i wydana częściowo w ramach dwóch projektów badawczych KBN nr 7 T11C 010 20 oraz 4 T11C 006 24.

Rozdział 1

PROJEKTOWANIE STEROWNIKÓW CYFROWYCH

W rozdziale niniejszym przedstawione zostaną najważniejsze zagadnienia związane z problematyką projektowania układów sterowania binarnego. W szczególności scharakteryzowano wybrane modele specyfikacji formalnej, stosowane obecnie metodologie projektowania oraz platformy realizacyjne.

1.1. Pojęcia podstawowe

Dla pełnego zrozumienia i lepszego usystematyzowania dalszej części pracy, niezbędne wydaje się wprowadzenie szeregu pojęć podstawowych. Definicje 1.1-1.6 mają charakter ogólny i można je odnaleźć w formie zbliżonej, w literaturze przedmiotu (Majewski 1999).

Istotą procesu sterowania jest odpowiednie kształtowanie sygnałów sterujących w odpowiedzi na zmiany sygnałów informacyjnych. Sygnały sterujące przyjęło się nazywać sygnałami wyjściowymi a informacyjne – wejściowymi. Dodatkowo można wyróżnić grupę pomocniczych sygnałów wewnętrznych, nazywanych dalej lokalnymi. Każdy sygnał może przyjmować wartości ze zbioru wartości dopuszczalnych. W niniejszej pracy ograniczono się do układów sterowania binarnego, w których dozwolone są jedynie dwie wartości logiczne, nazywane umownie 0 i 1.

Każdemu sygnałowi można przyporządkować określoną nazwę – literę, jednoznacznie go określającą, np.: x_1, y_1, itp.

Definicja 1.1 *Litera wejściowa x jest to nazwa przyporządkowana określonemu sygnałowi wejściowemu. Litera wyjściowa y jest to nazwa przyporządkowana określonemu sygnałowi wyjściowemu. Litera lokalna l jest to nazwa przyporządkowana określonemu sygnałowi lokalnemu.*

Zbiór wszystkich liter tworzy alfabet.

Definicja 1.2 *Alfabet wejściowy X jest to skończony, uporządkowany zbiór wszystkich liter wejściowych, np.* $X = \{x_1,x_2,...,x_n\}$. *Alfabet wyjściowy Y jest to skończony, uporządkowany zbiór wszystkich liter wyjściowych, np.* $Y = \{y_1,y_2,...,y_n\}$. *Alfabet lokalny L jest to skończony, uporządkowany zbiór wszystkich liter lokalnych, np.* $L = \{l_1,l_2,...,l_n\}$.

Dowolna kombinacja wartości wszystkich liter alfabetu tworzy wektor.

Definicja 1.3 *Wektor wejściowy* ***x*** *jest to dowolna kombinacja wartości liter alfabetu wejściowego X, np.* ***x*** *= <1,0,...,0>. Wektor wyjściowy* ***y*** *jest to dowolna kombinacja wartości liter alfabetu wyjściowego Y, np.* ***y*** *= <0,1,...,0>. Wektor lokalny* ***l*** *jest to dowolna kombinacja wartości liter alfabetu lokalnego L, np.* ***l*** *= <0,0,...,1>.*

Zbiór wszystkich możliwych wektorów utworzonych na danym alfabecie tworzy przestrzeń. Liczebność elementów przestrzeni określona jest poprzez liczebność alfabetu – $|X|=2^X$.

Definicja 1.4 *Przestrzeń wejścia* ***X*** *jest to zbiór wszystkich możliwych wektorów wejściowych. Przestrzeń wyjścia* ***Y*** *jest to zbiór wszystkich możliwych wektorów wyjściowych. Przestrzeń lokalna* ***L*** *jest to zbiór wszystkich możliwych wektorów lokalnych.*

Często zachodzi sytuacja, w której zmiana stanu układu sterowania uwarunkowana jest m.in. ustawieniem określonego wektora wejściowego, bądź dowolnego z określonego zbioru wektorów wejściowych.

Definicja 1.5 *Dowolny podzbiór przestrzeni wejścia* ***X****, warunkujący zmianę stanu układu sterowania nazywany jest warunkiem wejściowym. Dowolny podzbiór przestrzeni lokalnej* ***L****, warunkujący zmianę stanu układu sterowania nazywany jest warunkiem lokalnym. Złożenie warunku wejściowego i lokalnego tworzy warunek ogólny lub krócej warunek.*

Uwaga: Wygodnie jest podawać warunek w postaci formuły logicznej *cond*, opisanej na alfabetach wejściowym oraz lokalnym, z wykorzystaniem standardowych operatorów logicznych (*not*, *and*, *or*).

W przypadku opisywania cyfrowych systemów zależnych od czasu, wygodnie jest też posługiwać się pojęciem dyskretnej skali czasu.

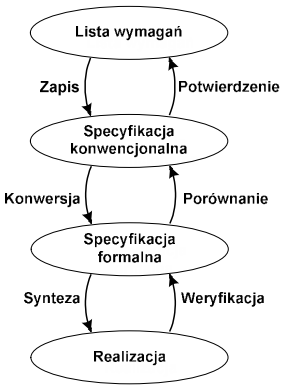
Definicja 1.6 *Dyskretną skalą czasu* $\mathcal{T}$ *nazywamy zbiór liczb przyporządkowanych dyskretnym wartościom czasu, uporządkowanych według relacji porządkującej.*

1.2. Wybrane modele specyfikacji formalnej

Projektowanie w sensie ogólnym można przedstawić jako szereg kolejnych czynności, mających na celu przejście od założeń do realizacji fizycznej. Na rys. 1.1 przedstawiono diagram ilustrujący proces projektowania (Hurk i Jess 1998).

W owalach oznaczono stan projektu (np. specyfikacja, realizacja), łuki natomiast określają akcje wykonywane na projekcie (np. konwersja, synteza, weryfikacja). Proces zaczyna się zwykle od ustalenia listy warunków i zadań, jakie ma spełniać projektowany układ. Na jej podstawie i po uzgodnieniach ze zleceniodawcą można przygotować opis działania. Jest to najczęściej tekstowo-graficzny zapis, pozbawiony formalizmów – specyfikacja konwencjonalna. Kolejnym etapem jest sporządzenie specyfikacji formalnej, uwzględniającej zasady leksykalne i semantyczne przyjętego

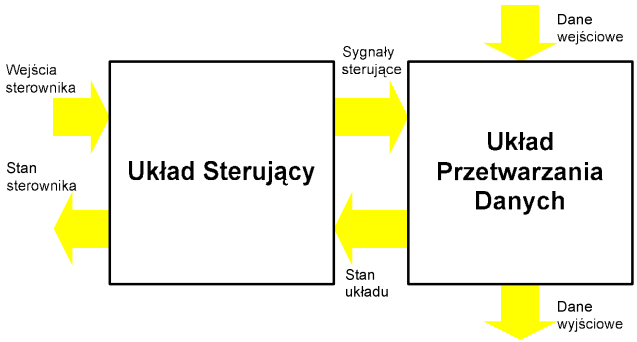
modelu opisu. Następnie na drodze syntezy i weryfikacji można wykonać produkt docelowy – realizację fizyczną.



Rys. 1.1. Metodyka projektowania

Pierwsze dwie fazy projektowania nie wymagają dodatkowych wyjaśnień, na uwagę zasługują natomiast fazy kolejne: specyfikacja formalna i implementacja.

Ogólnie przyjętą konwencją w projektowaniu systemów cyfrowych jest ich podział na dwie części (Edwards *i in.* 1997, Micheli 1998): jednostkę sterującą (ang. *Control Unit*) oraz układ przetwarzania danych (ang. *Data Path*), jak na rys. 1.2. Ma to istotne znaczenie nie tylko ze względu na większą przejrzystość samego projektu, ale głównie ze względu na odseparowanie danych od układu sterującego, co w sposób znaczący poprawia niezawodność i bezpieczeństwo pracy całego systemu.



Rys. 1.2. Podział systemu cyfrowego na części: sterującą i przetwarzania danych

Istnieją różne modele specyfikacji formalnej. Ze względu na swoje własności mogą znaleźć zastosowanie do opisu części sterującej lub przetwarzania danych. W pracy (Gajski 1997) przedstawiono klasyfikację modeli formalnych wskazującą możliwości ich stosowania:

- zorientowane na stan – do modelowania wykorzystuje się zbiory stanów i przejść pomiędzy nimi, najbardziej zalecany do opisu układów sterowania (np. FSM, CFSM, HCFSM, sieci Petriego);
- zorientowane na wykonywaną czynność – operują na zbiorach czynności związanych z przetwarzaniem danych (np. DFG);
- zorientowane na strukturę – zbiory fizycznych modułów wraz z połączeniami (np. schemat blokowy, ideowy),
- zorientowane na dane – priorytetem jest organizacja danych w systemie a nie sposób jego działania (np. diagramy Jacksona),
- heterogeniczne – łączą wybrane cechy różnych modeli (np. CDFG).

Ważnymi cechami modelu są m.in. wspieranie opisu behawioralnego oraz reprezentacja graficzna (Harel 1987). W chwili obecnej większość narzędzi wspomagających projektowanie kładzie duży nacisk na pierwszą własność, umożliwiającą opis systemu na poziomie jego zachowania, bez konieczności poznawania szczegółów realizacyjnych. Reprezentacja graficzna ma też niebagatelną rolę, ze względu na większą przejrzystość i czytelność projektowanego systemu. Typowymi narzędziami bez wsparcia graficznego są języki modelowania. Często umożliwiają one nie tylko opis zachowawczy projektowanego systemu, ale również strukturalny. Są to tzw. języki opisu sprzętu HDL. Najpopularniejszymi są VHDL (ang. *Very high speed integrated circuit Hardware Description Language*), VERILOG, SDL (ang. *Specification and Description Language*) czy ESTEREL (Berry 1998).

Poniżej przedstawiono krótkie charakterystyki wybranych modeli, najbardziej pożądanych z punktu widzenia tematyki pracy. Największy nacisk położono na precyzyjne określenie modelu interpretowanych sieci Petriego, wraz z najważniejszymi rozszerzeniami.

Zastosowano następującą zasadę nazewnictwa. Duże litery łacińskie (np. P, T) oznaczają nazwy zbiorów, a małe litery łacińskie (np. p, t) elementy tych zbiorów. Małe greckie litery (np. α, ε) oznaczają funkcje należące do modelu. Funkcje pomocnicze oznaczane są charakterystycznymi małymi literami alfabetu łacińskiego (np. ac, cond).

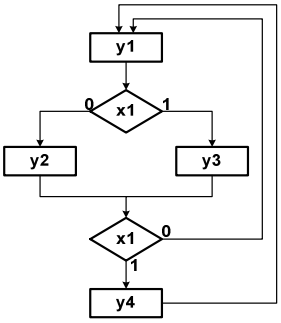
1.2.1. Grafy algorytmiczne

Jednym z pierwszych opracowanych modeli reprezentacji systemów sterowania był automat algorytmiczny ASM (ang. *Algorithmic State Machine*) (rys. 1.3).

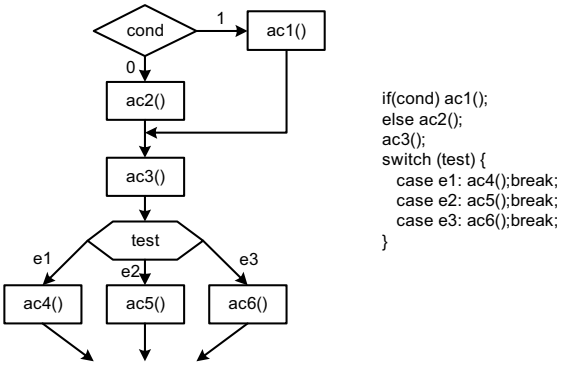
W najbardziej podstawowej wersji przedstawia się go jako graf zorientowany z dwoma rodzajami wierzchołków: decyzyjnymi i operacyjnymi, połączonymi łukami skierowanymi. Wierzchołki operacyjne służą do modelowania akcji wyjściowych systemu, a wierzchołki decyzyjne do wprowadzania warunków zmiany stanu.

Bardzo podobnymi modelami do automatu algorytmicznego są m.in. graf przepływu danych DFG (ang. *Data Flow Graph*) oraz jego rozszerzenie CDFG (ang. *Control-Data Flow Graph*), wspierający zarówno sterowanie jak i przetwarzanie danych (rys. 1.4). Model ten jest dodatkowo doposażony (względem DFG) w węzły *if*,

case oraz *loop*. Znalazł on zastosowanie m.in. w pakiecie wspomagającym projektowanie zintegrowane LYCOS, opracowanym na uniwersytecie w Lyngby w Danii (Staunstrup *i in.* 1997).



Rys. 1.3. Prosty przykład automatu algorytmicznego



Rys. 1.4. Przykład grafu CDFG

1.2.2. Skończony automat cyfrowy FSM

Najpopularniejszym modelem opisu układów sterowania jest niewątpliwie automat skończony FSM (ang. *Finite State Machine*).

Definicja 1.7 *Automat skończony jest szóstką uporządkowaną:*

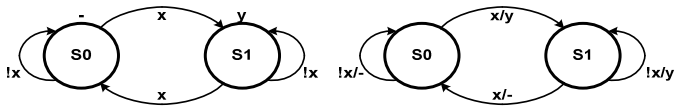
$$FSM = (X, S, Y, \delta, \lambda) \tag{1.1}$$

gdzie: $X = \{x_1,...,x_i\}$ *– skończony niepusty zbiór wejść;* $S = \{s_1,...,s_j\}$ *– skończony niepusty zbiór stanów;* $Y = \{y_1,...,y_k\}$ *– skończony niepusty zbiór wyjść;* δ *– funkcja przejść, taka że* δ*:* $D_\delta \rightarrow S$*, przy czym* $D_\delta \subset X \times S$*;* λ *– funkcja wyjść, taka że* λ*:* $D_\lambda \rightarrow Y$*, przy czym* $D_\lambda \subset X \times S$ *dla automatu Mealy'ego albo* $D_\lambda \subset S$ *dla automatu Moore'a.*

Graficznie przedstawiany jest grafem skierowanym o wierzchołkach reprezentujących stany i łukach odzwierciedlających przejścia pomiędzy stanami (rys.

1.5). Warunki skojarzone są z łukami, a wektor wyjściowy z łukami (dla automatu Mealy'ego, rys. 1.5-a) lub ze stanami (dla automatu Moore'a, rys. 1.5-b).

Uwaga: Często przyjmuje się konwencję podawania wektora wyjściowego w postaci zbioru tych liter alfabetu wyjściowego, które posiadają wartość 1. Jeżeli w zbiorze dana litera nie występuje, to przyjmuje się jej wartość jako 0.

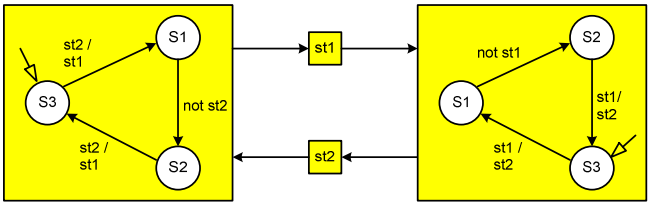


Rys. 1.5. Przykład automatu skończonego FSM: a) Moore'a, b) Mealy'ego

1.2.3. Rozszerzenia automatu cyfrowego

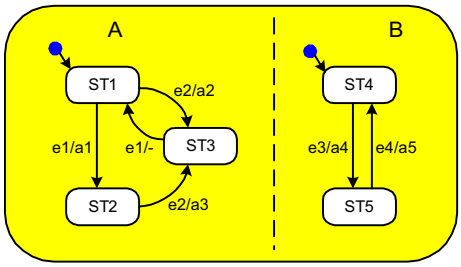
Jakkolwiek skończony automat cyfrowy jest prostym i popularnym modelem, to jednak budowa bardziej skomplikowanych systemów sterowania, w których występuje np. potrzeba wsparcia współbieżności jest zadaniem skomplikowanym. Można wówczas budować taki system z wykorzystaniem kilku automatów, których działanie synchronizowane jest określonym zbiorem wewnętrznych sygnałów synchronizujących. Połączenie takie nosi nazwę powiązanych grafów stanów (Belhadj *i in.* 1993).

Rozwinięciem takiej metodologii jest m.in. model CFSM (ang. *Co-Design Finite State Machine*), w którym zastosowano oprócz asynchronicznej komunikacji międzymodułowej również przetwarzanie danych (rys. 1.6) (Balarin 1997). Model ten znalazł zastosowanie w takich pakietach projektowania zintegrowanego jak POLIS (opracowany na Uniwersytecie w Berkeley), czy ESTEREL (opracowany przy współpracy Institut National de Recherche en Informatique et en Automatique oraz Centre de Mathématiques Appliquées w Sophia-Antipolis).



Rys. 1.6. Reprezentacja graficzna CFSM

Oprócz wspierania współbieżności, bardzo ważne są również inne aspekty opisu systemów reaktywnych. Należy do nich możliwość opisu hierarchicznego. Modelem nabierającym coraz większej popularności jest HCFSM (ang. *Hierarchical Concurrent Finite State Machine*), którego przykładami (rys. 1.7) mogą być m.in. ECSM (ang. *Extended CSM*) (Ratajczak i Pabiś 1999) lub znany ze standardu UML diagram stanów (ang. *Statechart*).



Rys. 1.7. Przykład automatu HCFSM

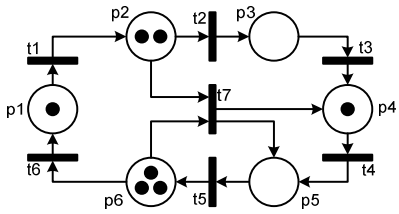
Ten ostatni model zastosowano m.in. w komercyjnym oprogramowaniu CAD: *Statemate* firmy I-Logix, *BetterState* firmy WindRiver oraz *VisualState* wyprodukowany przez IAR Systems (http://www.ilogix.com, http://www.windriver.com, http://www.iar.com).

Prezentowane modele w różnym stopniu wspierają wymienione wcześniej elementy opisu systemów reaktywnych, dlatego też ich zastosowanie może być wydajne tylko dla pewnych określonych klas systemów. Ostatnim prezentowanym modelem są sieci Petriego. Pozwalają one w naturalny sposób opisywać procesy współbieżne, a ich znane rozszerzenia (np. sieci interpretowane, sieci czasowe oraz sieci hierarchiczne) uwzględniają wiele istotnych elementów opisu zachowawczego systemów sterowania. Wobec powyższego oraz faktu, iż powstało wiele metod analizy i weryfikacji formalnej sieci (Andrzejewski i Łabiak 1999, Biliński 1996, Heiner 1998, Miczulski 2001, Murata 1989, Peterson 1981), mogą one stanowić dogodny model pośredni wspomagający projektowanie takich systemów (Mirkowski i Skowroński 1997, Skowroński 2000, Stoy 1995).

1.2.4. Klasyczne sieci Petriego

Pojęcia i definicje zawarte w tym podpunkcie są ogólnie znane i można je odnaleźć w literaturze przedmiotu (Murata 1989, Peterson 1981, Petri 1962, Starke 1987, Suraj i Szpyrka 1999).

Sieć Petriego w ujęciu klasycznym jest grafem skierowanym, posiadającym dwa rodzaje węzłów: *miejsca* – reprezentowane przez okręgi oraz *tranzycje* – reprezentowane przez prostokąty lub pogrubione beleczki. (rys. 1.8).



Rys. 1.8. Przykład klasycznej sieci Petriego

Łuki skierowane łączą miejsca z tranzycjami. Stan sieci określany jest przez *znakowanie*, charakteryzowane rozmieszczeniem *znaczników* – grubych punktów wewnątrz miejsc. Z miejscami związana jest funkcja pojemności miejsca, określająca maksymalną liczbę znaczników jakie może dane miejsce pomieścić, a z łukami – funkcja wagi łuków, określająca liczbę znaczników, które jednocześnie mogą się po danym łuku przemieścić.

Definicja 1.8 *Sieć Petriego jest szóstką uporządkowaną:*

$$PN = (P, T, F_o, \kappa, \omega, m_0) \tag{1.2}$$

gdzie: P – jest niepustym, skończonym zbiorem miejsc (np.: P = {p_1,p_2,...,p_n}), T – jest niepustym, skończonym zbiorem tranzycji (np.: T = {t_1,t_2,...,t_m}), oraz $P \cap T = \varnothing$, Fo – jest niepustym, skończonym zbiorem łuków skierowanych, takich że: $Fo \subset (P \times T) \cup (T \times P)$, κ– jest funkcją pojemności miejsc κ: $P \rightarrow N \cup \{\infty\}$ (N jest zbiorem liczb naturalnych), ω – jest funkcją wagi łuków ω: $F_o \rightarrow N$, m0 – jest funkcją znakowania początkowego m0: $P \rightarrow N \cup \{0\}$.

Ponadto można zdefiniować szereg pojęć pomocniczych, ułatwiających opis działania sieci oraz metody jej weryfikacji.

Definicja 1.9 *Funkcją aktywności miejsca nazywamy taką funkcję ac(p) → {true, false}, która każdemu miejscu przypisuje wartość true, jeśli miejsce to posiada co najmniej jeden znacznik, oraz false w sytuacji przeciwnej.*

Definicja 1.10 *Znakowaniem (markowaniem) m nazywamy taką funkcję, która każdemu miejscu przyporządkowuje pewną liczbę, równą liczbie znaczników przebywających w tym miejscu w określonym momencie czasu.*

Często znakowanie przedstawia się jako wektor (dla przykładu z rys. 1.8): m_0=<1,2,0,1,0,3>

Definicja 1.11 *Zbiorem miejsc wejściowych tranzycji t nazywamy zbiór $P_t^{in(o)}$, taki że: $P_t^{in(o)} = \{p \in P : (p,t) \in F_0\}$. Zbiorem miejsc wyjściowych tranzycji t nazywamy zbiór P_t^{out}, taki że: $P_t^{out} = \{p \in P : (t,p) \in F_0\}$.*

Definicja 1.12 *Zbiorem węzłów N nazywamy sumę zbiorów miejsc i tranzycji $N=P \cup T$.*

Definicja 1.13 *Współczynnikiem współbieżności sieci nazywamy liczbę, odpowiadającą maksymalnej liczbie jednocześnie oznakowanych miejsc tej sieci.*

Działanie sieci wyznaczone jest regułami wykonywania (odpalania) tranzycji dla określonego znakowania m.

Definicja 1.14 *Słabą regułą wykonania tranzycji nazywamy warunek: tranzycja $t \in T$ może być wykonana, gdy jej każde miejsce wejściowe zawiera co najmniej tyle znaczników ile wynosi waga łuku (p,t).*

Definicja 1.15 *Mocną regułą wykonania tranzycji nazywamy warunek: tranzycja $t \in T$ może być wykonana, gdy jej każde miejsce wejściowe zawiera co najmniej tyle*

znaczników ile wynosi waga łuku (p,t) oraz każde miejsce wyjściowe ma wystarczającą pojemność by przyjąć tyle znaczników ile wynosi waga łuku (t,p).

Wynikiem wykonania tranzycji t przy znakowaniu m jest nowe znakowanie m', takie że:

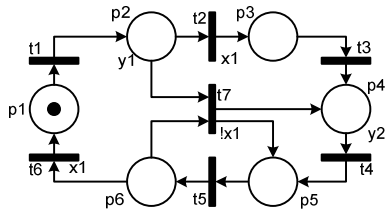
- od liczb przyporządkowanych wszystkim miejscom wejściowym tranzycji t odejmowane są liczby równe odpowiednim wagom łuków (p,t),
- do liczb przyporządkowanych wszystkim miejscom wyjściowym tranzycji t dodawane są liczby równe odpowiednim wagom łuków (t,p).

Dla przykładu z rys. 1.8 po wykonaniu tranzycji t1 (przy założeniu wagi łuków (p1,t1)≥1 i (t1,p2) ≥1 oraz pojemności miejsca ω(p2) ≥3) nowe znakowanie będzie miało postać: m'=<0,3,0,1,0,3>.

Powstało wiele klas sieci Petriego, uwzględniających pewne specyficzne własności modelu ogólnego oraz wiele rozszerzeń umożliwiających modelowanie wybranych systemów rzeczywistych (elektroniczne systemy reaktywne, sieci energetyczne, procesy przemysłowe). Z punktu widzenia niniejszego opracowania zdecydowano się jedynie na przedstawienie tych klas sieci, mających istotne znaczenie w modelowaniu współbieżnych, cyfrowych systemów sterowania.

1.2.5. Interpretowana sieć Petriego

Istotą sterowania jest odpowiednia reakcja systemu na zmiany parametrów wejściowych. Zakładając, iż parametry wejściowe będą opisane alfabetem wejściowym, a reakcja systemu będzie opisana na alfabecie wyjściowym, wówczas można dodatkowo skojarzyć miejsca i tranzycje sieci odpowiednio z literami alfabetu wyjściowego oraz z warunkami budowanymi na alfabecie wejściowym. Pozwala to na interpretację miejsc analogicznie do stanów w automacie skończonym, a tranzycji do akcji związanych ze zmianą stanu (rys. 1.9) (Adamski 1992, Adamski 2000, Adamski 2001, Banaszak *i in.* 1993, Kalinowski 1984, Varadharajan i Baker 1987).



Rys. 1.9. Przykład interpretowanej sieci Petriego

Definicja 1.16 *Sieć interpretowaną można przedstawić jako:*

$$IPN = \{P, T, F_o, X, Y, m_0, \delta, \lambda)\} \qquad (1.3)$$

gdzie: P,T,F_0,m_0 – definiowane są tak jak poprzednio, X i Y– są alfabetami wejściowym i wyjściowym, δ – jest funkcją przyporządkowującą każdej tranzycji pewien podzbiór z przestrzeni wejścia δ: $T \rightarrow \boldsymbol{X}$ (formułę logiczną cond), λ – jest

funkcją przyporządkowującą każdemu miejscu pewien podzbiór z przestrzeni wyjścia $\lambda: P \rightarrow \boldsymbol{Y}$.

Uwaga: W modelu pominięto funkcje pojemności miejsc oraz wagi łuków, gdyż dla sieci interpretowanych stosuje się ich ograniczenie do κ(p)=1 oraz ω(f)=1 (Banaszak *i in.* 1993, Reisig 1988).

Działanie sieci wyznaczone jest wówczas poprzez warunki przygotowania tranzycji oraz akcje związane z jej wykonaniem. Dla zachowania jednolitości opisu warunków i akcji we wszystkich omawianych rodzajach sieci, autor wprowadził następującą formę ich opisu:

Warunki przygotowania tranzycji *t*:

$$\forall_{p \in P_t^{in(o)}} ac(p) = true \tag{1.4}$$

$$cond(t) = true \tag{1.5}$$

Akcje związane z wykonaniem tranzycji *t*:

$$\forall_{p \in P_t^{in(o)}} ac(p) := false \tag{1.6}$$

$$\forall_{p \in P_t^{out}} ac(p) := true \tag{1.7}$$

$$\forall_{p \in P_t^{in(o)}} \forall_{y \in \lambda(p)} y := false \tag{1.8}$$

$$\forall_{p \in P_t^{out}} \forall_{y \in \lambda(p)} y := true \tag{1.9}$$

Dodatkowo można wprowadzić najbardziej istotne własności behawioralne sieci interpretowanych, których badanie może posłużyć do wstępnego określenia prawidłowości przyjętego modelu opisu projektowanego systemu.

Definicja 1.17 *Sieć jest uważana za bezpieczną, jeżeli liczba znaczników w każdym miejscu nie przekroczy jednego, dla dowolnego znakowania dostępnego z* m_0.

Łatwo zauważyć, iż miejscem potencjalnie niebezpiecznym jest takie miejsce, które jest wyjściowym dla więcej niż jednej tranzycji. Dla przykładu z rys. 1.9 są to miejsca P4 i P5 (ale ze względu na fakt, iż nigdy nie zostanie odpalona tranzycja t7, miejsca te można uznać za bezpieczne).

Definicja 1.18 *Sieć jest uważana za żywą, jeżeli jest możliwe odpalenie dowolnej tranzycji w sieci (poprzez postępującą sekwencję odpaleń), dla dowolnego znakowania dostępnego z* m_0.

Dla rys. 1.9 ze znakowania m_0 nie można odpalić tranzycji t7, a więc nie spełnia ona warunku żywotności.

Definicja 1.19 *Sieć jest uważana za trwałą, jeżeli dla dowolnych dwóch dostępnych tranzycji, odpalenie jednej z nich nie spowoduje zablokowania drugiej.*

Łatwo zauważyć, iż miejscem potencjalnie zagrażającym warunkowi trwałości jest takie miejsce, które jest wejściowym dla więcej niż jednej tranzycji. Dla przykładu z rys. 1.9 mogą to być miejsca P2 i P6 (ale ze względu na fakt, iż nigdy nie zostanie

odpalona tranzycja t7, warunek trwałości jest zachowany). Bliskoznacznym pojęciem do trwałości sieci jest determinizm.

Definicja 1.20 *Sieć jest uważana za deterministyczną, jeżeli dla dowolnego znakowania dostępnego z m_0 sieć może być przeprowadzona tylko do jednego znakowania następnego.*

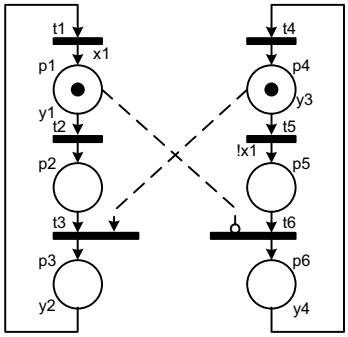
Znanymi metodami sprowadzania sieci niedeterministycznej do deterministycznej są (Banaszak *i in.* 1993):

- dodatkowe interpretowanie tranzycji sygnałami wejściowymi,
- wprowadzenie priorytetów.

Priorytet jest rozumiany jako funkcja przyporządkowująca każdej tranzycji pewną liczbę naturalną $\pi \in N$, taką że z dwóch dostępnych tranzycji wykonana będzie ta, której liczba określająca priorytet jest wyższa.

1.2.6. Interpretowana sieć Petriego z łukami zezwalającymi i zabraniającymi

Często zachodzi sytuacja, w której pożądane jest uwarunkowanie wykonania określonej tranzycji aktywnością (lub nieaktywnością) pewnego miejsca, bez wpływu na zmianę jego stanu (aktywności) po wykonaniu tranzycji. Przykładem może być sytuacja synchronizacji procesów współbieżnych. Wygodnie jest wówczas skorzystać z tzw. łuków zezwalających oraz zabraniających (rys. 1.10), zdefiniowanych w dalszej części podrozdziału (Murata 1989).



Rys. 1.10. Przykład interpretowanej sieci Petriego z łukami zezwalającymi i zabraniającymi

Definicja 1.21 *Interpretowaną siecią Petriego z łukami zezwalającymi i zabraniającymi nazywamy ósemkę:*

$$IPNA = \{P, T, F, X, Y, m_0, \delta, \lambda\} \qquad (1.10)$$

gdzie: F jest zbiorem łuków, takich że: $F = F_o \cup F_e \cup F_i$, F_o: $F_o \subset (P \times T) \cup (T \times P)$ oraz $\underset{f \in F_o}{\forall} \omega(f) \neq 0$, F_e: $F_e \subset (P \times T)$ oraz $\underset{f \in F_e}{\forall} \omega(f) = 0$, F_i: $F_i \subset (P \times T)$ oraz $\underset{f \in F_i}{\forall} \omega(f) = 0$, symbole pozostałe takie jak w definicji1.16.

Dodatkowo zdefiniować można zbiory miejsc zezwalających i zabraniających.

Definicja 1.22 *Zbiorem miejsc zezwalających tranzycji t nazywamy zbiór* $P_t^{in(e)}$*, taki że:* $P_t^{in(e)} = \{p \in P : (p,t) \in F_e\}$*. Zbiorem miejsc zabraniających tranzycji t nazywamy zbiór* $P_t^{in(i)}$*, taki że:* $P_t^{in(i)} = \{p \in P : (p,t) \in F_i\}$.

Warunki przygotowania tranzycji *t* (takie jak 1.4-1.5) oraz:

$$\forall_{p \in P_t^{in(e)}} \; ac(p) = true \tag{1.11}$$

$$\forall_{p \in P_t^{in(i)}} \; ac(p) = false \tag{1.12}$$

Akcje związane z wykonaniem tranzycji *t* (takie jak 1.6-1.9).

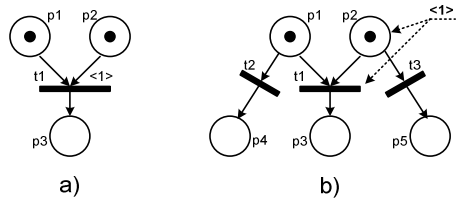
1.3. Interpretowana hierarchiczna sieć Petriego

Jakkolwiek można opisać dowolne działanie systemu reaktywnego z wykorzystaniem interpretowanych sieci Petriego, to jednak istnieją sytuacje, gdy opis taki będzie wymagał dużego nakładu pracy i może stać się nieczytelny, ze względu choćby na rozmiary sieci. Aby ułatwić proces modelowania, a tym samym i projektowania, niezbędnym wydaje się rozszerzenie klasycznego modelu sieci o takie elementy, które w wydajny sposób będą w stanie uprościć opis modelowanego systemu.

Bardzo ważnym elementem opisu rzeczywistych systemów sterowania jest możliwość modelowania zależności czasowych. W literaturze sieci Petriego napotkać można dwa podstawowe warianty ich realizacji:

- parametr czasowy przyporządkowany do tranzycji i oznaczający czas jej wykonania,
- parametr czasowy przyporządkowany do tranzycji i oznaczający czas jej przygotowania,

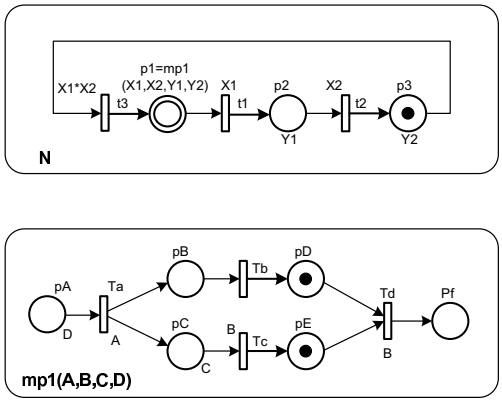
Pierwszy wariant jest charakterystyczny dla sieci wywodzących się od modelu nazywanego Timed Petri Net (Ramchandani 1974), a drugi dla sieci Time Petri Net (Merlin 1974). Na rys. 1.11 przedstawiono ilustrację obu wariantów wraz z ich porównaniem do parametrów czasowych wprowadzonych w modelu HPN. Dla sieci typu Timed PN, parametr czasowy opisany na rysunku jako <1> informuje, że po zezwoleniu tranzycji t1, znaczniki z miejsc p1 i p2 zostaną usunięte, a do miejsca p3 zostaną wprowadzone dopiero po czasie <1>. Dla sieci typu Time PN, parametr czasowy opisany na rysunku jako <1> informuje, że po wprowadzeniu znaczników do miejsc p1 i p2, tranzycja t1 będzie dozwolona dopiero po czasie <1> (rys. 1.11-a).



Rys. 1.11. Parametry czasowe w sieciach Petriego

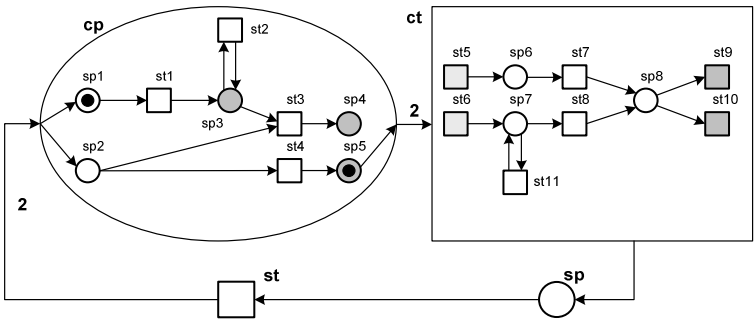
W sytuacji, jak na rys. 1.11-b, przypisanie czasu przygotowania do tranzycji t1 spowoduje, iż zostaną wykonane obie tranzycje boczne: t2 i t3. W modelu sieci HPN proponuje się użycie parametru czasu przypisanego do miejsca i oznaczającego min. czas, jaki upływa od momentu oznakowania miejsca do momentu opuszczenia z niego znacznika (podobnie jak w (David i Alla 1992)). W sytuacji powyższej, po wprowadzeniu znaczników do miejsc p1 i p2 wykonana zostanie tylko tranzycja t2. Technika taka wydaje się być bardziej naturalną w opisie zależności czasowych i poza tym może zwiększyć czytelność modelu. Łatwo bowiem zauważyć, iż w sytuacji gdy chce się blokować więcej niż jedną tranzycję wyjściową od pewnego miejsca, to wystarczy jedno przypisanie czasu do tego miejsca. W przypadku modelu Time PN należałoby czas przypisać do każdej tranzycji oddzielnie.

Wprowadzenie hierarchii do sieci Petriego, umożliwiło uproszczenie projektowania nie tylko na poziomie zapisu modelu, ale również jego analizy. Znanych jest wiele rozszerzeń sieci wspierających hierarchię. Najbardziej popularne to HCPN – Hierarchical Coloured PN (Jensen 1997, Węgrzyn 1998), OOPN – Object Oriented PN (Esser 1996, Vojnar 1997), HOONETS – Hierarchical Object-Oriented PN (Hong i Bae 1998), THORN – Timed Hierarchical Object-Related Nets (Schof *i in.* 1995), HHPN – Hybrid High-Level PN (Wieting i Sonnenschein 1995), MacroNets (Fernandes *i in.* 1997, Węgrzyn i Adamski 1999), PetriCharts (Holvoet i Verbaeten 1995), GrafChart – hierarchiczne rozszerzenie modelu Grafcet (David i Alla 1992), bazujące na High-Level PN (Johnson i Årzen 1994). Ze względu na przydatność do opisywania systemów sterowania cyfrowego na uwagę zasługują HCPN, MacroNets oraz PetriCharts. Przykłady ich notacji graficznej przedstawiono na rys. 1.12-1.13. Hierarchia jest w nich wprowadzana na różne sposoby. W sieciach HCPN oraz Macronets osiągana jest przez zwykłe podstawienie pewnego fragmentu sieci w miejsce makromiejsca lub makrotranzycji (hierarchia strukturalna), natomiast w sieciach PetriCharts zauważyć można elementy hierarchii behawioralnej, w której pewna podsieć jest aktywowana w momencie oznaczenia związanego z nią makrowęzła.



Rys. 1.12. Przykład sieci Macronets (zaczerpnięte z (Fernandes *i in.* 1997))

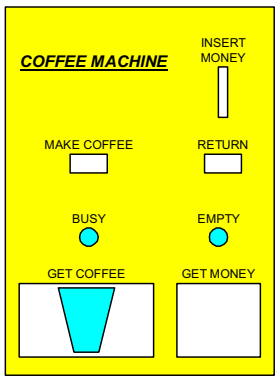
W tym drugim przypadku podstawianie jest niewystarczające, chociażby ze względu na fakt, iż odpowiedni makrowęzeł z sieci podstawowej nie jest usuwany. Sieci PetriCharts są szczególnie interesujące ze względu na zapożyczenie wielu reguł opisu ze standardu UML (ang. *Unified Modelling Language*), a w szczególności z diagramów Statecharts, dzięki czemu osiąga się w nich pewną spójność z najnowszymi trendami modelowania, a także dużą przejrzystość zapisu.



Rys. 1.13. Przykład sieci PetriCharts (zaczerpnięte z (Holvoet i Verbaeten 1995))

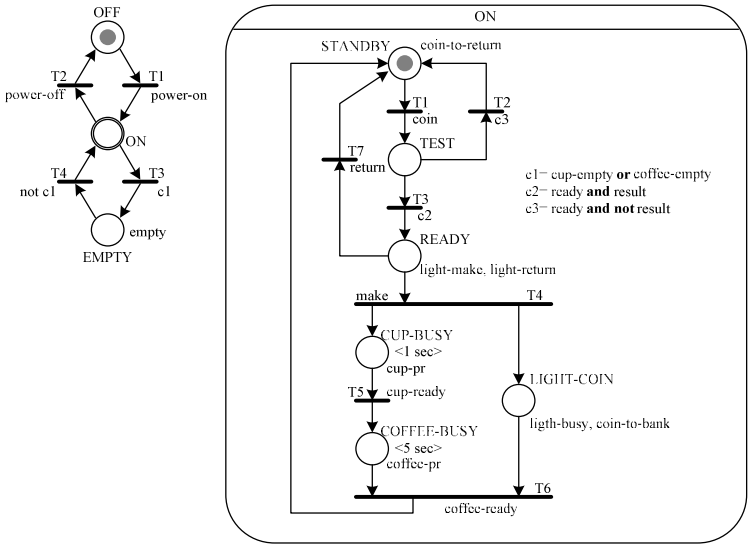
Nie wspiera on jednak bardzo interesującego elementu znanego właśnie z UML, a mianowicie historii (bardziej szczegółowy opis w rozdz. 1.3.1). Prezentowane modele niestety nie uwzględniają też opisu zależności czasowych.

Wobec powyższego zdecydowano się na próbę określenia takiego rozszerzenia sieci Petriego, które przy zachowaniu spójności semantycznej na najniższym poziomie abstrakcji z sieciami interpretowanych, uwzględniałyby w szerszym stopniu wszystkie omawiane wcześniej elementy opisu, a dodatkowo adoptowałyby przejrzystą szatę graficzną diagramów Statecharts. W konsekwencji doprowadziło to do powstania modelu sieci HPN (Hierarchical Petri Net), przedstawionego po raz pierwszy w pracy (Andrzejewski 2001c). Jako przykład modelowania z użyciem sieci hierarchicznej można podać nieco zmodyfikowaną wersję sterownika automatu do kawy (rys. 1.14), zaczerpniętego z opracowania (Kyeyune 2000).



Rys. 1.14. Automat do kawy

W automacie tym po wrzuceniu monety (i zaakceptowaniu jej przez automat), podświetlone zostają przyciski MAKE COFFEE oraz RETURN, służące odpowiednio do przygotowania kawy albo zwrotu monety. Po wybraniu MAKE COFFE kawa jest przygotowywana, co sygnalizowane jest lampką kontrolną BUSY. Po zakończeniu procesu, automat przechodzi w stan oczekiwania. Lampka kontrolna EMPTY sygnalizuje brak kubeczków lub kawy. Model procesu przedstawiono na rys. 1.15.



Rys. 1.15. Automat do kawy – model sieciowy

1.3.1. Podstawowe własności sieci hierarchicznych

Synchronizm

Jednym z podstawowych założeń przyjętych w modelu hierarchicznych sieci Petriego jest synchronizm. Polega on na tym, iż zarówno zmiany stanu wewnętrznego sieci, jak i zmiany stanów sygnałów wyjściowych, następują pod wpływem zmian sygnałów wejściowych w określonych momentach czasu, przyporządkowanych dyskretnej skali czasu. Pociąga to za sobą możliwość jednoczesnego wykonania wielu tranzycji, spełniających warunki przygotowania (oczywiście przy założeniu spełnienia własności trwałości sieci). Dzięki przyjęciu synchronizmu możliwe jest istotne uproszczenie metod weryfikacji formalnej oraz realizacji praktycznej tego modelu.

Hierarchia

Własność hierarchiczności jest realizowana poprzez taką dekompozycję sieci, w której z wyróżnionymi miejscami skojarzone są inne sieci. Miejsca te nazywane są makromiejscami, a przyporządkowane im sieci – podsieciami. Podsieć nie przyporządkowana do żadnego makromiejsca nazywana jest siecią podstawową, a podsieć nie zawierającą żadnych makromiejsc nazywa się siecią końcową.

Warunkiem aktywności danej podsieci jest posiadanie znacznika przez makromiejsce z nią skojarzone. Koncepcja taka pozwala nie tylko na konstruowanie bardziej przejrzystych projektów systemów o wysokim stopniu skomplikowania, ale również umożliwia badanie wybranych własności zachowawczych dla każdej podsieci z osobna, co zdecydowanie upraszcza proces analizy i weryfikacji formalnej projektu pod kątem sprawdzania własności opisanych w rozdz. 1.2.5 (bezpieczeństwo, żywość, trwałość oraz determinizm).

Historia

Często zachodzi potrzeba zapamiętania konfiguracji stanu wewnętrznego w wybranej podsieci. W sieciach hierarchicznych przewidziano możliwość obsługi takiej sytuacji, poprzez przypisanie atrybutu historii do określonego makromiejsca {H}. Przy opuszczaniu z niego znacznika, położenie znaczników wewnątrz podsieci z nim związanej jest zapamiętywane i po jego ponownej aktywacji, znaczniki są wprowadzane do miejsc ostatnio aktywnych. Dla wygody wprowadzono też możliwość przypisania atrybutu historii do wszystkich podsieci podrzędnych w strukturze hierarchii. Służy do tego operator {H*}.

Reakcja systemu

W sieciach hierarchicznych istnieje możliwość definiowania etykiety nazywanej *action*, przyporządkowywanej do węzła n. Etykieta ta tworzona jest na zbiorze S (zgodnie z ograniczeniami def. **1.23**) i składa się z elementów tego zbioru oddzielonych przecinkami, określających pewien podzbiór sygnałów aktywnych w momencie aktywności węzła n. Przy czym dla miejsca sygnały będą aktywne tak długo, jak długo miejsce to będzie posiadało znacznik, a dla tranzycji tak długo, jak długo aktywna jest ta tranzycja (ściślej w przedziale czasu wyznaczonym przez funkcję $\tau(t)$ i dyskretną skalę czasu).

Parametry czasowe

Z węzłami można kojarzyć parametry czasowe (patrz funkcja $\tau(t)$ def. 1.23). Przypisanie czasu t z dyskretnej skali czasu do miejsca p wyznacza min czas aktywności tego miejsca. To znaczy, że tranzycja, dla której miejsce p jest miejscem wejściowym, będzie dozwolona dopiero po upłynięciu czasu t licząc od momentu aktywacji miejsca p. Parametr czasowy przypisany do tranzycji wyznacza czas aktywności tranzycji i oznacza, iż po usunięciu znaczników ze wszystkich miejsc wejściowych tranzycji t dopiero po czasie t nastąpi wprowadzenie znaczników do wszystkich miejsc wyjściowych tej tranzycji. Niestety komplikuje to w znaczący sposób algorytmy analizy funkcjonalnej i weryfikacji formalnej sieci. Istnieją jednak znane metody analizy sieci czasowych (Adamski 1990, Berthomieu i Diaz 1991, Bolton 1990, Ramamoorthy i Ho 1980), które można wykorzystać w sieciach hierarchicznych. Wobec czego właściwość ta może w praktyce być bardzo użyteczna przy opisywaniu systemów silnie zależnych czasowo.

1.3.2. Model formalny interpretowanych sieci hierarchicznych

Definicja 1.23 *Interpretowaną hierarchiczną siecią Petriego (dalej siecią hierarchiczną) nazywamy n-tkę:*

$$HPN = \{P,T,F,S,\mathcal{T},\chi,\psi,\lambda,\alpha,\varepsilon,\tau\} \quad (1.13)$$

gdzie:

1) *P – jest skończonym niepustym zbiorem miejsc.*
2) *T – jest skończonym niepustym zbiorem tranzycji.*
3) *F – jest skończonym niepustym zbiorem łuków, takim że* $F = F_o \cup F_e \cup F_i$, *gdzie* F_o, F_e, F_i *oznaczają odpowiednio zbiory łuków zwykłych, zezwalających oraz zabraniających.*
4) *S – jest skończonym niepustym zbiorem sygnałów, takim że:* $S=X\cup Y\cup L$, *gdzie X, Y i L oznaczają odpowiednio alfabety wejściowy, wyjściowy oraz lokalny.*
5) $\mathcal{T}$ *– jest dyskretną skalą czasu.*
6) χ*:* $P\rightarrow 2^{\dot{N}}$, *jest funkcją hierarchii, określającą zbiór bezpośrednich podwęzłów miejsca p; wyrażenie* χ^* *oznacza przechodnie zwrotne dopełnienie funkcji* χ, *takie że dla każdego* $p \in P$ *zachodzą następujące warunki:*

a) $p \in \chi^*(p)$
b) $\chi(p) \in \chi^*(p)$
c) $p' \in \chi^*(p) \Rightarrow \chi(p') \subseteq \chi^*(p)$.

7) ψ*:* $P\rightarrow\{true,false\}$, *jest dwuwartościową funkcją historii, przyporządkowującą każdemu miejscu p takiemu że:* $\chi(p) \neq \varnothing$ *atrybut historii. Dla miejsc zwykłych funkcji się nie określa.*
8) λ*:* $N \rightarrow 2^S$, *jest funkcją etykietującą, przypisującą do węzłów wyrażenia budowane z elementów zbioru S. Obowiązują przy tym następujące zasady: miejsca można etykietować tylko ze zbioru* $Y\cup L$, *i etykieta ta (action) oznacza akcje związane z miejscami; etykieta tranzycji może się składać z trzech składników:*

cond – tworzony na zbiorze $X\cup L\cup\{0,1\}$ *i oznaczający warunek nałożony na tranzycję, będącym wyrażeniem logicznym generowanym z wykorzystaniem operatorów logicznych not, or i and, (brak warunku jest równoznaczny z przypisaniem cond=true),*

abort – tworzony jak cond, ale pozostający w innym stosunku logicznym ogólnego warunku zezwolenia tranzycji, graficznie poprzedzany znakiem #, (brak warunku jest równoznaczny z przypisaniem cond=false),

action – tworzony ze zbioru $Y\cup L$ *i oznaczający akcje związane z tranzycjami, graficznie poprzedzany znakiem /.*

9) α*:*$P\rightarrow\{true,false\}$, *jest funkcją znakowania początkowego, przyporządkowującą każdemu miejscu atrybut miejsca początkowego. Miejsca początkowe graficznie wyróżnione są znacznikiem wewnątrz okręgu reprezentującego miejsce.*
10) ε*:* $P \rightarrow \{true,false\}$, *jest funkcją znakowania końcowego, przypisującą do każdego miejsca atrybut miejsca końcowego. Miejsca końcowe graficznie wyróżnione są znakiem* $\times$ *wewnątrz okręgu reprezentującego miejsce.*
11) τ*:* $N \rightarrow \mathcal{T}$, *jest funkcją czasu, przypisującą liczbę z dyskretnej skali czasu do zbioru węzłów N.*

Uwaga: Metodykę formalizowania zaczerpnięto z pracy (Magiollo-Schettini i Merro 1996).

1.3.3. Działanie sieci hierarchicznych

Działanie sieci hierarchicznych określają warunki zezwolenia tranzycji oraz akcje związane z jej wykonaniem.

Niech t_0 oznacza moment aktywacji węzła $n \in N$. Funkcja $\tau(n)$ w momencie t_0 przypisuje pewną, ściśle określoną liczbę z dyskretnej skali czasu do węzła n: $\tau(n, t_0)=t$, gdzie $t \in T$. W kolejnych chwilach czasu liczba ta ulega dekrementacji i po czasie t osiąga wartość 0: $\tau(n, t_0+t)=0$. Używane dalej oznaczenie $\tau(n)=0$ opisuje wartość funkcji w momencie czasu, w którym jest ona równa 0.

Dla czytelniejszego przedstawienia warunków przygotowania tranzycji, można wprowadzić dodatkowe definicje, określające zbiory miejsc końcowych.

Definicja 1.24 *Zbiorem miejsc końcowych w podsieci skojarzonej z makromiejscem p jest taki zbiór* P_p^{end}*, że:*

$$\forall_{p' \in \chi(p)} \varepsilon(p') = true \Rightarrow p' \in P_p^{end} \qquad (1.14)$$

Definicja 1.25 *Funkcją zbioru miejsc końcowych nazywamy taką funkcję* $\xi: P \rightarrow P$*, która dla makromiejsca p zwraca jego zbiór miejsc końcowych:*

$$\xi(p) = P_p^{end} \qquad (1.15)$$

wyrażenie ξ^* *oznacza przechodnie zwrotne dopełnienie funkcji* ξ*, takie że dla każdego* $p \in P$ *oraz* $\chi(p) \neq \emptyset$*, zachodzą następujące warunki:*

$p \in \xi^*(p)$ (1.16)

$\xi(p) \in \xi^*(p)$ (1.17)

$p' \in \xi^*(p) \Rightarrow \xi(p') \subseteq \xi^*(p)$ (1.18)

Definicja 1.26 *Znakowaniem końcowym podsieci nazywamy każde takie znakowanie, że należą do niego wszystkie miejsca końcowe tej podsieci.*

Warunki przygotowania tranzycji *t*:

$$\exists_{p \in P} la(t) = p \Rightarrow ac(p) = true \qquad (1.19)$$

$$\forall_{p \in P_t^{in(o)} \cup P_t^{in(e)}} ac(p) = true \qquad (1.20)$$

$$\forall_{p \in P_t^{in(i)}} ac(p) = false \qquad (1.21)$$

$$cond(t) = true \qquad (1.22)$$

$$\forall_{p \in P_t^{in(o)}} \chi(p) \neq \emptyset \Rightarrow \forall_{p' \in \xi^*(p)} (ac(p') = true \text{ oraz } \tau(p') = 0) \qquad (1.23)$$

$$\forall_{p \in P_t^{in(o)}} \tau(p) = 0 \qquad (1.24)$$

$$abort(t) = true \qquad (1.25)$$

Uwaga: Ze wszystkich warunków można złożyć następujące wyrażenie logiczne (warunek ogólny): 1.19*1.20*1.21*(1.22*1.23*1.24+1.25), co oznacza możliwość zezwolenia tranzycji pomimo niespełnienia warunków 1.22, 1.23, oraz 1.24 jeśli tylko warunek 1.25 jest spełniony. Sytuacja taka znana jest pod nazwą *wywłaszczenia* (ang. *preemption*).

Akcje związane z wykonaniem tranzycji *t*:

$$\forall_{p\in P_t^{in(e)}} ac(p) := false \tag{1.26}$$

$$\forall_{p\in P_t^{in(e)}} \chi(p) \neq \varnothing \Rightarrow \forall_{p'\in\chi^*(p)} ac(p') := false \tag{1.27}$$

$$\forall_{p\in P_t^{in(e)}} \forall_{s\in action(p)} s := false \tag{1.28}$$

$$\forall_{p\in P_t^{in(e)}} \chi(p) \neq \varnothing \Rightarrow \forall_{p'\in\chi^*(p)} \forall_{s\in action(p')} s := false \tag{1.29}$$

$$\tau(t) = 0 \Rightarrow \forall_{p\in P_t^{out}} ac(p) := true \tag{1.30}$$

$$\forall_{p\in P_t^{out}} \chi(p) \neq \varnothing \Rightarrow \left(\forall_{p'\in\chi^*(p)} ac(la(p')) = true \text{ oraz } \psi(la(p')) = false \Rightarrow ac(p') := \alpha(p')\right) \tag{1.31}$$

$$\forall_{p\in P_t^{out}} \chi(p) \neq \varnothing \Rightarrow \left(\forall_{p'\in\chi^*(p)} ac(la(p')) = true \wedge \psi(la(p')) = true \wedge \exists_{p''\in\xi(la(p'))} ac(p'') = false \Rightarrow ac(p') := ac(p', t_e)\right) \tag{1.32}$$

$$\forall_{p\in P_t^{out}} \chi(p) \neq \varnothing \Rightarrow \left(\forall_{p'\in\chi^*(p)} ac(la(p')) = true \wedge \psi(la(p')) = true \wedge \forall_{p''\in\xi(la(p'))} ac(p'') = true \Rightarrow ac(p') := \alpha(p')\right) \tag{1.33}$$

$$\forall_{p\in P_t^{out}} \forall_{s\in action(p)} s := true \tag{1.34}$$

$$\forall_{p\in P_t^{out}} \chi(p) \neq \varnothing \Rightarrow \left(\forall_{p'\in\chi^*(p)} ac(p') = true \Rightarrow \forall_{s\in action(p')} s := true\right) \tag{1.35}$$

$$\tau(t, \iota_0) = \eta \Rightarrow \forall_{s\in action(t)} s := true \text{ w przedziale } <\iota_0, \iota_0 + \eta + 1> \tag{1.36}$$

gdzie ac(p,t_e) oznacza stan posiadania (lub nie posiadania) znacznika przez miejsce p w momencie czasu, w którym nastąpiło opuszczenie znacznika z makromiejsca la(p).

Uwaga: Akcje 1.30-1.35 wykonywane są dopiero, gdy τ(t) = 0. Akcja 1.36 jest wykonywana przez cały okres aktywności tranzycji t.

1.3.4. Pojęcia pomocnicze

W podrozdziale niniejszym zdefiniowano szereg pojęć pomocniczych, ułatwiających poruszanie się w tematyce sieci hierarchicznych, szczególnie wykorzystywanych w rozdziale 3.
Niech będzie dana sieć hierarchiczna HPN oraz miejsce p należące do zbioru miejsc tej sieci.

Definicja 1.27 *Miejsce p jest nazywane zwykłym, jeżeli* $\chi(p) = \varnothing$.

Definicja 1.28 *Miejsce p jest nazywane makromiejscem, jeżeli nie jest miejscem zwykłym:* $\chi(p) \neq \varnothing$.

Definicja 1.29 *Najbliższym przodkiem węzła n' (ang. lowest ancestor) nazywamy miejsce p, takie że: n'∈χ(p), co zapisać można: la(n')=p.*

Zbiór węzłów N^i, będący w stosunku hierarchii do miejsca p, tworzy podsieć Z^i.

Definicja 1.30 *Niech N^i oznacza zbiór węzłów skojarzonych funkcją hierarchii z makromiejscem p: $N^i = \chi(p)$, oraz niech F^i oznacza zbiór wszystkich łuków łączących węzły należące do zbioru N^i, wówczas podsiecią skojarzoną z makromiejscem p nazywamy taką sieć Z^i, że $Z^i = N^i + F^i$.*

Wymagane jest aby wszystkie podsieci skojarzone z makromiejscami były rozłączne, tzn. aby nie miały wspólnych węzłów oraz łuków.

Definicja 1.31 *Niech Z^i oraz Z^k oznaczają dowolne podsieci zawarte w HPN, oraz niech N^i, N^k oraz F^i, F^k będą odpowiednio zbiorami węzłów oraz łuków należących do podsieci Z^i i Z^k. Dwie podsieci Z^i i Z^k nazywamy rozłącznymi jeśli spełniają warunki:*

$$\forall_{n \in N^i} n \notin N^k \quad \text{oraz} \quad \forall_{f \in F^i} f \notin F^k$$

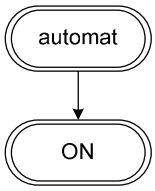
Definicja 1.32 *Siecią podstawową nazywamy taką sieć, w której dla żadnego węzła n należącego do tej sieci nie istnieje najbliższy przodek: la(n)=∅*

Definicja 1.33 *Podsiecią końcową nazywamy taką podsieć, w której każde miejsce p należące do tej podsieci jest miejscem zwykłym: χ(p) = ∅.*

Definicja 1.34 *Diagramem hierarchii nazywamy graf skierowany o strukturze drzewiastej, posiadający następujące własności:*

a) węzłami diagramu są podsieci,
b) węzłem wierzchołkowym jest sieć podstawowa,
c) łuki łączą dwa węzły, wg zasady: łuk bierze początek z węzła reprezentującego podsieć zawierającą makromiejsce p, oraz wskazuje węzeł reprezentujący podsieć Z', taką że χ(p)⊂Z',
d) węzły końcowe reprezentują podsieci końcowe.

Dla przykładu z rys. 1.15 można stworzyć następujący diagram hierarchii (rys. 1.16):



Rys. 1.16. Diagram hierarchii dla przykładu z rys. 1.15.

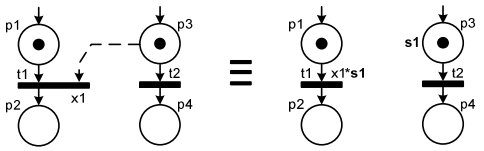
Definicja 1.35 *Łuk łączący dwa węzły w diagramie hierarchii wyznacza dwie podsieci, które względem siebie można nazwać: nadrzędną (wyznaczoną początkiem łuku), oraz podrzędną (wyznaczoną końcem łuku).*

1.3.5. Zgodność semantyczna z sieciami płaskimi

Wprowadzenie tylu cech do modelu sieci Petriego może spowodować wrażenie, iż ich weryfikacja formalna jest niezwykle trudna, a nawet wręcz niemożliwa. W podrozdziale niniejszym zostanie pokazane, że wszystkie wprowadzone elementy opisu, a więc: łuki zezwalające i zabraniające, parametry dynamiczne, hierarchia, historia, oraz wywłaszczenie – można przedstawić z wykorzystaniem tylko i wyłącznie elementów opisu interpretowanych sieci płaskich (to znaczy tak, aby były one równoważne pod względem funkcjonalnym). Jest to istotne, ze względu na możliwość wykorzystania profesjonalnych narzędzi umożliwiających formalną analizę płaskich sieci Petriego (np. DesignCPN, PNTools, PeNCAD) do analizy sieci hierarchicznych.

Łuki zezwalające

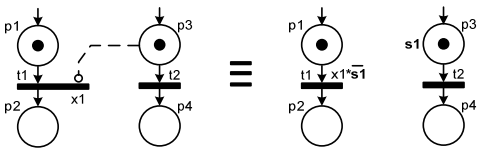
Łuki zezwalające można w prosty sposób zastąpić sygnałami wewnętrznymi. W tym celu należy wszystkie miejsca, z których biorą początek łuki zezwalające zaetykietować nazwami sygnałów wewnętrznych, przy czym dla każdego miejsca przypisuje się tylko jeden sygnał wewnętrzny niezależnie od liczby łuków z niego wychodzących; oraz każde takie miejsce etykietowane jest inną nazwą sygnału wewnętrznego. Następnie tranzycję, do której łuk dochodzi etykietuje się nazwą sygnału wewnętrznego, takiego jak przy odpowiednim miejscu zezwalającym, przy czym: etykieta ta stanowi warunek tranyzcji nałożony poprzez operator iloczynu logicznego na warunek istniejący (rys. 1.17).



Rys. 1.17. Zastąpienie łuku zezwalającego sygnałem wewnętrznym

Łuki zabraniające

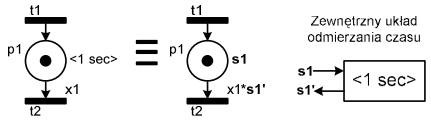
Łuki zabraniające można również zastąpić sygnałami wewnętrznymi. Procedura zamiany jest bardzo podobna, z tym, że tranzycję, do której łuk dochodzi, etykietuje się negacją sygnału wewnętrznego, takiego jak przy odpowiednim miejscu zabraniającym (rys. 1.18).



Rys. 1.18. Zastąpienie łuku zabraniającego sygnałem wewnętrznym

Parametr czasowy przypisany do miejsca

Parametry czasowe związane z miejscami mogą być postrzegane jako czasy przebywania znacznika w tych miejscach, co ilustruje rys. 1.19.



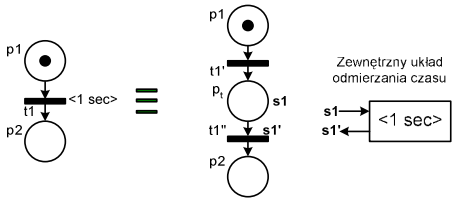
Rys. 1.19. Interpretacja parametru czasowego przypisanego do miejsca

Warunkiem zezwolenia tranzycji t2 jest minięcie określonego czasu (1 sec), licząc od momentu aktywacji miejsca p1. Sytuację taką można zamodelować z wykorzystaniem interpretowanych sieci płaskich i zewnętrznego układu odmierzania czasu (startowanego sygnałem s i sygnalizującego koniec akcji sygnałem s'). W tym celu należy wszystkie miejsca, do których przypisano parametry czasowe zaetykietować nazwami sygnałów wewnętrznych, przy czym każde takie miejsce etykietowane jest inną nazwą sygnału wewnętrznego. Następnie wszystkie tranzycje, dla których miejsca te są miejscami wejściowymi, etykietuje się odpowiednimi nazwami sygnałów wewnętrznych, takich jak przy odpowiednich miejscach wejściowych rozszerzonych o znak „'"; przy czym: etykiety te stanowią warunki tranyzcji nałożone poprzez operator iloczynu logicznego na warunki istniejące.

Parametr czasowy przypisany do tranzycji

Parametry czasowe związane z tranzycjami mogą być postrzegane jako czasy mijające od momentu usunięcia znaczników z miejsc wejściowych danych tranzycji, do momentu umieszczenia znaczników w ich miejscach wyjściowych (rys. 1.20).

Sytuację taką można zamodelować z wykorzystaniem mechanizmu przedstawionego w poprzednim podpunkcie. W tym celu wystarczy w miejsce tranzycji, do których przypisano parametry czasowe wprowadzić konstrukcje (t',p_t,t"), takie że t' etykietowana jest tak jak t, do miejsca p_t przypisany jest sygnał s1 startujący akcję czasową realizowana przez układ zewnętrzny, a tranzycja t" etykietowana jest sygnałem s1', sygnalizującym koniec akcji czasowej.



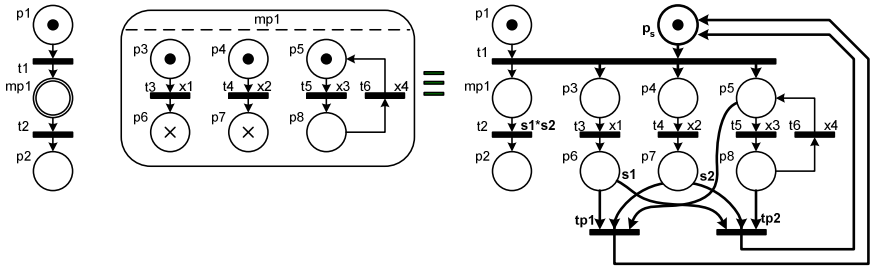
Rys. 1.20. Interpretacja parametru czasowego przypisanego do tranzycji

Makromiejsce bez historii

W sieciach HPN wprowadza się założenie, iż w momencie deaktywacji makromiejsca, wszystkie znaczniki wewnątrz jego rozwinięcia są usuwane. Jest to założenie

upraszczające zapis, ponieważ użytkownik nie musi zastanawiać się, w jaki sposób to osiągnąć. Jednakże implikuje to niestety niespełnienie własności bezpośredniego podstawienia podsieci w miejsce odpowiedniego makromiejsca.

Sprowadzając zapis do interpretowanej sieci płaskiej, należy wprowadzić jedno dodatkowe miejsce spoczynkowe p_s, oraz szereg tranzycji pomocniczych, których zadaniem jest właśnie usunięcie wszystkich znaczników z makromiejsca i aktywacja miejsca spoczynkowego. Ilustruje to rys. 1.21.

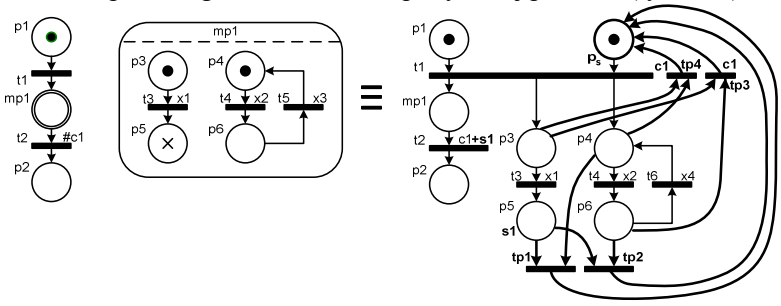


Rys. 1.21. Rozwinięcie hierarchii do sieci płaskiej

Biorąc pod uwagę fakt, iż warunkiem opuszczenia znacznika z makromiejsca jest osiągnięcie wszystkich miejsc końcowych wewnątrz podsieci związanej z tym makromiejscem, wówczas łatwo zauważyć, iż tranzycje pomocnicze będą potrzebne do łączenia wszystkich tych miejsc, które są równocześnie oznakowane wraz z miejscami końcowymi. Tranzycje pomocnicze muszą posiadać wyższy priorytet od pozostałych, w celu likwidacji ewentualnych niedeterminizmów. Miejsca końcowe można zaetykietować sygnałami wewnętrznymi, które są następnie użyte do warunkowania tranzycji wyjściowych odpowiedniego makromiejsca (na rys. 1.21 sygnały s1 oraz s2, aktywowane w miejscach p6 i p7, warunkują tranzycję t2).

Etykieta warunku silnego (wywłaszczenie)

Makromiejsce można pozbawić aktywności m.in. poprzez nałożenie na jego tranzycję wyjściową warunku silnego *abort* i spełnienie go. Oznacza to usunięcie wszystkich znaczników z odpowiedniej podsieci. Aby sytuację taką można było przedstawić z wykorzystaniem sieci płaskiej należy wprowadzić tyle tranzycji pomocniczych ile wierzchołków posiada graf znakowań rozpatrywanej podsieci (rys. 1.22).

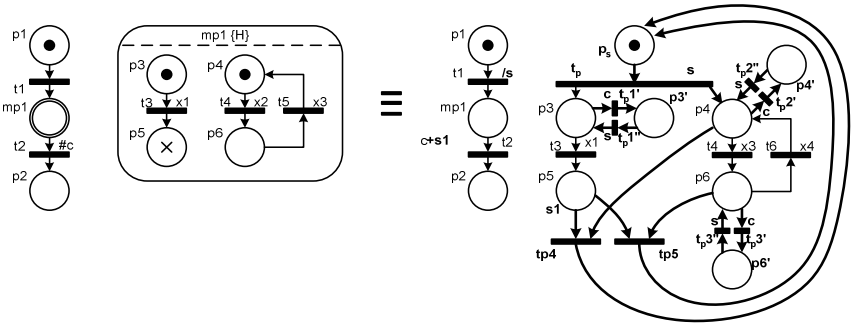


Rys. 1.22. Interpretacja warunku wywłaszczającego

Każdemu wierzchołkowi odpowiada odrębna tranzycja a jej miejscami wejściowymi są wszystkie miejsca należące do tego wierzchołka. Miejscem wyjściowym każdej tranzycji pomocniczej jest pomocnicze miejsce spoczynkowe p_s. Warunkiem zezwolenia tranzycji pomocniczych jest warunek wywłaszczający (*abort*), przy czym po osiągnięciu znakowania końcowego podsieć zachowuje się tak jak w pkt. poprzednim (bez wywłaszczenia). Warunek wywłaszczenia w sieci nadrzędnej dopisuje się poprzez operator sumy logicznej do pozostałych warunków zezwolenia tranzycji.

Makromiejsce z historią

W celu sprowadzenia do sieci płaskich makromiejsc z historią konieczne jest dodatkowe (w stosunku do sytuacji przedstawionej w poprzednim podpunkcie) wprowadzenie dla każdego miejsca podsieci dodatkowego miejsca spoczynkowego, do którego może zostać wprowadzony znacznik w momencie wywłaszczenia podsieci rys.1.23.



Rys. 1.23. Interpretacja parametru historii

Wyprowadzenie z niego znacznika do odpowiadającego mu miejscu głównemu, następuje po ponownej aktywacji makromiejsca w sieci nadrzędnej. Warto zauważyć, że tak jak w podpunkcie poprzednim, osiągnięcie znakowania końcowego, oznacza możliwość deaktywacji odpowiedniego makromiejsca, a jego ponowne aktywowanie spowoduje wprowadzenie znaczników do miejsc początkowych podsieci.

Użycie w jednym modelu wielu elementów opisu, w celu sprowadzenia do sieci płaskiej, wymaga superpozycji odpowiednich metod opisanych powyżej.

W rozdziale 1.3 pokazane zostało, w jaki sposób można sieć hierarchiczną sprowadzić do interpretowanej sieci płaskiej. Na podstawie tego nasuwa się wniosek, iż do analizy i weryfikacji formalnej sieci hierarchicznych można użyć znanych metod opracowanych dla sieci płaskich. Jednakże sama transformacja jest zadaniem bardzo skomplikowanym, a sieć wynikowa może mieć duże rozmiary, wobec czego wydaje się raczej koniecznym opracowanie metod alternatywnych analizy dla sieci hierarchicznych (Karatkievich i Andrzejewski 2002).

1.4. Platformy realizacyjne

Oprócz stosowanych modeli specyfikacji formalnej, w procesie projektowania cyfrowych układów sterowania, bardzo ważne jest również sprecyzowanie docelowej platformy realizacyjnej. Jest to istotny element, stanowiący o najważniejszych parametrach wdrażanych implementacji. W pracach (Balarin 1997, Jerraya *i in.* 1999) wskazuje się następujące możliwości rozwiązań praktycznych:

- sprzętowe,
- programowe,
- sprzętowo-programowe.

Pod pojęciem platformy sprzętowej należy rozumieć szeroko pojęte struktury układowe zarówno małej i średniej skali integracji (takie jak bramki przerzutniki czy rejestry) jak i nowoczesne matryce reprogramowalne (np. CPLD, FPGA). Realizacja programowa jest rozumiana jako połączenie pewnego zestawu instrukcji (program) oraz odpowiednich struktur sprzętowych (np. mikroprocesor, pamięć) zdolnych do wykonywania określonych działań zapisanych w kodzie tego programu. Sprecyzowanie to wydaje się niezbędne, ze względu na możliwość implementacji całych rdzeni mikrokontrolerów wraz z pamięciami programu i danych w nowoczesnych strukturach reprogramowalnych uważanych za platformę sprzętową, a dostępnych w postaci modułów IP (ang. *Intelectual Property*) (Gajski 2001, Staunstrup *i in.* 1997, Trung 2001).

Zalety i wady obu rozwiązań, tzn. realizacji sprzętowej oraz programowej, generalnie rozpatruje się pod względem dwóch podstawowych kryteriów:

- czas reakcji – ogólnie jest przyjęte uważać za szybsze rozwiązania sprzętowe,
- koszt prototypowania – koszty przyjmuje się niższe dla rozwiązań programowych, ze względu na niższą cenę zarówno samych układów, jak i narzędzi wspomagających proces projektowania.

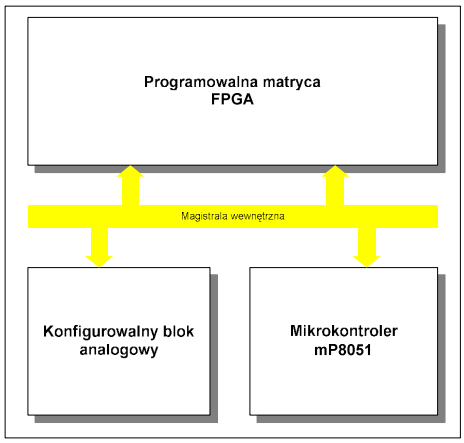
Przy czym własności te wydają się być wzajemnie sprzeczne. Alternatywą może być zastosowanie jednoczesnego połączenia obu metod, w którym ustala się pewien kompromis łączący zalety szybkości działania z niskimi kosztami produkcji. Połączenia takie mogą mieć miejsce w jednorodnych systemach od siebie oddalonych (systemy rozproszone), budowanych na bazie modułów projektowanych na wspólnej płycie bazowej (systemy osadzone) lub zintegrowanych w jednej strukturze scalonej (mikrosystemy cyfrowe). Ostatnie rozwiązanie jest w chwili obecnej najbardziej intensywnie rozwijane, ze względu na możliwość miniaturyzacji i oszczędność energii dorównujące rozwiązaniom jednorodnym.

1.5. Mikrosystemy cyfrowe

Spośród systemów realizacji sprzętowo-programowej wyodrębnić można grupę układów integrujących w jednej strukturze m.in. blok sprzętowy i rdzeń mikroprocesorowy. Jest to rodzina układów, o której pierwsze wzmianki w publikacjach naukowych zaistniały ok. połowy lat 90-tych. Od tego czasu istnieje silna tendencja rozwojowa tej grupy, a w tok prac włączyło się wiele znaczących firm, takich jak: ATMEL, CYPRESS czy TRISCEND.

Ze względu na brak precyzyjnego określenia pojęcia mikrosystemu cyfrowego w literaturze przedmiotu, zdecydowano się na próbę jego uściślenia na podstawie przeglądu istniejących rozwiązań konstrukcyjnych struktur objętych nazwą „mikrosystem cyfrowy" (ang. *digital microsystem*), wytwarzanych przez różnych producentów.

Jedną z pierwszych opracowanych struktur tego typu jest układ nazwany FIPSOC (ang. *Field Programmable System On Chip*) firmy SIDSA (Napieralski 1998). Schemat blokowy przedstawiony został na rys. 1.24.



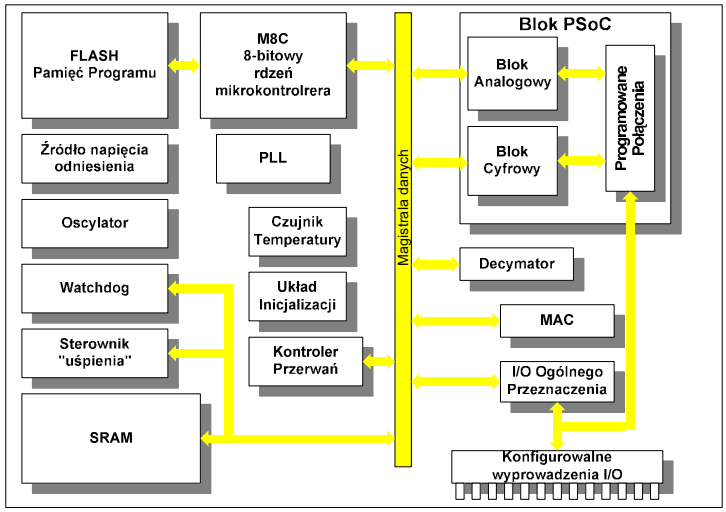
Rys. 1.24. Struktura ogólna układów FIPSOC

W układzie tym zintegrowano reprogramowalną matrycę FPGA, rdzeń mikroprocesora mP8051 (zgodny ze standardem Intel 8051) oraz konfigurowalny blok analogowy. W zależności od typu układu oferuje on matrycę od 8x16 do 16x16 programowalnych makrokomórek cyfrowych, odpowiadającą 20 tys. bramek logicznych. Rdzeń mikrokontrolera wyposażony jest w pamięć programu o rozmiarze od 4 do 16kB. Częstotliwość zegara taktującego procesor wynosi 40MHz. Konfigurowalny blok analogowy oferuje zestaw przetworników A/C i C/A wraz z układami próbkująco-pamiętającymi S/H oraz wzmacniaczami i filtrami dynamicznymi.

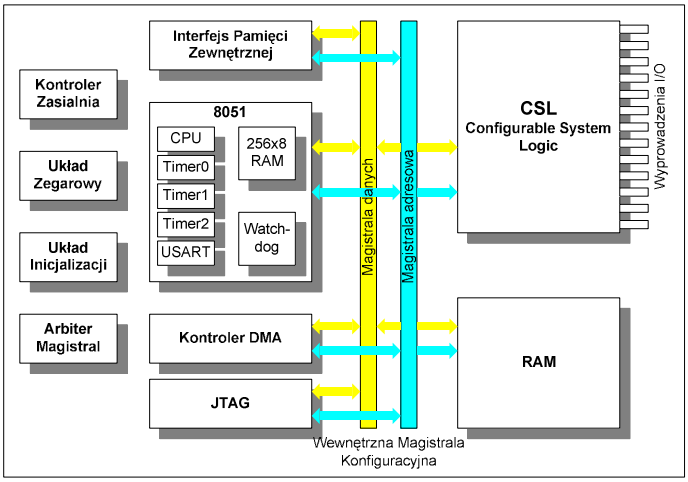
Kolejnym przykładem takiego systemu jest układ CY8C25xxx/26xxx firmy CYPRESS, gdzie oznaczenie xxx charakteryzuje jego konkretny typ (rys. 1.25). Zintegrowano w nim następujące bloki funkcjonalne:

- 8-bitowy mikroprocesor wykonany w architekturze typu harvard, taktowany z częstotliwością 24MHz,
- pamięć programu typu FLASH o rozmiarze od 4kB do 16kB (w zależności od układu),
- pamięć statyczną SRAM o rozmiarze 128B do 256B (w zależności od układu),
- programowalny blok analogowy (ADC, DAC, S/H, filtry, wzmacniacze, czujnik temperatury),

- układ programowania konfiguracji wyprowadzeń (przerzutniki Shmitta, dołączane rezystory, otwarty kolektor) 6-44 wyprowadzeń,
- precyzyjny, programowany układ zegarowy 48/24MHz (przy zastosowaniu zewnętrznego oscylatora 32,768kHz),
- układ czuwający watchdog, czujnik temperatury, źródło napięcia odniesienia,
- programowalny blok cyfrowy (liczniki, układy czasowe, UART's, generatory CRC, PWM's).



Rys. 1.25. Schemat blokowy układu CY8C25xxx/26xxx firmy CYPRESS

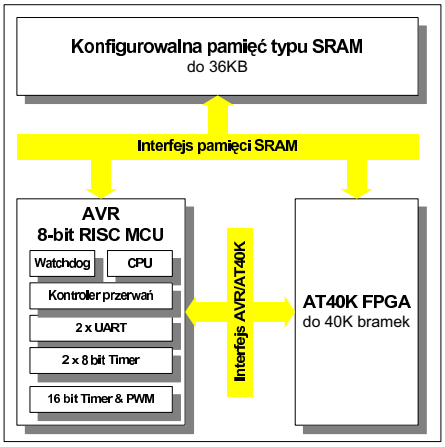


Rys. 1.26. Struktura układu TRISCEND E5 CSoC

Układ TRISCEND E5 CSoC powstał w roku 2001, i integruje w swej strukturze m.in. następujące elementy (rys. 1.26):

- mikrokontroler standardu 8051/52 taktowany zegarem 40MHz,
- do 64kB wewnętrznej pamięci danych RAM,
- programowalną matrycę CSL (ang. *Configurable System Logic*) do 3.200 cel (odpowiada ok. 40.000 bramek),
- kontroler bezpośredniego dostępu do pamięci DMA.

Ostatnim prezentowanym układem jest FPSLIC (rys. 1.27) firmy ATMEL. Struktura ta jest dostępna na rynku od roku 2001 i wciąż ulega modyfikacjom, co podkreśla jej tendencje rozwojowe.



Rys. 1.27. Struktura mikrosystemu cyfrowego AT40K firmy ATMEL

W strukturze zawarte są następujące bloki:

- konfigurowalna pamięć typu SRAM o pojemności do 36KB,
- ośmiobitowy procesor AVR, taktowany zegarem 25MHz,
- reprogramowalna matryca FPGA od 5 do 40 tys. bramek logicznych.

Zauważyć można w każdej z prezentowanych struktur inny, specyficzny zestaw bloków funkcjonalnych. Jednak wspólną cechą ich wszystkich jest posiadanie rdzenia mikroprocesorowego oraz programowalnej części sprzętowej. Dlatego też w dalszej części pracy pod pojęciem mikrosystemu cyfrowego będzie rozumiany taki układ scalony, który w swej strukturze integruje (obok innych) właśnie te dwa elementy.

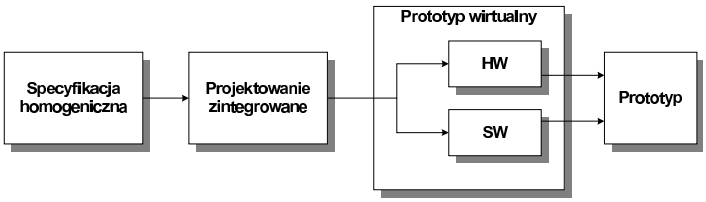
Definicja 1.36 *Mikrosystemem cyfrowym nazywa się taki układ scalony, który w swej strukturze integruje rdzeń mikroprocesorowy oraz programowalny blok sprzętowy.*

Mikrosystemy cyfrowe nabierają coraz większego znaczenia, co szczególnie jest widoczne wobec szeregu prac o tematyce łączącej możliwości matryc konfigurowalnych FPGA z elastycznością programową rdzeni mikroprocesorowych, prowadzonych w różnych ośrodkach akademickich (zwłaszcza krajowych) (Trung

2001). Nie do pominięcia jest również fakt, iż dzięki ich zastosowaniu można zmniejszyć rozmiary gabarytowe produktu, zużycie energii, a nawet koszty związane z produkcją i prototypowaniem, w porównaniu do rozwiązań bazujących na łączeniu osobnych struktur sprzętowych i mikroprocesorowych.

1.6. Projektowanie zintegrowane

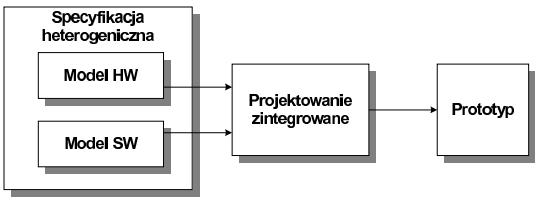
Ostatnim z omawianych elementów procesu projektowania jest jego metodologia. Z punktu widzenia prowadzonych prac badawczych, interesujące są te metodologie, które można wykorzystać do projektowania omawianych wcześniej mikrosystemów cyfrowych. W naturalny sposób zaadoptowane zostały do tego reguły tzw. projektowania zintegrowanego (ang. *Hardware/Software Co-design*), dotyczącego sprzętowo-programowej platformy realizacyjnej. Wcześniej projekt traktowano jako dwa odrębne moduły, podlegające niezależnym procesom analizy i syntezy. Zależności między nimi określane były głównie na poziomie specyfikacji i dotyczyły ustalenia interfejsu komunikacji międzymodułowej. Metodologia taka jest jednak bardzo pracochłonna i mało podatna na automatyzację. Alternatywę stanowi nowoczesne podejście, którego wyróżnikiem jest traktowanie projektu jako integralnej całości, podlegającej wspólnym procesom modelowania i weryfikacji.



Rys. 1.28. Homogeniczna metodologia projektowania zintegrowanego

W pracy (Jerraya *i in.* 1999) wskazuje się dwie odmiany projektowania zintegrowanego, różniące się ogólnie sposobem specyfikacji projektowanego systemu. W metodologii homogenicznej (rys. 1.28) istnieje jeden model specyfikacji formalnej, a podział na część sprzętową i programową następuje zwykle po wstępnym etapie weryfikacji.

W metodologii heterogenicznej (rys. 1.29) do specyfikacji używa się rozłącznych modeli dla obu części projektu, podlegających odpowiednim transformacjom w celu przeprowadzenia wspólnej weryfikacji i syntezy (koweryfikacja i kosynteza).



Rys. 1.29. Heterogeniczna metodologia projektowania zintegrowanego

Powstało wiele pakietów wspomagających projektowanie sprzętowo-programowe zarówno akademickich jak i komercyjnych. W tabelach 1.1 oraz 1.2 przedstawiono krótką charakterystykę najpopularniejszych pakietów, sporządzoną na podstawie przeglądu literatury przedmiotu (Hurk i Jess 1998, Jerraya *i in.* 1999, Staunstrup *i in.* 1997), oraz informacji udostępnionych na stronach internetowych.

Tab. 1.1. Wybrane pakiety akademickie

Pakiet	**Pochodzenie**	**Specyfikacja**
Castle	German National Research Center for Information Technology (GMD) at Sankt Augustin	C/C++, Verilog
Cobra	Universitet of Tübingen, Universidad Politecnica Madrid	VHDL
CodeSign	Computer Engineering Research Group of the Computer Engineering and Communication Networks Lab at the ETH Zurich	Object Petri Nets
Comet	Computer Engineering and Science of Case Western Reserve University	C, VHDL
Cosmos	TIMA, Grenoble University in France	EFSM,SDL
Cosyma	Technical University of Braunschweig	C*
CoWare	IMEC	C/C++
Polis	UC Berkeley	CFSM
ProCos	UK Engineering and Physical Sciences Research Council	C, VHDL
SpecSyn	UC Irvine	SLIF
Vulcan	Stanford	HardwareC

Tab.1.2. Wybrane pakiety komercyjne

Pakiet	**Pochodzenie**	**Narzędzia**	**Specyfikacja**
Cadence	Cadence	Alta Tools	C, VHDL
Esterel	INRIA, Antipolis, France	SyncCharts	Esterel
Mentor Graphics	Mentor Graphics	Leonardo, ModelSim	C, VHDL
Omniview	Omniview	Cosmos, Galaxy	EFSM, SDL
Synopsys	Synopsys	Eagle,COSSAP	C, VHDL
SystemDesig.ner	Atmel	Mentor Graphics	C, VHDL
Triscend FastChip	Triscend	Triscend, Keil	C, VHDL

Jak przedstawiono w tabelach 1.1 i 1.2, więcej niż 40% pakietów wspomagających, dla których wejściem jest specyfikacja heterogeniczna, wykorzystuje jako model wejściowy języki C oraz HDL, a ponad 60% bazuje na języku w standardzie C (lub zbliżonym). Opracowanie narzędzia, które umożliwiłoby automatyczne przygotowanie takiej specyfikacji na podstawie jednorodnego opisu (np. sieciami Petriego), wydaje się być zadaniem celowym, szczególnie wobec coraz większego zainteresowania hybrydowymi strukturami mikrosystemów cyfrowych.

W pracy podejmuje się próbę stworzenia modułu syntezy programowej, dla którego wejściem są interpretowane hierarchiczne sieci Petriego HPN, a wyjściem kompilowalny kod zgodny ze standardem języka ANSI C.

1.7. Podsumowanie i wnioski

W niniejszym rozdziale przedstawiono szeroko problematykę projektowania cyfrowych systemów sterowania. Pokazano szereg najpopularniejszych modeli specyfikacji formalnej, wskazując na interpretowane sieci Petriego, jako dogodny model pośredni, wykorzystywany w tworzonych na Uniwersytecie Zielonogórskim narzędziach wspomagających projektowanie. Na podstawie literatury przedmiotu rozpoznano istniejące rozszerzenia sieci Petriego, wspierające hierarchię oraz opis parametrów dynamicznych. Na tej podstawie wysunięto następujące wnioski:

- istniejące modele sieci Petriego nie wspierają jednocześnie w wystarczający sposób możliwości opisu hierarchicznego i opisu zależności czasowych,
- większość modeli hierarchicznych buduje hierarchię strukturalną, w której zamiast wyróżnionych makrowęzłów można bezpośrednio podstawić skojarzoną z nimi podsieć, podczas gdy aktualnym trendem jest kompozycja hierarchii behawioralnej (patrz rozdz. 1.3).

W związku z powyższym zdecydowano się na opracowanie autorskiego modelu interpretowanych sieci hierarchicznych, pozbawionych wzmiankowanych wad, a jednocześnie zachowującego pewną spójność ze standardem technologii UML (autor ma na myśli diagramy Statecharts).

W rozdziale tym opisano również platformy realizacji praktycznej sterowników cyfrowych oraz istniejące metodologie projektowania w kontekście pojawienia się w produkcji nowoczesnych mikrosystemów cyfrowych, integrujących w swych strukturach zarówno blok sprzętowy jak i programowy.

Rozdział 2

METODY PROGRAMOWEJ REALIZACJI CYFROWYCH UKŁADÓW STEROWANIA

Z punktu widzenia niniejszej pracy najbardziej interesujące są metody programowej realizacji układów sterowania binarnego układów danych w postaci interpretowanych sieci Petriego. Dlatego też większa część rozdziału poświęcona zostanie przeglądowi istniejących metod implementacji sieci. Jednakże dla podkreślenia istotności podjęcia tematyki oraz jej aktualności, zdecydowano się także na charakterystykę implementacji innych modeli wykorzystywanych w nowoczesnych systemach wspomagających projektowanie systemów sterownia.

2.1. Przegląd istniejących metod realizacji programowej sieci Petriego

Istniejące metody realizacji programowej sieci Petriego można ogólnie sklasyfikować wg ich rozwiązań architektonicznych:

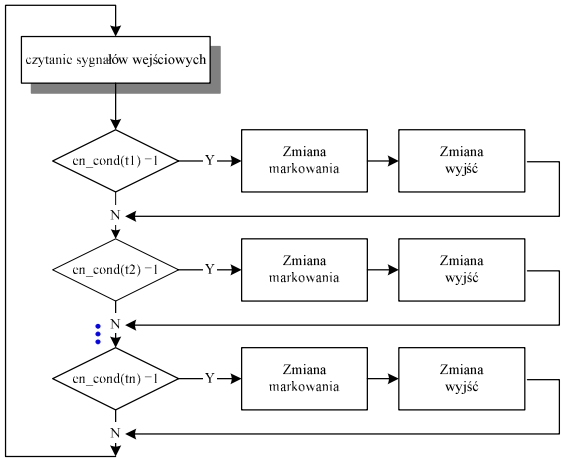
a) jednorodna:
- sekwencyjna,
- dynamiczna;

b) z podziałem na część sterującą i blok danych:
- z wykorzystaniem grafu znakowań,
- z wykorzystaniem tablic behawioralnych,
- z wykorzystaniem macierzy incydencji,
- z wykorzystaniem systemu wnioskującego.

2.1.1. Jednorodna realizacja sekwencyjna

Przykład monolitycznego programu sekwencyjnego, realizującego interpretowaną sieć Petriego zaprezentowany został w pracy (Misiurewicz 1980). Schemat blokowy programu przedstawiono na rys. 2.1.

Algorytm działania bazuje na sekwencyjnym sprawdzaniu warunków zezwolenia każej tranzycji, a następnie (o ile warunki zosatły spełnione), na wykonaniu procedur zmiany zankowania i zmiany stanu wyjść.

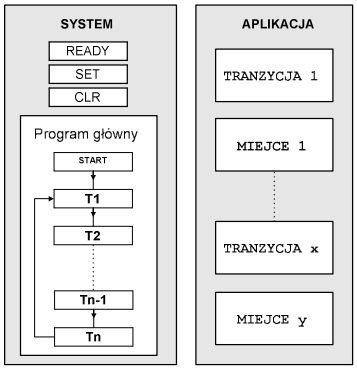


Rys. 2.1. Sekwencyjna realizacja sieci Petriego

Metodę tę cechuje prostota rozwiązania, jednak może ona prowadzić do nadmiernego zwiększenia objętości programu w sytuacji, gdy sieć będzie bardziej skomplikowana. Wynika to z powielania funkcji zmiany znakowania i zmiany wyjść w każdej gałęzi programu. Dodatkowo przy większej liczbie tranzycji czasy reakcji systemu na zmiany sygnałów wejściowych mogą się znacząco wydłużyć.

2.1.2. Jednorodna realizacja dynamiczna

System funkcji dynamicznych przedstawiono w pracy (Ciesłowki i Andrzejewski 2001). Podstawą opisywanej metody jest odwzorowanie aktualnego stanu sieci Petriego za pomocą tablicy T(1...n), w której przechowywane są informacje o tranzycjach pobudzonych (tzn. takich, których co najmniej jedno miejsce wejściowe jest aktywne) (rys. 2.2).



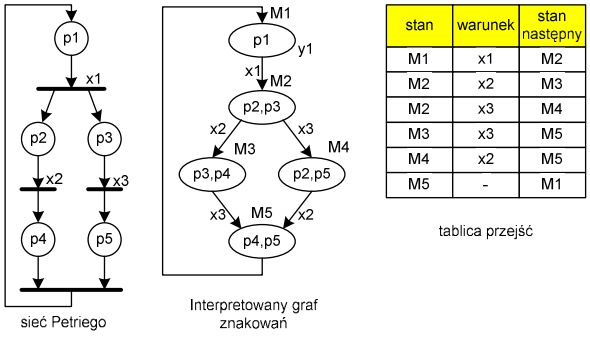
Rys. 2.2. System funkcji dynamicznych

Program aplikacyjny składa się z funkcji reprezentujących miejsca i tranzycje sieci Petriego oraz pewnego zestawu funkcji wspólnych (READY, SET oraz CLR). Próbie uaktywnienia podlegają tylko tranzycje z tablicy T. Po aktywacji wywoływane są funkcje: tranzycji i miejsc (wejściowych i wyjściowych), które modyfikują zawartość tablicy oraz sterują sygnałami wyjściowymi.

Realizacja taka jest efektywna pod względem ograniczenia czasów reakcji systemu, jednak pochłania znaczne zasoby pamięciowe (każdemu węzłowi odpowiada osobna funkcja).

2.1.3. Realizacja z wykorzystaniem grafu znakowań

W metodach realizacyjnych z wydzieloną częścią sterującą i blokiem danych przechowujących informacje o specyfikacji i stanie systemu zauważyć można trzy podejścia. Pierwsze sprowadza sieć Petriego do zwykłego automatu bez współbieżności, poprzez wyznaczenie odpowiedniego grafu znakowań, a następnie informacje o zachowaniu takiego automatu przedstawiane są za pomocą odpowiednich tablic, na których wykonuje działania część operacyjna systemu (rys. 2.3). Przykład takiego podejścia przedstawiono w pracach (Baranowski 1981, Andrzejewski 2000a). Jej podstawową wadą jest nadmierny rozrost objętości bloku danych (a zarazem zwiększenie czasu jego przeszukiwania, co wprost przekłada się na czasy reakcji systemu) spowodowany problemem „eksplozji stanów", w sytuacji gdy realizowana sieć ma duży współczynnik współbieżności. Z tego względu autorzy zalecają realizację sieci o współczynniku współbieżności raczej nie przekraczającym 4.



stan	warunek	stan następny
M1	x1	M2
M2	x2	M3
M2	x3	M4
M3	x3	M5
M4	x2	M5
M5	-	M1

tablica przejść

Rys. 2.3. Metoda realizacji wykorzystująca graf znakowań

2.1.4. Realizacja z wykorzystaniem tablic behawioralnych

Rozwiązanie podobne pod względem strukturalnym, ale operujące nie na grafie znakowań lecz bezpośrednio na sieci Petriego, przedstawiono w pracy (Sami i Courvoisier 1981). Program realizacyjny wykorzystuje szereg tablic (tablica specyfikacji tranzycji, tablica specyfikacji miejsc, tablica warunków – wyrażeń logicznych), w których zawarta jest informacja zarówno o działaniu systemu jak i o jego aktualnym stanie. Realizacja tej metody pozbawiona jest wady wynikającej

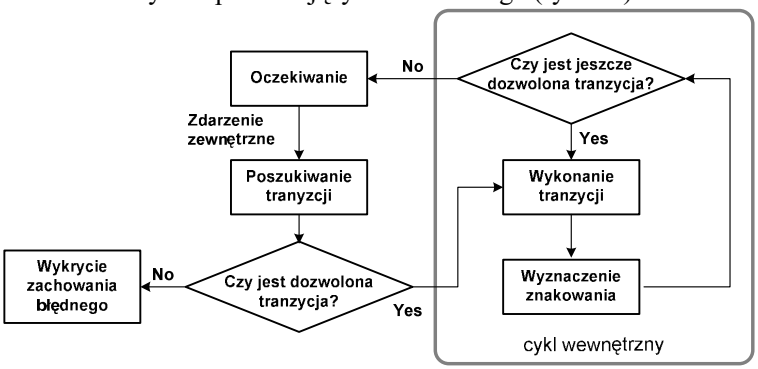
z „eksplozji stanów”, jednak niski poziom realizacyjny (asembler) oraz struktura tablic zorientowana na bitową reprezentację zachowania sieci w danych jednobajtowych ograniczają jej stosowanie jedynie do mikroprocesorów 8-bitowych.

2.1.5. Realizacja z wykorzystaniem macierzy incydencji

Kolejną prezentowaną metodologią jest wykorzystanie do specyfikacji sieci tablic bazujących na macierzach incydencji (Silva i David 1979, Silva i Velilla 1982). Program sterujący na ich podstawie i na podstawie znakowania aktualnego dokonuje obliczenia znakowania następnego. Pomimo braku przedstawienia przez autorów ograniczeń tej metody, wydaje się, iż jest ona przydatna jedynie dla małych sieci, ze względu na ograniczenia rozmiaru macierzy oraz czasu potrzebnego na analityczne obliczanie kolejnych znakowań.

2.1.6. Realizacja Token Player

W pracach (Silva i Valette 1990, Valette 1995) przedstawiono realizację sieci Petriego, wykorzystującą system wnioskujący (nazywany przez autora Token Player), operujący na strukturach danych reprezentujących sieć Petriego (rys. 2.4).



Rys. 2.4. Metoda token player

Głównym zadaniem systemu jest zmiana stanu systemu w odpowiedzi na zdarzenie, mające miejsce w systemie sterowania. Stan „Oczekiwanie” jest stanem stabilnym. W momencie odebrania przez system wnioskujący komunikatu (zdarzenie zewnętrzne), poszukuje on tranzycji związanej z tym zdarzeniem. Jeśli taka tranzycja zostanie odnaleziona, to zostanie ona wykonana, wyznaczone zostanie nowe znakowanie, a następnie w pętli wykonane zostaną wszystkie kolejno zezwalane tranzycje. Jeśli natomiast w odpowiedzi na zdarzenie zewnętrzne system nie odnajdzie dozwolonej tranzycji, to generuje on wówczas komunikat o błędzie („Wykrycie zachowania błędnego). Do rozwiązania takiego nasuwają się następujące komentarze:

- musi istnieć zewnętrzny blok kształtujący komunikaty o zdarzeniach zewnętrznych, tak by wywołanie takiego komunikatu odpowiadało sytuacji zezwolenia konkretnej tranzycji, np. jeśli tranzycja jest warunkowana funkcją logiczną $x1 + \overline{x2}$, to układ ten generuje odpowiedni komunikat dopiero w

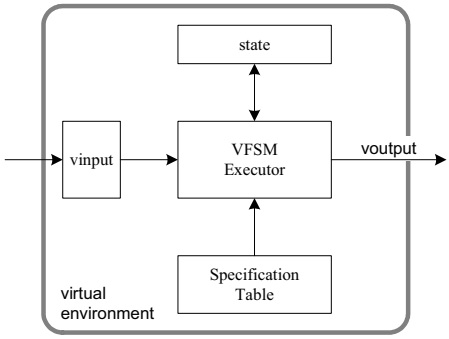
sytuacji, gdy x1=1 i x2=0, a co za tym idzie – musi on w sposób stały kontrolować stan sygnałów zewnętrznych;
- dla zapewnienia jednoznaczności działania systemu, z każdym zdarzeniem musi być związana inna tranzycja (co wymaga dodatkowego mechanizmu dla sytuacji, w których klika tranzycji warunkowanych jest tym samym podzbiorem przestrzeni wejścia);
- ze względu na istnienie pętli wewnętrznej, w której wykonywane są kolejno zezwalane tranzycje, jest bardzo trudny do określenia czas cyklu wnioskowania (bez szczegółowej analizy działania sieci).

2.2. Inne realizacje programowe systemów sterowania binarnego

W punkcie tym skupiono się na pobieżnej charakterystyce metod realizacji programowej innych modeli formalnych, stosowanych w nowoczesnych systemach wspomagania projektowania. W szczególności skoncentrowano się na systemie decyzyjnym realizacji wirtualnego automatu cyfrowego, ze względu na podobieństwo metody z prezentowaną w pracy.

2.2.1. Realizacja z wykorzystaniem wirtualnego systemu decyzyjnego

Wirtualny automat cyfrowy został przedstawiony w pracy (Wagner 1994). Definiowany jest on w programowym środowisku decyzyjnym nazywanym środowiskiem wirtualnym (rys. 2.5). Tworzone ono jest przez zbiory nazw sygnałów wejściowych, wyjściowych oraz nazw stanów. VFSM definiowany jest przez następujące elementy: tablicę specyfikacji (specification table), aktualny stan systemu (state) oraz zbiór nazw aktywnych sygnałów wejściowych (vinput).



Rys. 2.5. Środowisko definiujące VFSM

Blok decyzyjny (VFSM Executor) jest maszyną decyzyjną, która na podstawie informacji zapisanych w tablicy specyfikacji, aktualnego zawartości bloku vinput podejmuje następujące decyzje:
- jaką akcję wyjściową wykonać,
- na jaki zmienić stan aktualny systemu.

Voutput jest zbiorem nazw sygnałów wyjściowych, jednak nie funkcjonuje jako zmienna systemowa, tylko służy do natychmiastowego wyzwolenia akcji wyjściowych, związanych ze zmianą stanu sygnałów wyjściowych.

Tak przedstawione środowisko realizowane jest jako program złożony z odpowiednich konstrukcji językowych (zgodnych ze standardem ANSI C): zmiennych, struktur oraz zespołu funkcji realizujących zadania bloku decyzyjnego. Model ten został wykorzystany w komercyjnym pakiecie wspomagającym projektowanie VIS firmy TCI (Gesellschaft für technische Informatik mbH). W chwili obecnej jest przedmiotem badań nad opracowaniem metod weryfikacji formalnej w projekcie „Model Checking and Program Analysis" prowadzonym przez firmę ©Lucent Technologies.

2.2.2. Realizacja z wydzielonym systemem operacyjnym czasu rzeczywistego

System operacyjny definiuje się jako program główny systemu mikroprocesorowego, zarządzający m.in.: pamięcią, wykonywanymi zadaniami, oraz komunikacją z urządzeniami zewnętrznymi. Jego główna część, nazywana kernelem, jest zawsze aktywna. Wszystkie programy uruchamiane w danym systemie procesorowym muszą współpracować z jego systemem operacyjnym. Systemem operacyjnym czasu rzeczywistego (ang. *Real Time Operating System*), nazywa się taki system, który pracuje w środowisku czasu rzeczywistego (Silberschatz *i in.* 2001).

Rozwiązania bazujące na systemach RTOS, traktują wydzielone fragmenty układu sterowania jako zadania, zarządzane przez RTOS. Najczęściej sytuacja ta dotyczy realizacji systemów sterowania współbieżnego. Przykładami pakietów wspomagających taką metodologię syntezy programowej są m.in. POLIS oraz COSYMA (Balarin 1997).

Wydaje się jednak bardzo trudne przygotowanie takiego systemu RTOS, który byłby w stanie poprawnie pracować niezależnie od procesora docelowego, a tym bardziej opracowanie programu automatycznej generacji takiego systemu dla dowolnego procesora. Z tego tez też względu rozwiązania takie bazują na ściśle określonych typach lub rodzinach procesorów (np. Motorola 68HC11 w pakiecie POLIS), a ewentualna ekspansja na dowolne systemy jest praktycznie niemożliwa.

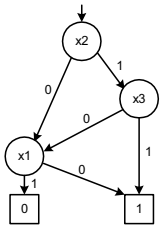
2.2.3. Binarne diagramy decyzyjne

W najogólniejszym przypadku binarny diagram decyzyjny (ang. *Binary Decision Diagram*) jest acyklicznym grafem skierowanym reprezentującym funkcję boolowską (Minato 1996). Posiada on dwa węzły końcowe, nazywane umownie: 0-terminal oraz 1-terminal. Każdy węzeł nieterminalny jest indeksowany zmienną wejściową funkcji logicznej i posiada dwa łuki wyjściowe, nazywane: 0-edge oraz 1-edge. Węzły reprezentują rozwinięcie Shanona funkcji logicznej, takie że:

$$f = \bar{x}_i \cdot f_0 \vee x_i \cdot f_1 \qquad (2.1)$$

gdzie: x_i jest indeksem węzła, a f_0 i f_1 są funkcjami węzłów wskazywanych odpowiednio łukami 0-edge oraz 1-edge.

Przykład takiego diagramu reprezentującego funkcję logiczną postaci $f = x2 \cdot x3 \vee \overline{x1}$ przedstawiono na rys. 2.6.



Rys. 2.6. Przykład binarnego diagramu decyzyjnego

Za pomocą diagramów decyzyjnych można przedstawiać systemy sterowania opisywalne funkcjami logicznymi (Biliński 1996, Kozłowski 1996). Jednakże ich zastosowanie do projektowania takich systemów ogranicza się raczej do weryfikacji formalnej oraz optymalizacji modeli, niż do syntezy programowej, choć nie wyklucza się stosowania BDD do realizacji wykorzystujących komputery klasy PC. Wynika to głównie z faktu, iż diagramy pochłaniają stosunkowo dużo pamięci do ich reprezentacji w systemie procesorowym (struktura list dynamicznych, pakiet obsługi diagramów), co w przypadku mikrosystemów cyfrowych z ograniczoną pamięcią stanowi podstawowy problem.

2.3. Podsumowanie i wnioski

W rozdziale 2 przedstawiono znane metody realizacji programowej układów sterowania cyfrowego danych w postaci interpretowanych sieci Petriego oraz najważniejsze przykłady implementacji innych modeli, ze szczególnym uwzględnieniem wirtualnego automatu skończonego VFSM. Na podstawie analizy wysunąć można następujące wnioski i spostrzeżenia:

- zdecydowana większość istniejących metod realizacyjnych interpretowanych sieci Petriego powstała stosunkowo dawno, dla jednych z pierwszych opracowanych struktur systemów mikroprocesorowych (Intel 8080), z wykorzystaniem języka niskiego poziomu – asemblera,
- zastosowanie ich w nowoczesnych strukturach wymaga szerokich zmian o charakterze metodycznym (Adamski 2001),
- istnieją metody rokujące możliwość ich adaptacji dla potrzeb realizacji sieci hierarchicznych HPN (TokenPlayer oraz VFSM), jednak wymagają one szeregu rozszerzeń strukturalnych,
- nie spotkano się w literaturze z próbą sformalizowania reprezentacji programowej interpretowanych sieci Petriego (próba taka została podjęta w pracy (Wagner 1994) dla modelu automatu skończonego VFSM).

Stąd też zdecydowano się na opracowanie metodologii, wywodzącej się niejako z obu wcześniej wzmiankowanych metod TokenPlayer oraz VFSM, oraz znacznie je rozszerzającej. Koncepcja ta szczegółowo została opisana w rozdziale 3.

Rozdział 3

PROGRAMOWY MODEL INTERPRETOWANYCH SIECI PETRIEGO

W ogólności przyjęto zasadę opracowania stałego szkieletu programu, wykonującego określone działania zgodnie z informacjami zapisanymi w zmiennej strukturze specyfikacji. Doprowadziło to do powstania programowego systemu decyzyjnego VDS (ang. *Virtual Decision System*), wykorzystującego teoriomnogościową reprezentację sieci, nazywaną dalej programowym modelem sieci Petriego.

3.1. Programowy model hierarchicznej sieci Petriego

Niech będzie dana sieć hierarchiczna HPN, złożona z m wyróżnionych podsieci.

Definicja 3.1 *Zbiorem podsieci Z nazywamy zbiór złożony ze wszystkich wyróżnionych podsieci sieci HPN.*

Niech K będzie uporządkowanym zbiorem złożonym z liczb naturalnych, takim że zawiera on następujące elementy K={1,2,...,k}, przy czym k=m. Można utworzyć różnowartościową funkcję ρ, taką że będzie ona jednoznacznie przypisywać każdemu elementowi zbioru K dokładnie jeden element zbioru Z, a więc: ρ: K→Z. W ten sposób każda wyróżniona podsieć może być jednoznacznie określona poprzez liczbę należącą do zbioru K. Zapis Z^k oznaczać będzie podsieć indeksowaną przez k.

Definicja 3.2 *Zbiorem tranzycji podsieci T^k nazywamy zbiór złożony ze wszystkich tranzycji należących do wyróżnionej podsieci Z^k.*

Definicja 3.3 *Zbiorem miejsc podsieci P^k nazywamy zbiór złożony ze wszystkich miejsc należących do wyróżnionej podsieci Z^k.*

Definicja 3.4 *Programowym modelem sieci hierarchicznej nazywa się piątkę uporządkowaną, taką że:*

$$PHPN = (P, T, S, C, Ŧ) \qquad (3.1)$$

gdzie: P – jest skończonym niepustym zbiorem nazw miejsc sieci; T – jest skończonym niepustym zbiorem nazw tranzycji sieci; S – jest skończonym niepustym zbiorem

nazw sygnałów, takim że: S=X∪Y∪L; C – jest strukturą specyfikacji; Ŧ – jest dyskretną skalą czasu.

Struktura specyfikacji jest indeksowaną rodziną zbiorów, taką że zawiera ona *k* podzbiorów reprezentujących wszystkie wyróżnione podsieci (łącznie z siecią podstawową). Każdy podzbiór jest indeksowany numerem odpowiedniej podsieci (zgodnie z zasadami określonymi w def. **3.1**) i oznacza się go jako C^k. Złożony jest on z czternastu składowych, takich że:

C^k_1 – jest uporządkowaną rodziną zbiorów nazw miejsc wejściowych wszystkich tranzycji ze zbioru T^k, taką że $C^k_1(t)$ odpowiada zbiorowi miejsc wejściowych połączonych łukami zwykłymi z tranzycją *t*,

C^k_2 – jest uporządkowaną rodziną zbiorów nazw miejsc zezwalających wszystkich tranzycji ze zbioru T^k, taką że $C^k_2(t)$ odpowiada zbiorowi miejsc połączonych łukami zezwalającymi z tranzycją *t*,

C^k_3 – jest uporządkowaną rodziną zbiorów nazw miejsc zabraniających wszystkich tranzycji ze zbioru T^k, taką że $C^k_3(t)$ odpowiada zbiorowi miejsc połączonych łukami zabraniającymi z tranzycją *t*,

C^k_4 – jest uporządkowanym zbiorem etykiet warunków zwykłych postaci cond, przyporządkowanych do wszystkich tranzycji ze zbioru T^k, takim że $C^k_4(t)$ odpowiada warunkowi zwykłemu związanemu z tranzycją *t*,

C^k_5 – jest uporządkowanym zbiorem etykiet warunków wywłaszczających postaci abort, przyporządkowanych do wszystkich tranzycji ze zbioru T^k, takim że $C^k_5(t)$ odpowiada warunkowi wywłaszczającemu związanemu z tranzycją *t*,

C^k_6 – jest uporządkowanym zbiorem liczb z dyskretnej skali czasu, przyporządkowanych do wszystkich tranzycji ze zbioru T^k, takim że $C^k_6(t)$ odpowiada wartości funkcji czasu $\tau(t)$, gdzie t jest tranzycją *t*,

C^k_7 – jest uporządkowaną rodziną zbiorów nazw miejsc wyjściowych wszystkich tranzycji ze zbioru T^k, taką że $C^k_7(t)$ odpowiada zbiorowi miejsc wyjściowych połączonych łukami zwykłymi z tranzycją *t*,

C^k_8 – jest uporządkowanym zbiorem etykiet postaci action, przyporządkowanych do wszystkich tranzycji ze zbioru T^k, takim że $C^k_8(t)$ odpowiada etykiecie action związanej z tranzycją *t*,

C^k_9 – jest uporządkowanym zbiorem etykiet postaci action, przyporządkowanych do wszystkich miejsc ze zbioru P^k, takim że $C^k_9(p)$ odpowiada etykiecie action związanej z miejscem *p*,

C^k_{10}– jest uporządkowanym zbiorem liczb z dyskretnej skali czasu, przyporządkowanych do wszystkich miejsc ze zbioru P^k, takim że $C^k_{10}(p)$ odpowiada wartości funkcji czasu $\tau(p)$, określonej dla *p*,

C^k_{11}– jest uporządkowanym zbiorem trójek $(\sigma_1,\sigma_2,\sigma_3)$, dookreślającym miejsca w podsieci Z^k, takim że $C^k_{11}(p)$ określa następujące własności miejsca *p*:

- σ_1 – jest zbiorem jednoelementowym (przybierającym wartości {0,1}), określającym typ miejsca, przy czym 0 oznacza miejsce zwykłe, a 1 makromiejsce,
- σ_2 – jest zbiorem jednoelementowym (przybierającym wartości {0,1,2}), określającym rodzaj miejsca, przy czym 0 oznacza miejsce, które nie jest ani początkowe, ani końcowe, 1 – miejsce początkowe, 2 – miejsce końcowe,

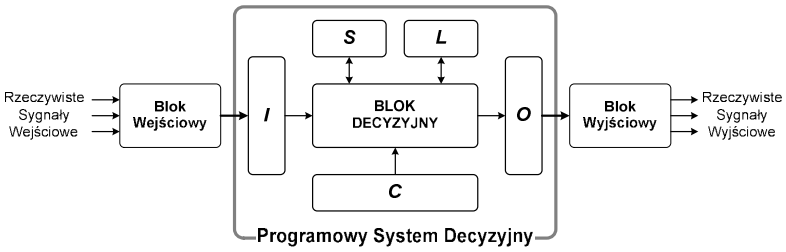
σ_3 – jest zbiorem jednoelementowym (przybierającym wartości ze zbioru K}), określającym indeks wyróżnionej podsieci związanej z miejscem funkcją hierarchii, przy czym jeśli miejsce jest zwykłym to σ_3 jest zbiorem pustym.

C^k_{12}– jednoelementowy zbiór (przybierający wartości {0,1}), określający atrybut historii podsieci o indeksie *k*.

Tak zdefiniowany model jest modelem ogólnym, nie wyróżniającym żadnej podklasy sieci. W ogólności może on ulegać modyfikacjom na rzecz uproszczenia realizacji o elementy nie występujące w opisie. Jest on zarazem wystarczający, aby można było w sposób jednoznaczny odtworzyć z niego sieć w zapisie klasycznym (np. graficznie).

3.2. Programowy system decyzyjny

Jednym z możliwych sposobów realizacji programowej automatycznego układu sterowania danego siecią Petriego jest zdefiniowanie abstrakcyjnego środowiska poprzez nazwy miejsc, nazwy sygnałów (wejściowych, wyjściowych i lokalnych) oraz programowy system decyzyjny VDS (rys. 3.1).



Rys. 3.1. Programowy System Decyzyjny – VDS

W jego skład wchodzą następujące bloki funkcjonalne: ***S*** – blok przechowujący informacje o stanie systemu, ***C*** – blok specyfikacji, ***I***, ***O***, ***L*** – bloki przechowujące informacje o stanie sygnałów odpowiednio wejściowych, wyjściowych, oraz lokalnych. Przedstawiony układ na podstawie przesłanek wynikających z zawartości bloków ***I***, ***L***, ***S*** oraz o informacji zawartych w bloku specyfikacji ***C***, podejmuje decyzję o przeprowadzeniu systemu do stanu kolejnego, bądź też pozostawieniu w stanie poprzednim.

Bloki zewnętrzne wejściowy i wyjściowy mają za zadanie pozyskiwanie informacji o stanie zewnętrznych sygnałów wejściowych oraz sterowanie sygnałami wyjściowymi (zgodnie z zawartością bloku ***O***). Ich wykluczenie z systemu decyzyjnego podyktowane jest względami oczywistymi, ponieważ uniezależnia to przyjęte rozwiązanie od konkretnej realizacji praktycznej.

Blok stanu systemu ***S*** złożony jest z indeksowanej rodziny zbiorów S, takiej że zawiera ona k podzbiorów reprezentujących wszystkich wyróżnione podsieci (łącznie z siecią podstawową) oraz zbioru STS, przechowującego nazwy sygnałów aktywnych tylko w następnym cyklu decyzyjnym. Określony podzbiór zbioru S jest indeksowany

numerem odpowiedniej podsieci i oznacza się go jako S^k. Każdy taki podzbiór S^k złożony jest z następujących składowych:

S^k_1 – zbiór nazw tych miejsc sieci, które w danym momencie czasu są aktywne (tzn. w ujęciu klasycznym posiadają znacznik), a ich funkcja czasu zwraca wartość 0: $\tau(p)=0$,

S^k_2 – zbiór par uporządkowanych $(p,\text{ŧ})$, takich że p jest nazwą aktywnego miejsca, dla którego $\tau(p)>0$, a ŧ liczbą z dyskretnej skali czasu określającą stan akcji czasowej związanej z miejscem p,

S^k_3 – zbiór par uporządkowanych $(t,\text{ŧ})$, takich że t jest aktywną tranzycją (tzn. w stanie wykonywania), a ŧ liczbą z dyskretnej skali czasu określającą stan akcji czasowej związanej z tranzycją t.

Struktura taka jest wystarczająca do przedstawienia aktualnego stanu sieci oraz prawidłowego przebiegu procedur podejmowania decyzji przez system decyzyjny VDS. Jednakże (szczególnie przy sieciach, w których liczba makromiejsc jest stosunkowo niewielka), wydaje się uzasadnione wprowadzenie pewnego nadmiaru informacji w bloku S, ze względu na korzyści wynikające z uproszczenia funkcji decyzyjnych. Mowa tu o wprowadzeniu statusu określającego aktywność (bądź nieaktywność) danego makra:

S^k_4 – jednoelementowy zbiór (mogący przybierać wartości {false,true}), przechowujący informacje o aktywności podsieci o indeksie k.

3.3. Zasady działania modelu programowego sieci hierarchicznej

Dla czytelniejszego przedstawienia zasad działania sieci, można wprowadzić dodatkowe definicje analogicznie jak w rozdziale 1.3.3, określające zbiory miejsc końcowych.

Niech p będzie miejscem należącym do pewnej podsieci Z^k. Jeśli $C^k_{11,\sigma_1}(p)=1$ czyli p jest makromiejscem, to $k'=C^k_{11,\sigma_3}(p)$, gdzie k' jest indeksem podsieci skojarzonej z makromiejscem p. Przez analogię k" będzie oznaczać indeks podsieci skojarzonej z miejscem p'., przy czym $p'\in Z^{k'}$, a $p''\in Z^{k''}$.

Definicja 3.5 *Zbiorem miejsc końcowych w podsieci skojarzonej z makromiejscem p jest taki zbiór* P^{end}_p *, że:*

$$\forall_{p'\in Z^{k'}} C^{k'}_{11,\sigma_2}(p')=2 \Rightarrow p'\in P^{end}_p \qquad (3.2)$$

Uwaga: Jest to definicja analogiczna do definicji 1.24.

Co oznacza, że do zbioru miejsc końcowych należy każde miejsce z podsieci $Z^{k'}$, takie że posiada ono atrybut miejsca końcowego.

Definicja 1.6 *Funkcją zbioru miejsc końcowych nazywamy taką funkcję* $\xi: P\rightarrow P$*, która dla makromiejsca p zwraca jego zbiór miejsc końcowych:*

$$\xi(p)=P^{end}_p \qquad (3.3)$$

wyrażenie ξ^* oznacza przechodnie zwrotne dopełnienie funkcji ξ, takie że dla każdego makromiejsca $p \in P$, zachodzą następujące warunki:

$p \in \xi^*(p)$ *(3.4)*

$\xi(p) \in \xi^*(p)$ *(3.5)*

$p' \in \xi^*(p) \Rightarrow \xi(p') \subseteq \xi^*(p)$ *(3.6)*

Uwaga: Jest to definicja analogiczna do definicji 1.25.

Działanie sieci wyznaczane jest poprzez podejmowanie decyzji o zmianie stanu systemu. Zasady podejmowania decyzji określa zbiór warunków, odpowiadający warunkom przygotowania tranzycji oraz akcje związane z jej wykonaniem.

Warunki przygotowania tranzycji t w podsieci o numerze *k*:

$S_4^k = \text{true}$ *(3.7)*

$$\bigvee_{p \in \{C_1^k(t) \cup C_2^k(t)\}} p \in S_1^k \cup S_2^k \qquad (3.8)$$

$$\bigvee_{p \in C_3^k(t)} p \notin S_1^k \cup S_2^k \qquad (3.9)$$

$$C_4^k(t) = \text{true} \qquad (3.10)$$

$$\bigvee_{p \in C_1^k(t)} C_{11,\sigma_1}^k(p) = 1 \Rightarrow \bigvee_{p' \in \xi^*(p)} p' \in S_1^{k'} \text{ lub } ((p', ŧ) \in S_2^{k'} \text{ i } ŧ = 0) \qquad (3.11)$$

$$\bigvee_{p \in C_1^k(t)} (p, ŧ) \in S_2^k \Rightarrow ŧ = 0 \qquad (3.12)$$

$$C_5^k(t) = \text{true} \qquad (3.13)$$

Zapis $C^k{}_4(t)$=true oraz $C^k{}_5(t)$=true jest skróconym zapisem, takim że odpowiednia formuła logiczna (*cond* lub *abort*) zwraca wartość logiczną true.

Uwaga: Analogicznie do punktu 1.3.3, ze wszystkich warunków można złożyć warunek ogólny postaci: 3.7*3.8*3.9*(3.10*3.11*3.12+3.13).

Niech $P^H(p)$ będzie zbiorem wszystkich miejsc, które należą do jakiejkolwiek podsieci wyprowadzonej od makromiejsca p w całej strukturze hierarchii, tzn. niech zbiór $P^H(p)$ odpowiada zbiorowi $\chi^*(p)$.

Niech ζ_1 zastępuje zapis $- S_1^{k''} := \{p'' \} : C_{11,\sigma_2}^{k''}(p'') = 1 \wedge C_{10}^{k''}(p'') = 0$,

$\zeta_2 - S_2^{k''} = \{(p'', ŧ)\} : C_{11,\sigma_2}^{k''}(p'') = 1 \wedge C_{10}^{k''}(p'') \neq 0$, $\zeta_3 - S_3^{k'} = \varnothing$, a $\zeta_4 - S_4^{k'} = \text{false}$.

Akcje związane z wykonaniem tranzycji t w podsieci o numerze *k*:

$$\bigvee_{p \in C_1^k(t)} p \in S_1^k \Rightarrow S_1^k := S_1^k - p \qquad (3.14)$$

$$\bigvee_{p \in C_1^k(t)} p \in (p, ŧ) : (p, ŧ) \in S_2^k \Rightarrow S_2^k := S_2^k - (p, ŧ) \qquad (3.15)$$

$$\bigvee_{p \in C_1^k(t)} C_{11,\sigma_1}^k(p) = 1 \Rightarrow \bigvee_{p' \in P^H(p)} (C_{11,\sigma_1}^{k'}(p') = 1 \wedge C_{12}^{k'} = 1 \wedge \wedge \bigvee_{p'' : C_{11,\sigma_2}^{k'}(p'') = 2} p'' \notin S_1^{k'} \cup S_2^{k'}) \Rightarrow \zeta_4 \qquad (3.16)$$

$$\bigvee_{p\in C_1^k(t)} C_{11,\sigma_1}^k(p)=1 \Rightarrow \bigvee_{p'\in P^H(p)} (C_{11,\sigma_1}^{k'}(p')=1 \wedge C_{12}^{k'}=1 \wedge \wedge \bigvee_{p'':C_{11,\sigma_2}^{k'}(p'')=2} p''\in S_1^{k'} \cup S_2^{k'}) \Rightarrow \zeta_1 \wedge \zeta_2 \wedge \zeta_3 \wedge \zeta_4 \quad (3.17)$$

$$\bigvee_{p\in C_1^k(t)} C_{11,\sigma_1}^k(p)=1 \Rightarrow \bigvee_{p'\in P^H(p)} (C_{11,\sigma_1}^{k'}(p')=1 \wedge C_{12}^{k'}=0) \Rightarrow \Rightarrow \zeta_1 \wedge \zeta_2 \wedge \zeta_3 \wedge \zeta_4 \quad (3.18)$$

$$C_6^k(t)\neq 0 \Rightarrow S_3^k := S_3^k + (t, C_6^k(t)) \quad (3.19)$$

$$C_6^k(t)=0 \Rightarrow \bigvee_{s\in C_8^k(t)} STS := STS + s \quad (3.20)$$

Jeśli $C_6^k(t)=0$ lub w momencie, gdy $t=0$, gdzie: $(t,t)\in S^k{}_3$ wtedy wykonywane są kolejne działania:

$$\bigvee_{p\in C_7^k(t)} C_{10}^k(p)=0 \Rightarrow S_1^k := S_1^k + p \quad (3.21)$$

$$\bigvee_{p\in C_7^k(t)} C_{10}^k(p)\neq 0 \Rightarrow S_2^k := S_2^k + (p, C_{10}^k(p)) \quad (3.22)$$

$$\bigvee_{p\in C_7^k(t)} C_{11,\sigma_1}^k(p)=1 \Rightarrow \bigvee_{p'\in P^H(p)} C_{11,\sigma_1}^{k'}(p)=1 \wedge C_{11,\sigma_2}^{k'}(p)=1) \Rightarrow \Rightarrow S_4^{k'} := \text{true} \quad (3.23)$$

Dodatkowo na końcu każdego cyklu decyzyjnego wykonywane są następujące działania dla każdej podsieci:

$$\bigvee_{s\in L\cup Y} s := \text{false} \quad (3.24)$$

$$S_4^k=\text{true} \Rightarrow \bigvee_{p\in\{S_1^k\cup S_2^k\cup S_3^k\}} \bigvee_{s\in C_9^k(p)} s := \text{true} \quad (3.25)$$

$$\bigvee_{s\in STS} s := \text{true} \quad (3.26)$$

$$STS := \varnothing \quad (3.27)$$

Akcje związane z wykonaniem tranzycji uszeregowano zgodnie z kolejnością działań wykonywanych przez VDS i podzielone są na trzy grupy, z których pierwsza wykonywana jest natychmiast po odnalezieniu dozwolonej tranzycji, druga dopiero wówczas, gdy parametr wskazywany funkcją czasu dla tej tranzycji osiągnie wartość 0, a trzecia na końcu każdego cyklu decyzyjnego.

Akcje 3.14 oraz 3.15 odpowiadają odpowiednio usunięciu znaczników z miejsc wejściowych tranzycji w przypadku, gdy miejsca nie mają przypisanych parametrów czasowych 3.14 oraz 3.15 w przypadku przeciwnym. Akcja 3.16 odpowiada deaktywacji wszystkich tych podsieci wyprowadzonych funkcją hierarchii od makromiejsca wejściowego tranzycji t, które posiadają atrybut historii oraz nie spełniają warunku osiągnięcia miejsc końcowych. Akcja 3.17 odpowiada deaktywacji wszystkich tych podsieci wyprowadzonych funkcją hierarchii od makromiejsca wejściowego tranzycji, które posiadają atrybut historii oraz spełniają warunek osiągnięcia miejsc końcowych. Akcja 3.18 odpowiada deaktywacji wszystkich tych podsieci wyprowadzonych funkcją hierarchii od makromiejsca wejściowego tranzycji, które nie posiadają atrybutu historii. Akcje 3.19 i 3.20 oznaczają odpowiednio uruchomienie odpowiedniej akcji czasowej, w przypadku gdy tranzycja ma przypisany

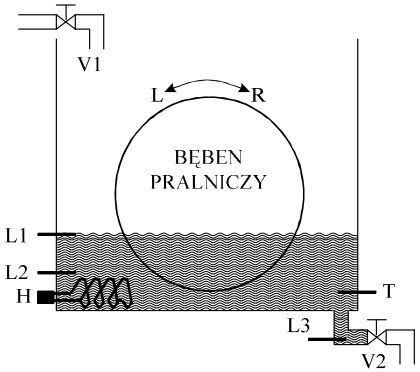
parametr czasowy, oraz ustawienie sygnałów przypisanych etykietą *action* do wykonywanej tranzycji.

W przypadku osiągnięcia wartości 0 przez parametr wskazywany funkcją czasu dla danej tranzycji, wykonywane są kolejne czynności. Akcje 3.21 oraz 3.22 odpowiadają wprowadzeniu znaczników do miejsc wyjściowych tranzycji w przypadku, gdy miejsca te nie posiadają parametrów czasowych 3.21 oraz 3.22 w przypadku przeciwnym. Akcja 3.23 odpowiada aktywacji wszystkich tych podsieci wyprowadzonych funkcją hierarchii od makromiejsca wyjściowego tranzycji, które posiadają atrybut miejsc początkowych.

Dodatkowo, na końcu każdego cyklu decyzyjnego wykonywane jest ustawianie sygnałów wewnętrznych oraz wyjściowych zgodnie ze stanem sieci określanym zawartością bloku ***S*** – akcje 3.24-3.27.

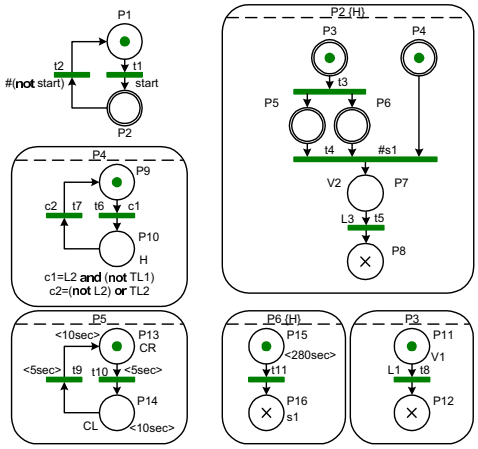
3.4. Przykład programowego modelu sieci

Prostym przykładem ilustrującym omawiane zagadnienia może być uproszczony układ sterowania praniem wstępnym w pralce automatycznej (rys. 3.2 i rys. 3.3). Po włączeniu programu otwierany jest zawór V1 i do pralki nalewana jest woda. Proces nalewania trwa do momentu osiągnięcia poziomu sygnalizowanego czujnikiem L1. Jednocześnie po przekroczeniu poziomu L2 (całkowite zanurzenie grzałki H) jeśli temperatura jest niższa od wymaganej włączany jest układ grzania H. Po zamknięciu zaworu V1 i podgrzaniu wody do wymaganej temperatury uruchamiany jest proces prania polegający na naprzemiennym obracaniu bębna pralniczego w lewą i prawą stronę przez 10s, z 5s przerwy przed zmianą kierunku obrotów. Proces utrzymywania temperatury wody na zadanym poziomie jest aktywny przez cały cykl prania. Cykl prania zamyka się po upływie 280s, po czym zatrzymywany jest ruch bębna, wyłączana grzałka i otwierany zawór usuwający wodę z pralki.



Rys. 3.2. Przykład układu sterowania pralką automatyczną

Do tak przedstawionego systemu sterowania można stworzyć model hierarchicznej sieci interpretowanej, realizujący wyżej wzmiankowane elementy funkcjonalne (rys. 3.3).



Rys. 3.3. Hierarchiczna sieć Petriego opisująca sterownik pralki automatycznej

Bloki funkcjonalne systemu decyzyjnego ***I***, ***O*** oraz ***L*** proponuje się w prosty sposób przedstawić jako jednowymiarowe tablice (rys. 3.4):

	start	L1	L2	L3	TL1	TL2
I	true	false	false	false	false	false

	V1	V2	H	CR	CL
O	false	false	false	false	false

	s1
L	false

Rys. 3.4. Bloki ***I***, ***O*** oraz ***L*** systemu decyzyjnego

Blok stanu systemu przedstawiono w formie tablicy, przy czym wiersze przyporządkowane są do kolejnych podsieci zbioru S (indeksy górne), ostatni wiersz reprezentuje zbiór STS, a kolumny przedstawiają odpowiednie podzbiory każdego zbioru S^k (indeksy dolne) (rys. 3.5).

	S_1	S_2	S_3	S_4		
S	{P1}	{}	{}	true	S1 sieć podstawowa	
	{P3,P4}	{}	{}	false	S2	P2
	{P11}	{}	{}	false	S3	P3
	{P9}	{}	{}	false	S4	P4
	{}	{(P13,10sec)}	{}	false	S5	P5
	{}	{(P15,280sec)}	{}	false	S6	P6
STS	{}					

Rys. 3.5. Blok stanu systemu dla sterownika pralki

Zawartość bloku stanu odzwierciedla stan sieci w momencie jej inicjalizacji, tzn. aktywne jest tylko miejsce STANDBY należące do sieci podstawowej. Jeśli spełniony

zostanie warunek *cond(t1)=true* (zgodnie z rys. 3.3), to po wykonaniu tranzycji t1 stan sieci zmieni się na następujący (rys. 3.6):

	S_1	S_2	S_3	S_4
S	{P2}	{}	{}	true
	{P3,P4}	{}	{}	true
	{P11}	{}	{}	true
	{P9}	{}	{}	true
	{}	{(P13,10sec)}	{}	false
	{}	{(P15,280sec)}	{}	false
STS	{}			

S1	sieć podstawowa
S2	P2
S3	P3
S4	P4
S5	P5
S6	P6

Rys. 3.6. Blok stanu systemu po wykonaniu tranzycji t1

Strukturę specyfikacji sieci dla powyższego przykładu przedstawiono w sposób następujący (rys. 3.7):

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
t1	{P1}	{}	{}	{start}	{}	0	{P2}	{}
t2	{P2}	{}	{}	{}	{not start}	0	{P1}	{}
t3	{P3}	{}	{}	{}	{}	0	{P5,P6}	{}
t4	{P4,P5,P6}	{}	{}	{}	{s1}	0	{P7}	{}
t5	{P7}	{}	{}	{L3}	{}	0	{P8}	{}
t8	{P11}	{}	{}	{L1}	{}	0	{P12}	{}
t6	{P9}	{}	{}	{c1}	{}	0	{P10}	{}
t7	{P10}	{}	{}	{c2}	{}	0	{P9}	{}
t9	{P14}	{}	{}	{}	{}	5sec	{P13}	{}
t10	{P13}	{}	{}	{}	{}	5sec	{P14}	{}
t11	{P15}	{}	{}	{}	{}	0	{P16}	{}

	C_9	C_{10}	σ_1	σ_2	σ_3	C_{12}	
P1	{}	0	0	1	-	-	C^1
P2	{}	0	1	0	2		
P3	{}	0	1	1	3	true	C^2
P4	{}	0	1	1	4		
P5	{}	0	1	0	5		
P6	{}	0	1	0	6		
P7	{V2}	0	0	0	-		
P8	{}	0	0	2	-		
P11	{V1}	0	0	1	-	false	C^3
P12	{}	0	0	2	-		
P9	{}	0	0	1	-	false	C^4
P10	{H}	0	0	0	-		
P13	{CR}	10sec	0	1	-	false	C^5
P14	{CL}	10sec	0	0	-		
P15	{}	280sec	0	1	-	true	C^6
P16	{s1}	0	0	2	-		

Rys. 3.7. Struktura specyfikacji dla kontrolera pralki automatycznej

3.5. Podsumowanie i wnioski

W rozdziale niniejszym zaprezentowano programowy model interpretowanych hierarchicznych sieci Petriego (PHPN), oraz zdefiniowano środowisko decyzyjne, umożliwiające jego realizację praktyczną. Model ten stanowi teoriomnogościową reprezentację modelu postaci klasycznej sieci hierarchicznych HPN (rozdz. 1.3). Zamieszczony materiał ilustracyjny obrazuje zarówno zasady tworzenia opisu zgodnego z PHPN, jak również sugeruje kierunek praktycznej realizacji, wykorzystującej zasady tworzenia struktur danych w językach programowania wysokiego poziomu.

Nasuwają się tu następujące spostrzeżenia:

- model programowy sieci hierarchicznej PHPN jest sformalizowaniem zasad realizacji praktycznej, wykorzystującej wirtualny system decyzyjny VDS,
- model umożliwia bezpośrednie odwzorowanie zbiorów składowych w odpowiednie struktury danych, znacznie ułatwiając tym samym proces syntezy programowej.

Rozdział 4

SYNTEZA SIECI PETRIEGO W MIKROSYSTEMACH CYFROWYCH

Przedstawiony w rozdziale 3 programowy model sieci Petriego można realizować praktycznie wykorzystując dowolny język programowania wysokiego poziomu. W niniejszym rozdziale przedstawiono zasady jego realizacji z wykorzystaniem standardu języka ANSI C, oraz wskazano najistotniejsze problemy związane z ich implementacją w monolitycznych strukturach mikrosystemów cyfrowych.

4.1. Realizacja modelu z wykorzystaniem standardu ANSI C.

Standard ANSI C opracowany został w roku 1989 przez American National Standards Institute i opublikowany jako ANS X3.159-1989 w roku 1990. Standard ten powstał na bazie języka opracowanego przez B.Kernighana i D.Ritchie (Kernighan i Ritchie 1988). Zyskał on bardzo dużą popularność nie tylko w sensie pisania oprogramowania dla komputerów klasy IBM PC, ale również jako podstawowy język używany do oprogramowywania wszelkich mikroprocesorowych systemów przemysłowych, a w szczególności mikrokontrolerów jednoukładowych oraz mikrosystemów cyfrowych. Praktycznie żaden nowy procesor nie opuszcza wytwórni bez opracowanego dla niego kompilatora języka C. Z tego względu zdecydowano się w pracy przedstawić (bardzo ogólnie) zasady realizacji z wykorzystaniem właśnie tego standardu.

4.1.1. Implementacja struktur danych

Blok ***S*** przechowujący informacje o stanie systemu można w prosty sposób zaimplementować w języku C jako strukturę danych zgodną z modelem przedstawionym w punkcie 3.2 oraz na rys. 3.5. Standard ANSI C przewiduje możliwość deklarowania tablic struktur, tak więc jej ogólny wygląd można przedstawić następująco:

```
struct _TNodes
{
        ntype   *Nodes;
        ntype   *Times;
```

```
};

struct _S
{
    ntype       *Places;
    struct      _TNodes     TPlaces;
    struct      _TNodes     TTransitions;
    logic       Enabled;
};

struct _BS
{
    struct      _S      S[k];
    ntype       STS;
};
```

Rys. 4.1. Implementacja bloku stanu systemu

Kolejne pola w strukturze _S (Places, TPlaces, TTransitions, Enabled) odpowiadają składowym zbioru S^k (S_1, S_2, S_3 oraz S_4), przy czym pola TPlaces oraz TTransitions implementowane są jako wyróżnione struktury _TNodes, odpowiadające odpowiednim zbiorom par uporządkowanych (patrz S_2, S_3).
Bloki ***I***, ***L*** oraz ***O*** można realizować jako odrębne tablice takie że:

```
logic   BI[number_of_inputs];
logic   BL[number_of_internal_signals];
logic   BO[number_of_outputs];
```

Rys. 4.2. Implementacja bloków ***I***, ***L*** oraz ***O***

Blok specyfikacji realizowany jest analogicznie do bloku stanu systemu:

```
struct _C
{
    ntype       **PrePlaces;
    ntype       **EnablingPlaces;
    ntype       **InhibitPlaces;
    ntype       *Conditions;
    ntype       *Aborts;
    ntype       *TTimes;
    ntype       **PostPlaces;
    ntype       **TActions;
    ntype       **PActions;
    ntype       *PTimes;
    ntype       **KindOfPlaces;
    logic       History;
} C[k];
```

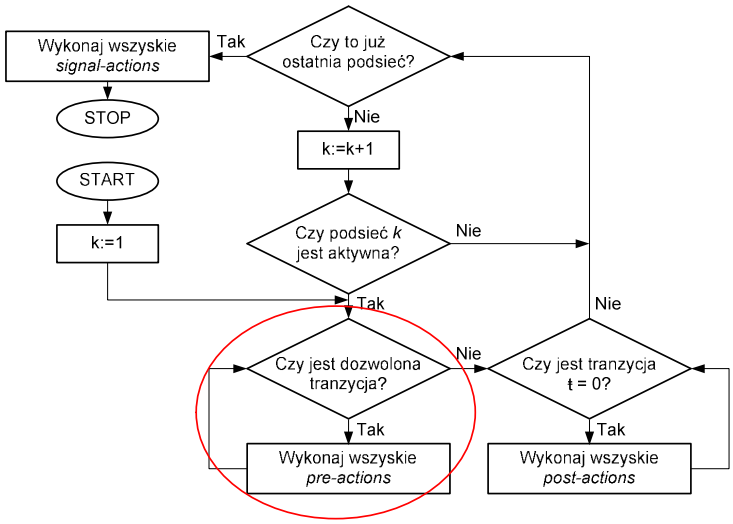
Rys. 4.3. Implementacja bloku specyfikacji

Kolejność pól w strukturze _C odpowiada kolejności składowych w zbiorze C^k zdefiniowanym w rozdziale 3.1.

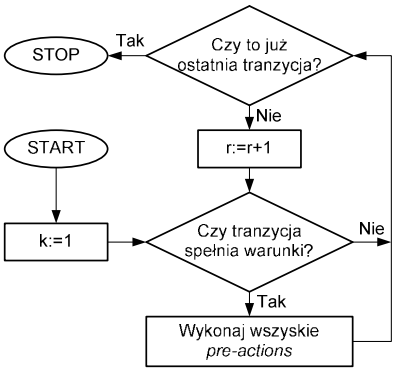
4.1.2. Implementacja systemu decyzyjnego

Blok decyzyjny odpowiada za przeprowadzenie systemu do stanu kolejnego, a więc zarówno za sprawdzenie warunków zezwolenia dowolnej tranzycji, jak i ewentualne przeprowadzenie akcji związanych z jej wykonaniem. W ogólnym zarysie sposób realizacji może zostać przedstawiony tak jak na rys. 4.4. Jest to oczywiście duże uproszczenie, przybliżające jedynie ogólną zasadę działania VSD.

Nazwą *pre-actions* objęte są akcje 3.14-3.20 związane z wykonaniem tranzycji i opisane w rozdziale 3.3. Nazwą *post-actions* objęte są akcje 3.21-3.23, natomiast *signal-actions* dotyczą odpowiednio akcji 3.24-3.27. Zapis $\dot{t}=0$ odnosi się do sytuacji takiej, że: $C_6^k(t)=0$ lub w momencie, gdy $\dot{t}=0$ dla $(t,\dot{t})\in S^k{}_3$.



Rys. 4.4. Schemat działania bloku decyzyjnego



Rys. 4.5. Pętla sprawdzania tranzycji

W diagramie z rys. 4.4 nie pokazano zasad sprawdzania (i wykonania) tranzycji dozwolonych w obrębie podsieci. Schemat przedstawiony na rys. 4.5 stanowi rozwinięcie fragmentu diagramu z rys. 4.4 zaznaczonego owalem, ilustrując jednocześnie wzmiankowane zasady.

Wewnątrz każdej aktywnej podsieci sprawdzane są wszystkie tranzycje w poszukiwaniu dozwolonej. Jeśli taka zostanie odnaleziona, to wykonane zostaną odpowiednie akcje (*pre-actions*), a system powróci do poszukiwania kolejnych dozwolonych tranzycji. W ten sposób realizowany jest postulat synchroniczności sieci, mówiący o wykonaniu wszystkich dozwolonych tranzycji w jednym cyklu decyzyjnym.

Realizacja bloku systemu decyzyjnego z wykorzystaniem języka C, może ogólnie być przedstawiona następująco:

```
void VDS(void)
{
    ntype k,r;
    for(k=1;k<=number_of_macros;k++) {
        if(S.S[k].Enabled) {
            for(r=1;r<=number_of_transition_in_subnet;r++) {
                if(Conditions(k,r))
                    Pre_Actions(k,r);
            }
            for(r=1;r<=number_of_transition_in_S^k_3;r++) {
                if(End_Of_TTime(k,r))
                    Post_Actions();
            }
        }
    }
    Signal_Actions();
}
```

Rys. 4.6. Realizacja systemu decyzyjnego

Budowę poszczególnych funkcji składowych, a więc Conditions(), Pre_Actions(), Post_Actions(), Signal_Actions oraz End_Of_TTime() pominięto ze względu na ich zbyt szczegółowy charakter.

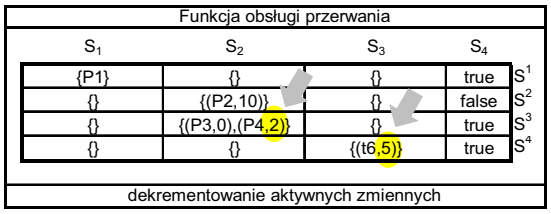
4.1.3. Realizacja parametrów czasowych

Istotnym problemem programowej implementacji czasowych sieci Petriego jest realizacja parametrów czasowych. Jego waga spoczywa głównie na wskazaniu metody odmierzania żądanych interwałów czasowych, z błędem mieszczącym się w przedziale dopuszczalnym.

Jak zostało przedstawione w rozdziale 3, wartości funkcji czasu dla aktywnych węzłów sieci, przechowywane są jako liczby w bloku ***S*** stanu systemu. Zadaniem systemu decyzyjnego jest zainicjowanie odpowiedniej zmiennej, sprawdzenie czy osiągnęła ona wartość końcową i jeśli tak, to usunięcie jej z listy zmiennych. Natomiast zarządzanie zmiennymi od momentu ich inicjalizacji do usunięcia, sprowadza się do opracowania funkcji zarządzającej, wywoływanej w ściśle określonych chwilach czasu.

Jedną z najprostszych metod realizacji tak postawionego problemu wydaje się wykorzystanie systemu przerwań modułu mikroprocesorowego. Wywołanie obsługi przerwania z ustaloną częstotliwością może być wynikiem dołączenia do zewnętrznego wejścia przerwań układu generatora, lub też wykorzystania wewnętrznego układu czasowo/licznikowego (większość mikrosystemów cyfrowych oraz mikrokontrolerów takowe posiada).

W momencie aktywacji węzła z niezerowym parametrem czasowym w bloku BS tworzona jest jej odpowiednia zmienna i nadawana wartość początkowa t_p, taka że $t_p=t_a*f_i$, gdzie t_a jest całkowitym czasem przyporządkowanym do węzła (w [s]), f_i jest częstotliwością wywoływania przerwania (w [Hz]). Wartość t_a jest pobierana z odpowiednich pól bloku specyfikacji (C^k_6 oraz C^k_{10}). Generalnie częstotliwość wywoływania przerwań należy tak dobrać, ażeby wartości t_p były całkowite. Po wywołaniu funkcji obsługi parametrów czasowych, wszystkie aktywne zmienne (tzn. niezerowe oraz dla podsieci aktywnych) ulegają dekrementacji. Usunięcie zmiennych następuje po osiągnięciu przez nie wartości 0, w momencie wykonywania odpowiednich akcji związanych z przeprowadzeniem systemu do stanu kolejnego (patrz akcje związane z wykonaniem tranzycji, rozdz. 3.3). Ilustrację przedstawionego mechanizmu zamieszczono na rys. 4.7.



Rys. 4.7. Mechanizm zarządzania zmiennymi parametrów czasowych

Na dokładność odmierzania interwałów czasowych w proponowanym systemie ma wpływ wiele czynników. Wymienić można chociażby kilka najważniejszych:

- niedokładność układu generowania przerwań,
- opóźnienie wywołania procedury obsługi przerwania,
- czas obsługi przerwania,
- ilość aktywnych zmiennych czasowych.

Zakładając wystarczająco dobry pod względem dokładności układ generowania przerwań, można go pominąć bez istotnego wpływu na błąd interwałów czasowych. Pozostałe czynniki silnie uzależnione są m.in. od: użytego procesora, prędkości taktowania procesora, a także od jakości wykonywalnego kodu wygenerowanego przez użyty kompilator.

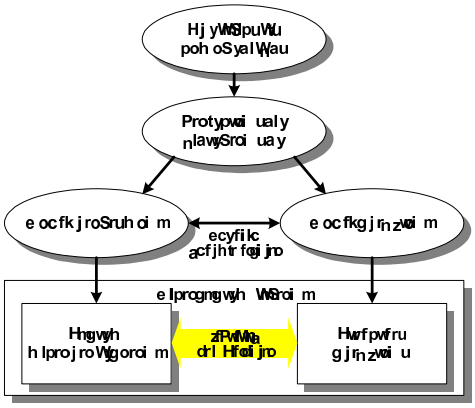
4.2. Dekompozycja sieci

Rzeczywiste systemy sterowania bardzo często wymagają spełnienia określonych założeń związanych z czasami reakcji systemu na wymuszenia wejściowe. Realizacja

programowa, jakkolwiek może być tańsza od sprzętowej, to jednak jej czasy odpowiedzi mogą być wielokrotnie dłuższe.

Gdy do takiej sytuacji dołączony zostanie jeszcze aspekt realizacji skomplikowanych systemów sterowania, o bardzo dużej liczbie stanów wewnętrznych, to niezbędnym wydaje się opracowanie algorytmu dekompozycji sieci Petriego, wspomagającego automatyczny podział systemu na część sprzętową i programową. Szczególne znaczenie ma to w przypadku wykorzystania mikrosystemów cyfrowych, jako platformy implementacyjnej. Podział taki będzie dalej nazywany dekompozycją systemową.

W ogólności zakłada się homogeniczny model opisu zachowania systemu sterowania, wykorzystujący interpretowane hierarchiczne sieci Petriego (rys. 4.8).



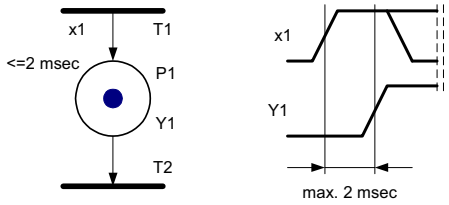
Rys. 4.8. Dekompozycja systemowa w mikrosystemie cyfrowym

Problematyka dekompozycji systemowej jest ważnym zagadnieniem, które w znaczący sposób decyduje o najważniejszych parametrach realizacji: kosztach oraz parametrach dynamicznych (czasach reakcji). W szeregu prac (Gajski 1997, Jerraya *i in.* 1999, Skowroński 1998) znaleźć można zalecenia dotyczące przyjmowania takich rozwiązań, które przy zachowaniu spójności z założonym modelem zachowawczym umożliwią osiągnięcie rozsądnych kompromisów w odniesieniu do wzmiankowanych parametrów. Jednakże rozwiązania tam omawiane bazują w ogólności na szacowaniu parametrów systemu po zakończeniu procesu syntezy, i w oparciu o nie przeprowadza się korekcję, prowadzącą do kolejnej syntezy. Wynikająca z takiej metodologii cykliczność może w znaczący sposób wpłynąć zarówno na wydłużenie czasu procesu projektowania, jak i na problemy w jego zautomatyzowaniu. W pracy omówiono najważniejsze aspekty wpływające na podział systemu sterowania wraz z propozycją ich interpretacji pod kątem wykorzystania w procedurach dekompozycji systemowej.

4.2.1. Czas reakcji

Czas reakcji zwykle jest określany na podstawie symulacji, bądź badań fizycznych. Dopiero na tej podstawie podejmowana jest decyzja co do poprawności przyjętego rozwiązania, tzn. czy czasy reakcji są wystarczająco małe. Takie podejście

utrudnia proces projektowania automatycznego. Można spróbować odwrócić problem i dodać informacje o dynamice systemu do modelu behawioralnego. W ten sposób narzędzia dekompozycji i syntezy będą miały dostarczone informacje, niezbędne do przeprowadzenia dalszych etapów projektowania. Tak postawiony problem można stosunkowo prosto zrealizować, dodając do opisu zachowawczego elementy, niosące informacje o maksymalnych czasach reakcji systemu na zmiany warunków wejściowych w określonych miejscach systemu. Ilustruje to rys. 4.9.



Rys. 4.9. Interpretacja czasu reakcji

Parametr <=2 msec nazwać można czasem reakcji i należy interpretować go jako maksymalny czas, który może upłynąć od momentu ustawienia sygnału wejściowego x1 do ustawienia sygnału wyjściowego Y1.

Parametr ten nie został uwzględniony w żadnym z prezentowanych modeli sieci Petriego, ze względu na fakt, iż nie jest on elementem opisu zachowania, a jedynie niesie informację o zaleceniach realizacyjnych. Wobec powyższego, traktowany jest on tylko jako dodatkowa informacja, która może być dodana w systemie wspomagającym projektowanie, a więc w plikach edytora graficznego sieci lub w jej reprezentacji tekstowej.

4.2.2. Koszt realizacji

W przypadku realizacji systemu sterowania w pojedynczym mikrosystemie cyfrowym, szacowanie kosztów realizacji układu rzeczywistego nie ma większego sensu. Jednakże dla zachowania uniwersalności metody, tzn. możliwości jej zastosowania dla innych platform implementacyjnych, celowym wydaje się omówienie tego parametru.

Przyjmuje się ogólne założenie, iż koszty związane z realizacją programową są mniejsze od kosztów realizacji sprzętowej (Balarin 1997, Jerraya *i in.* 1999). Wynika to głównie z faktu, iż wciąż jeszcze zarówno same struktury układowe systemów mikroprocesorowych są tańsze, jak również tańsze są narzędzia wspomagające projektowanie.

Koszt realizacji praktycznej, niezależnie od przyjętej platformy realizacyjnej, posiada pewne elementy stałe, np.:

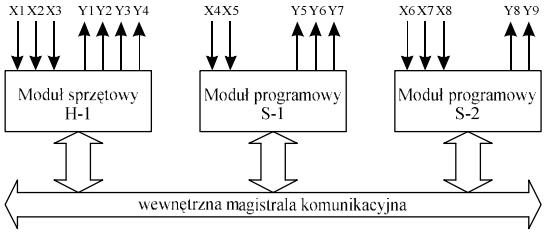
- koszt użytych struktur,
- koszt układów dodatkowych, niezbędnych do prawidłowej pracy systemu,
- koszt płytek drukowanych,
- koszt interfejsów komunikacyjnych.

Optymalny dobór struktur układowych jest istotnym czynnikiem projektowania, zapewniającym właściwe ograniczenie kosztów i związane z tym zwiększenie

potencjalnych zysków. Optymalność w tym przypadku rozumie się jako zaspokojenie potrzeb układowych (zapotrzebowanie na wielkość określonych zasobów), przy minimalizacji kosztów. Celowym więc wydaje się wprowadzenie do systemu projektowania zintegrowanego – biblioteki (bazy danych), w której będą zawarte informacje o dostępnych modułach: zasobach przez nie wnoszonych, kosztach związanych z ich zastosowaniem, oraz wnoszonym przez nie zużyciu energii. Pozwoli to na szybkie wyszukanie elementów spełniających zadane kryteria.

4.2.3. Zasoby wejścia/wyjścia

Przy podziale systemu na moduły należy w miarę możliwości starać się, aby każdy z nich posiadał własny zasób sygnałów wejścia/wyjścia (rys. 4.10). Spełnienie powyższego może w wydatny sposób ułatwić implementację i zmniejszyć czasy reakcji. Transmisja wartości nawet jednego sygnału zewnętrznego pomiędzy modułami może spowodować znaczące pogorszenie dynamiki projektowanego systemu.



Rys. 4.10. Przykład podziału uwzględniającego zasoby we/wy

4.2.4. Algorytm dekompozycji systemowej

W istniejących pakietach współprojektowania sprzętu i oprogramowania wyróżnić można dwie metody dekompozycji systemowej, w której jako podstawowy wyróżnik podziału przyjmuje się koszt realizacji. W COSYMIE założono, iż całość projektu realizowana jest wstępnie jako oprogramowanie, a następnie odrywane są te elementy, które nie spełniają wymogów czasowych i stąd przenoszone są do części sprzętowej. VULCAN natomiast początkowo implementuje wszystko w sprzęcie, a następnie dla obniżenia kosztów przenosi kolejne moduły do programu (oczywiście te, które nie są obarczone silnymi ograniczeniami dynamicznymi) (Herman *i in.* 1994, Ku i Micheli 1988).

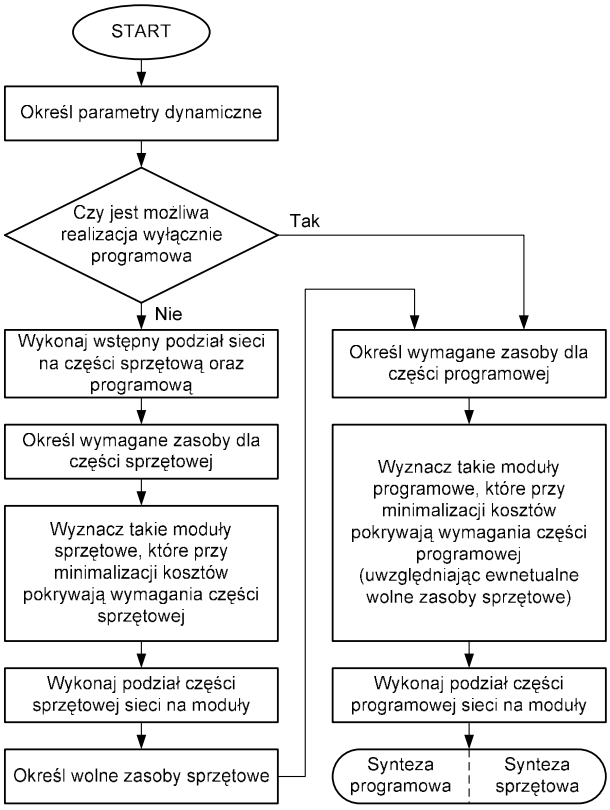
Podejścia takie jakkolwiek dają pożądane rezultaty, to jednak pociągają za sobą wydłużenie czasu samego projektowania, ze względu na wprowadzenie pętli, w której cyklicznie przeprowadza się syntezę, szacowanie rezultatów i weryfikację rozwiązania prowadzącą do kolejnej syntezy.

Alternatywą może być takie przeprowadzenie dekompozycji, które w oparciu o przesłanki wynikające ze specyfikacji systemu oraz dane dotyczące dostępnych modułów realizacyjnych umożliwi osiągnięcie spełnienia wszystkich kryteriów

w możliwie najmniejszej ilości iteracji, co przy dużej ilości wymogów (często ze sobą sprzecznych) nie jawi się jako zadanie trywialne.

Proponuje się wprowadzenie algorytmu dekompozycji systemowej, promującego realizację programową i uwzględniającego omawiane kryteria podziału (rys. 4.11).

Oczywiście jest to tylko ogólny zarys właściwej dekompozycji, pozbawiony m.in. elementów weryfikacji. Każdy element diagramu stanowi skomplikowany problem, nad którego rozwiązaniami prowadzone są odrębne prace. Chociażby określenie wymaganych zasobów wymaga wyznaczenia precyzyjnych funkcji estymacyjnych, zarówno dla różnych topologii sieci, jak i dla wykorzystywanych modułów realizacyjnych.



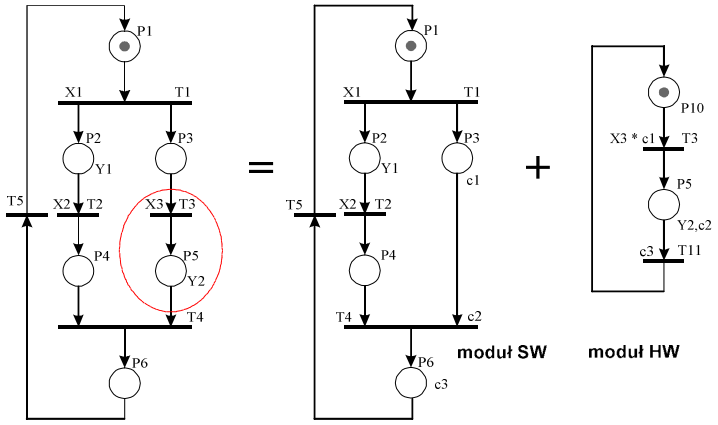
Rys. 4.11. Szkic algorytmu dekompozycji systemowej

W proponowanej metodzie najważniejszym kryterium wyznaczającym wstępne rozdzielenie sieci pierwotnej są czasy reakcji. Spełnienie warunków wynikających z ich nałożenia na model formalny ma nadrzędne znaczenie wobec wszystkich pozostałych parametrów. Pozostałe elementy algorytmu bazują na kryterium kosztu promującym dobór takich modułów bibliotecznych, które pozwolą na najtańszą realizację systemu. Końcowe ustalenie podziału sieci powinno uwzględniać również

zasoby wejścia/wyjścia, co w efekcie wyczerpuje listę omawianych parametrów dekompozycji systemowej.

Warta omówienia wydaje się jeszcze metoda samego podziału sieci na fragmenty. Istnieją co prawda znane algorytmy podziału sieci Petriego (Augin *i in.* 1980, Banaszak *i in.* 1993, Gallier 1986, Tal i Yuditskiy 1982), lecz ich możliwość zastosowania dla potrzeb dekompozycji systemowej jest mocno ograniczona. Wynika to przede wszystkim z faktu, iż zostały one opracowane dla innych potrzeb (np. kodowanie, wyodrębnienie składowych automatowych) i zastosowanie ich w dekompozycji systemowej nie przynosi zadowalających rezultatów. Stąd proponuje się tworzenie sieci autonomicznych z odrywanych elementów, z równoczesnym uzupełnieniem sieci bazowej do stanu zapewniającego zgodność działania z siecią pierwotną.

Na rys. 4.12 przedstawiono przykład ilustrujący omawiane zagadnienie. Załóżmy, iż odrywany fragment sieci {T3, P5} nie spełnia wymagań dynamicznych. Z fragmentu tego tworzona jest odrębna sieć (HW module). Prawidłową jej pracę uzyskuje się wprowadzając węzły pomocnicze {P10,T11}. Sieć bazową (SW Module) uzupełnia się łukiem, a dla zapewnienia właściwej synchronizacji obu sieci wprowadza się dodatkowe sygnały c1, c2, i c3.



Rys. 4.12. Przykład podziału sieci Petriego na podsieci autonomiczne

W sytuacji, gdy do miejsca P3 byłaby przyporządkowana etykieta akcji wyjściowej, wówczas w miejsce łuku zastępczego (P3,T4), należałoby wprowadzić kolejno elementy: łuk (P3,T31), tranzycję T31 warunkowaną akcją *cond*(T31)=X3, miejsce P51, łuk (P51,T4). Działanie takie jest niezbędne ze względu na konieczność deaktywacji sygnałów wyjściowych, związanych z miejscem P3, po wykonaniu tranzycji T3.

4.3. Komunikacja międzymodułowa i synchronizacja zadań

Realizacja systemów sterowania cyfrowego, a w szczególności systemów współbieżnych, w strukturach integrujących moduł sprzętowy z mikroprocesorem,

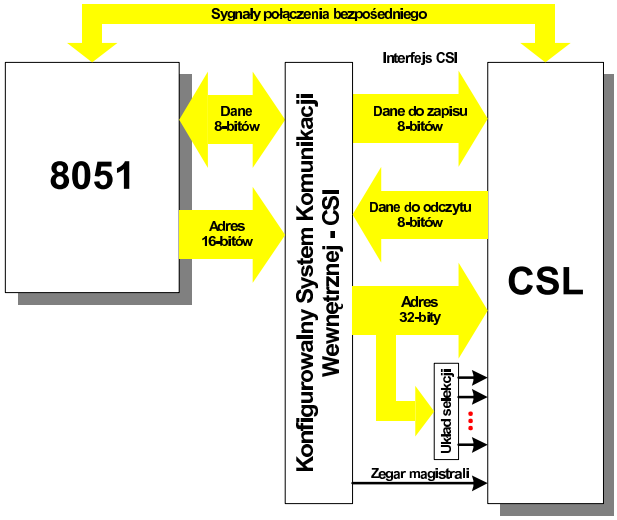
wymaga nie tylko dekompozycji systemowej na część programową oraz sprzętową, ale również precyzyjnego określenia zasad wymiany informacji pomiędzy obu modułami, czyli zdefiniowania tzw. interfejsu systemowego (ang. *On-Chip Subsystem Interface*).

W ogólności, w mikrosystemach cyfrowych stosowane są różne metody wymiany informacji pomiędzy modułami wewnętrznymi. Problem ten jednakże daje się sprowadzić do komunikacji mikroprocesora z urządzeniami peryferyjnymi. Do najczęściej stosowanych metod należą:

- wspólna przestrzeń pamięci,
- system przerwań,
- sygnały połączenia bezpośredniego.

Mikrosystem TRISCEND E5 CSoC

W układzie TRISCEND E5 CSoC (rys. 4.13) zastosowano interfejs komunikacji wewnętrznej nazwany konfigurowalnym systemem połączeń CSI (ang. *Configurable System Interconnect*). Od strony procesora dostępny jest on poprzez magistrale: danych i adresową, a od strony struktury sprzętowej CSL (ang. *Configurable System Logic*) dostęp odbywa się z wykorzystaniem magistrali danych odczytywanych, danych zapisywanych, układu selekcji oraz zegarowego sygnału synchronizującego. System CSI został zaprojektowany jako niezależny od konkretnej struktury układu, dlatego też zapewnia komunikację nie tylko pomiędzy procesorem a CSL, ale również pomiędzy innymi blokami mikrosystemu (np. DMA, JTAG). Struktura sprzętowa jest przyporządkowana do określonego zakresu przestrzeni adresowej systemu (0x10_0000 – 0x7F_FFFF).



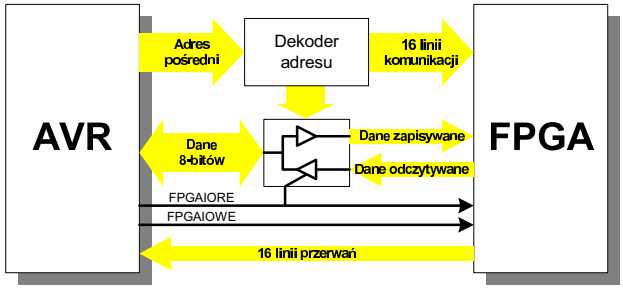
Rys. 4.13. Interfejs komunikacji międzymodułowej w układzie TRISCEND E5 CSoC

Ze względu na zastosowanie procesora zgodnego ze standardem Intel 8051 (o 16-bitowej magistrali adresowej), dostęp do całej przestrzeni odbywa się z wykorzystaniem układu mapującego. Operacja przekazania informacji do struktury

sprzętowej (zapis), wykonywana jest przez zapisanie 8-bitowej danej do określonego rejestru, wspólnego dla wszystkich makrokomórek w wybranym banku CSL. Informacja pobrania informacji ze struktury sprzętowej (odczyt), odbywa się przez jednoczesne pobranie stanu wyjść wszystkich makrokomórek w wybranym banku CSL i złożenie ich w słowo 8-bitowe. Dodatkowo możliwe jest też wykorzystanie kilku linii portów I/O procesora do bezpośredniej komunikacji ze strukturą CSL, pod warunkiem jednak, iż porty te nie spełniają innych zadań, np. komunikacji szeregowej przez port UART czy zliczania impulsów zewnętrznych.

Mikrosystem ATMEL AT94K

W układzie ATMEL AT94K (rys. 4.14) zastosowano rozwiązanie wykorzystujące pośrednie adresowanie struktury FPGA poprzez układ dekodera zawartości pięciu bajtów z przestrzeni pamięci procesora AVR (0x33-0x37). Wykorzystano w nim możliwość definiowania 16 niezależnych linii komunikacji oraz 8-bitowej dwukierunkowej magistrali danych. Dostęp do konkretnych makrokomórek określany jest na poziomie operacji programowania matrycy połączeń struktury sprzętowej. Opcjonalnie przewidziano do komunikacji jednostronnej FPGA→AVR, możliwość wykorzystania systemu przerwań procesora poprzez bezpośrednie przyporządkowanie do niego 16 sygnałów z FPGA.

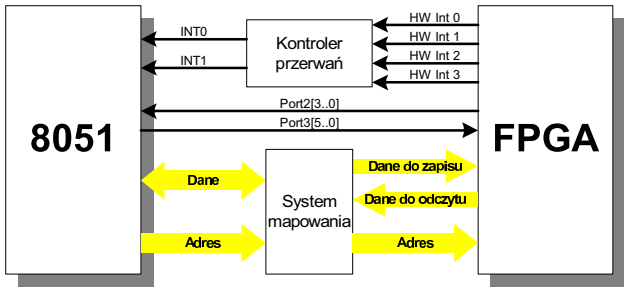


Rys. 4.14. Interfejs komunikacji międzymodułowej w układzie Atmel AT94K

Mikrosystem FIPSoC

W układzie FIPSoC (rys. 4.15) dostęp do struktur FPGA z poziomu rdzenia mP8051 jest realizowany z wykorzystaniem przestrzeni pamięci wewnętrznej lub/i zewnętrznej RAM procesora oraz systemu mapującego (adresy 0x30-0x6F oraz 0x1800-0x18FF). Dodatkowo, jeśli niektóre porty procesora są niewykorzystane w inny sposób, to jest możliwe użycie ich jako szybkiego połączenia bezpośredniego: Port2[3..0] z kierunkiem FPGA→mP8051 oraz Port3[5..0] z kierunkiem mP8051→FPGA. Poza tym przewidziano możliwość wykorzystania zewnętrznego kontrolera przerwań, oraz czterech sygnałów przerwań pochodzących z części sprzętowej (FPGA).

Niebagatelne znaczenie ma też informacja o prędkości przesyłania informacji, szczególnie gdy realizowany system ma docelowo pracować w czasie rzeczywistym.



Rys. 4.15. Interfejs komunikacji międzymodułowej w układzie FiPSoC

W tabeli 4.1 zebrano informacje dotyczące przybliżonej szybkości transferu danych dla wybranych metod komunikacji, przedstawionych wcześniej mikrosystemów cyfrowych, na podstawie danych katalogowych (http://www.atmel.com, http://www. sidsa.com, http://www.triscend.com).

Tab. 4.1. Przybliżone szybkości transferu danych różnych interfejsów komunikacyjnych

Parametr	**Układ**			**Jednostka**
	FiPSoC	E5 CSoC	AT94K	
częstotliwość taktowania µSC.	16	40	32	MHz
częstotliwość taktowania rdzenia µP	48	40	32	MHz
czas odpowiedzi na przerwanie	170ns	120	40	ns
dostęp do pamięci RAM	24	20	7	ns
propagacja sygnału z FPGA do µP	-	12	0,2	ns

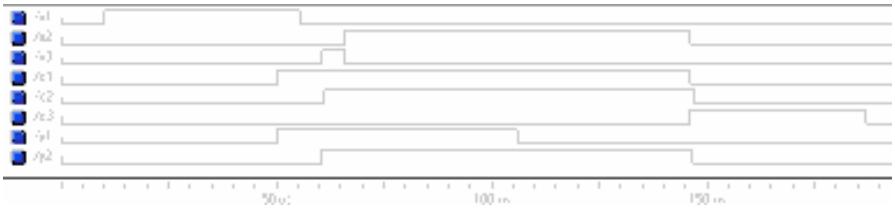
Przedstawione sposoby wymiany informacji pomiędzy modułami procesora i sprzętowej matrycy programowalnej w różny sposób nadają się do wykorzystania w implementacji systemów sterowania cyfrowego, zdekomponowanych na moduły programowe i sprzętowe. Dla realizacji systemów sterowania danych w postaci interpretowanych sieci Petriego i realizowanych z wykorzystaniem programowego modelu sieci, bazującego na programowym systemie decyzyjnym VDS, najbardziej korzystne wydaje się wykorzystanie metod komunikacji, których działanie opiera się na przesłaniach grupowych informacji.

Wynika to głównie z faktu, iż stan sygnałów zewnętrznych (wejściowych i wyjściowych) oraz synchronizujących (komunikacja z modułem sprzętowym), przechowywany jest w dwóch blokach funkcjonalnych systemu I oraz O. Bloki te realizowane są programowo jako jednowymiarowe tablice, których zawartość w naturalny sposób tworzy wzmiankowane grupy informacji. Dodatkowe bloki wejściowy oraz wyjściowy, mają za zadanie pozyskiwanie informacji o stanie sygnałów rzeczywistych. W przypadku, gdy dostęp do modułu sprzętowego i portów I/O następuje z wykorzystaniem wspólnej przestrzeni adresowej, wówczas bloki I oraz O systemu decyzyjnego można umieścić w obszarze wspólnym, eliminując w ten sposób realizację bloków wejściowego i wyjściowego.

Wykorzystanie sygnałów połączenia bezpośredniego może dać także pozytywne rezultaty przy syntezie interfejsu komunikacji, choć należy się liczyć z faktem, iż

liczba ich jest ograniczona zwykle do najwyżej kilkunastu, co determinuje w konsekwencji również ograniczoną liczbę modułów sprzętowych. Stosowanie systemu przerwań w zasadzie nie jest zalecane, ze względu na jednokierunkowy charakter połączenia.

Testy praktyczne wykonane zostały z wykorzystaniem struktury ATMEL AT94K40AL-25DQC oraz pakietu SystemDesigner, wspomagającego projektowanie mikrosystemów cyfrowych FPSLIC. Jako system sterowania cyfrowego przyjęto bardzo prosty przykład, opisany interpretowaną siecią Petriego i zdekomponowany na moduły: programowy i sprzętowy (jak na rys. 4.12), zrealizowane odpowiednio w języku ANSI C oraz VHDL. Na rys. 4.16. przedstawiono wybrany fragment przebiegów czasowych koweryfikacji wykonanej po syntezie strukturalnej omawianego przykładu.



Rys. 4.16. Przebiegi czasowe koweryfikacji dla przykładu z rys. 4.17.

W tabeli 4.2 zebrano czasy reakcji poszczególnych modułów oraz czasy transmisji sygnałów komunikacyjnych po magistralach wewnętrznych, wyznaczonych w trakcie symulacji systemu. Ze względu na jednolity charakter zarządzania sygnałami z 16 linii komunikacyjnych oraz magistrali danych w układzie FPSLIC, czasy wykorzystujące obie metody są identyczne i nie wyróżniono ich w zestawieniu.

Tab. 4.2. Czasy propagacji sygnałów oraz reakcji modułów sprzętowo-programowych

Parametr	Wartość	Jednostka
częstotliwość taktowania rdzenia FPGA	25	MHz
częstotliwość taktowania rdzenia AVR	25	MHz
czas reakcji modułu VDS	40	µs
czas pobierania/zapisu informacji z/do magistrali przez AVR	0,5	µs
czas propagacji po magistralach wewnętrznych FPGA	0,2	ns
czas reakcji modułu sprzętowego	42*	ns

* czas ten jest głównie uzależniony od częstotliwości taktowania modułu sprzętowego

4.4. Podsumowanie i wnioski

W rozdziale niniejszym pokazano zasady realizacji modelu PHPN z wykorzystaniem standardu języka C. Zarysowano szkielet systemu decyzyjnego oraz metodykę odmierzania interwałów czasowych. Na tej podstawie można wnioskować, iż:

- mogą zaistnieć sytuacje, w których fragmenty sieci muszą być wykonywane szybciej niż może to zapewnić realizacja programowa, co pociąga konieczność podziału sieci na części realizowane w sposób programowy i sprzętowy,
- istniejące algorytmy dekompozycji sieci opracowane zostały dla potrzeb kodowania lub wyodrębniania składowych automatowych, a ich zastosowanie do dekompozycji systemowej nie przynosi zadowalających rezultatów,
- koniecznym wydaje się określenie zasad dekompozycji systemowej sieci Petriego na części programową i sprzętową.

W rozdz. 4 zaprezentowano propozycję algorytmu dekompozycji systemowej sieci Petriego, bazującego na odrywaniu fragmentów sieci, nie spełniających narzuconych rygorów czasowych i tworzeniu z nich odrębnych modułów realizowanych sprzętowo. Pokazano też zasady komunikacji wewnątrz-modułowej w istniejących strukturach mikrosystemów cyfrowych, wskazując technikę przesłań grupowych jako najbardziej zalecaną dla potrzeb realizacji zdekomponowanej sieci programowej PHPN.

Rozdział 5

WYNIKI EKSPERYMENTÓW

W niniejszym rozdziale zebrano i opisano wyniki syntezy różnych systemów sterowania cyfrowego opisanych sieciami Petriego, z wykorzystaniem modelu programowego. Przedstawiono porównania z wynikami syntezy innych narzędzi wspomagających (np. ESTEREL, POLIS), oraz wskazano najważniejsze zależności pomiędzy topologią sieci a rozmiarem kodu wynikowego syntezy.

5.1. Porównanie wyników syntezy z wykorzystaniem różnych narzędzi wspomagających

Porównanie wyników syntezy proponowanego modelu z wynikami innymi narzędzi wspomagających projektowanie zintegrowane przeprowadzone zostało z wykorzystaniem kilku przykładów testowych o różnej skali skomplikowania i różnej strukturze, umożliwiającej uwzględnienie m.in. takich elementów opisu jak hierarchia, czy zależności czasowe. I tak pod rozwagę wzięto następujące testy:

- Cruse – specyfikacja układu stabilizacji prędkości jazdy pojazdów samochodowych (Andre 1996),
- DefFeq – specyfikacja rozwiązująca przykładowe równanie różniczkowe (Brener i Gajski 1990, Skowroński 2000),
- Elliptic – specyfikacja funkcjonalna filtru eliptycznego (Orchad 1990, Skowroński 2000),
- WWatch – specyfikacja układu zegarka (Berry 1991),
- DckCntr – model układu kontrolera magnetofonu kasetowego (Andre 1996),
- Parker86 – standardowy model porównawczy, wykorzystywany w syntezie wysokiego poziomu, zaczerpnięty z pracy (Skowroński 2000),
- RS232 – specyfikacja modułu odbiornika RS232 (Skowroński 2000),
- Count3 – synchroniczny licznik trzybitowy z zerowaniem, opis bez współbieżności (automatowy),
- TVpilot – uproszczona wersja kontrolera pilota telewizyjnego, opis ze współbieżnością modelowaną zgodnie z technologią diagramów Statecharts, zaczerpnięte z (Łabiak 2001),

- Washer – kontroler pralki automatycznej, opis wykorzystujący zależności czasowe (patrz rys. 3.3).

Dla każdego przykładu przygotowano jego dwie reprezentacje: sieć Petriego oraz odpowiedni model SynCharts, przy czym starano się zachować ich możliwie największe podobieństwo topologiczne przy zapewnieniu jednakowego działania. Wyjątek stanowi przykład WWatch, dla którego wykorzystano oryginalny zapis w języku ESTEREL.

Dla celów porównawczych testy przeprowadzono z wykorzystaniem w pełni profesjonalnego narzędzia ESTEREL, akademickiego narzędzia POLIS (opracowanego na Uniwersytecie w Berkeley) oraz autorskiego pakietu ORION (patrz rozdział 6.2). W dodatku B („Skrypty testowe") zawarto pełną informację o przeprowadzanych transformacjach na modelach przykładów testowych. W wyniku działania wyżej wzmiankowanych pakietów otrzymywano kompilowalne pliki zgodne ze standardem języka C, które następnie poddawano kompilacji z wykorzystaniem różnych narzędzi programistycznych, dla różnych docelowych systemów operacyjnych:

- Microsoft Visual Studio (platforma Windows2000),
- gcc (platforma Linux),
- Keil (platforma mikrokontrolerów przemysłowych rodziny MCS'51).

Jako podstawowy parametr charakterystyczny dla prowadzonych badań testowych przyjęto zajętość pamięci programu. Innym ważnym wyróżnikiem są czasy reakcji, jednak ze względu na zbyt duże różnice w przyjętych konwencjach rozwiązań uzyskanie wiarygodnych danych porównawczych staje się zadaniem bardzo złożonym.

W tabeli 5.1 zebrano informacje dotyczące najistotniejszych parametrów przykładów testowych, stanowiących o warunkach przeprowadzonych badań praktycznych.

Tab. 5.1. Zestawienie najistotniejszych parametrów przykładów testowych

Test	**Parametr**			
	Cmax / Lpw	Lm / Ls	Lmm / Lms	Lt
Cruise	12 / 12	18 / 20	8 / 8	16
DifFeq	5 / 6	20 / 34	1 / 6	12
Elliptic	4 / 8	41 / 53	1 / 11	31
WWatch	15 / --	38 / --	12 / --	32
DckCntr	10 / 17	30 / 36	6 / 20	28
Parker86	2 / 1	24 / 24	1 / 1	30
RS232	5 / 8	50 / 66	1 / 8	46
Count3	1 / 1	8 / 8	1 / 1	14
TVPilot	4 / 3	8 / 6	3 / 3	8
Washer	7 / 6	14 / 57	5 / 11	9

Przyjęto następujące oznaczenia:

- Cmax – współczynnik współbieżności dla sieci Petriego,
- Lpw – liczba procesów współbieżnych dla odpowiedniego modelu SyncCharts,
- Lm – liczba miesjc sieci Petriego,
- Ls – liczba stanów odpowiedniego modelu SyncCharts,

- Lmm – liczba makromiejsc sieci Petriego,
- Lms – liczba makrostanów odpowiedniego modelu SyncCharts,
- Lt – liczba tranzycji sieci Petriego.

W tabeli 5.2 zebrano wyniki testów przeprowadzonych dla wyżej opisanych przykładów. Wszystkie liczby przedstawiają rozmiar kodu wynikowego w bajtach, przy czym dla platform: Windows2000 oraz Linux przez kod wynikowy rozumie się plik „*.obj" a dla platformy MCS'51 ilość zajętej pamięci programu (na podstawie informacji zawartych w plikach raportowych „*.lst"). Wyniki te należy traktować jedynie orientacyjnie, ze względu na fakt, że każdy z pakietów wykorzystuje inny model specyfikacji formalnej, a tym samym powstałe różnice mogą częściowo wynikać z przyjętej metody opisu danych elementów zachowań (np. parametry czasowe, historia).

Tab. 5.2. Zestawienie wybranych testów

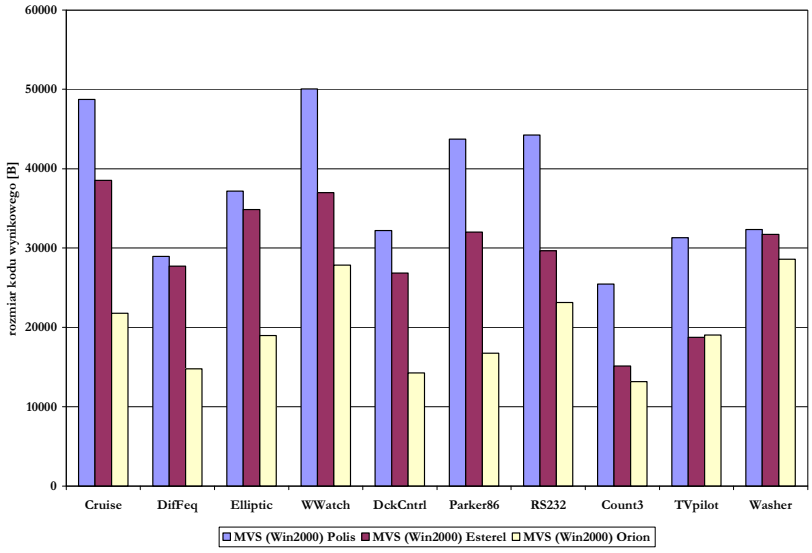
Test	**Rozmiar kodu wynikowego w bajtach**								
	MVS (Win2000)			GCC (Linux)			Keil (MCS'51)		
	Polis	Esterel	Orion	Polis	Esterel	Orion	Polis	Esterel	Orion
Cruise	48722	38538	21763	19482	27284	17846	--	8462	3502
DifFeq	28925	27696	14768	15337	15328	13699	--	4884	1115
Elliptic	37154	34848	18960	16166	20092	14117	--	6099	1090
WWatch	50042	36972	27848	20687	29468	19347	--	9868	4006
DckCntrl	32200	26854	14240	17527	22874	15634	--	2835	2018
Parker86	43725	32025	16729	18805	15708	14125	--	4638	1249
RS232	44248	29645	23140	18963	21685	14930	--	6749	1426
Count3	25435	15136	13139	15517	8000	15891	--	2618	1120
TVpilot	31298	18746	19047	17157	8716	15898	--	2634	2231
Washer	32341	31730	28595	17553	29064	19501	--	10020	4038

Brak danych w kolumnie Keil (MCS'51)/Polis wynika z faktu, iż pakiet ten generuje dedykowany system operacyjny dla konkretnej platformy programowej. Niestety autorzy pakietu udostępnili jedynie modele dwóch platform: UNIX (wykorzystywana w badaniach), oraz Motorola 68HC11.

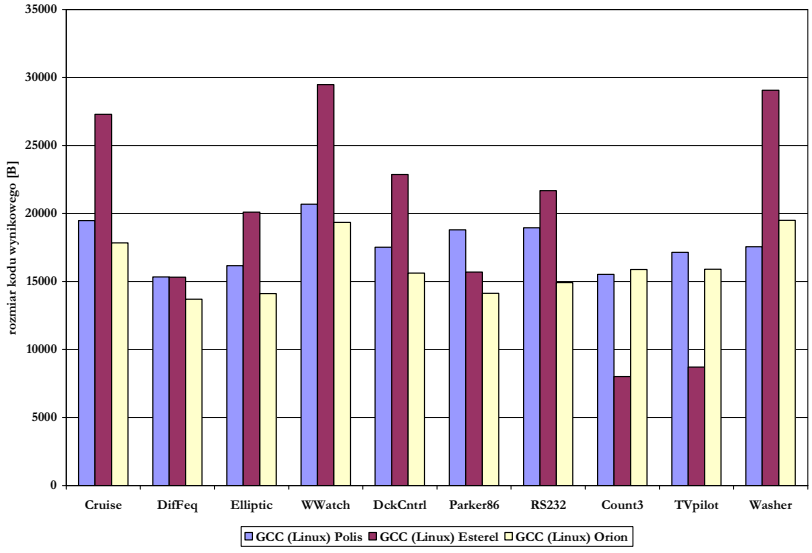
Na podstawie danych zawartych w tabelach 5.1 oraz 5.2 przygotowano szereg charakterystyk porównawczych. Na rys. 5.1 zestawiono zajętość pamięci programu przez kod wynikowy po kompilacji z użyciem narzędzia Microsoft Visual Studio, dla platformy Windows2000.

Zauważyć można, iż w każdym z badanych przykładów największym zużyciem pamięci cechują się pliki wygenerowane przez pakiet POLIS. Wyniki z autorskiego pakietu ORION wskazują na najniższą zajętość pamięci programu, z wyjątkiem przykładu TVPilot, dla którego są gorsze od wyników z pakietu ESTEREL o ok. 1,5%. Średnio wyniki z pakietu ORION są lepsze od wyników z pakietu ESTEREL o ok. 52%, a od wyników z pakietu POLIS aż o ok. 95%.

Na rys. 5.2 zestawiono zajętość pamięci programu przez kod wynikowy po kompilacji z użyciem standardowego kompilatora gcc, dla platformy Linux (dystrybucja Mandrake 7.2).



Rys. 5.1. Charakterystyki porównawcze dla kompilatora MVS

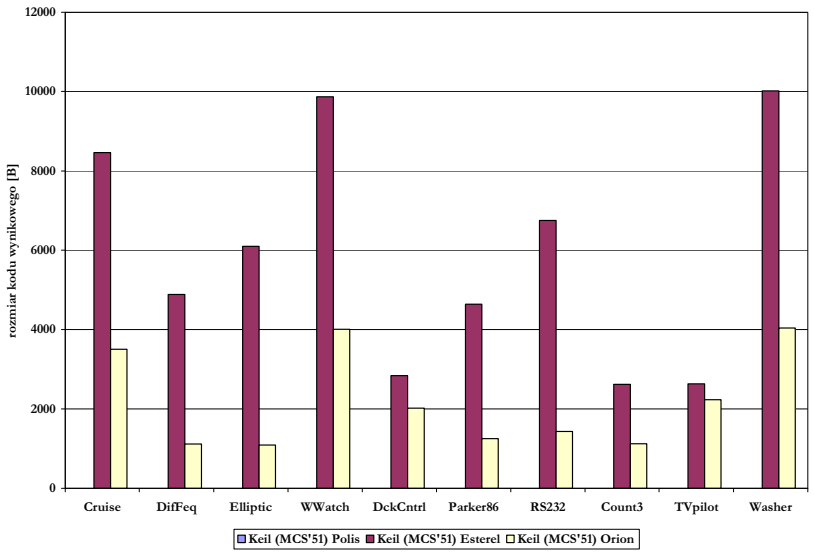


Rys. 5.2. Charakterystyki porównawcze dla kompilatora GCC

W tej serii badań zauważyć można większą różnorodność wyników. Stosunkowo najgorzej przedstawiają się wyniki uzyskane z pakietu ESTEREL. Wyniki z pakietu ORION wskazują na najniższą zajętość pamięci programu, z wyjątkiem przykładów TVPilot, Count3, oraz Washer, dla których są gorsze od wyników z pakietu ESTEREL

o ok. 47%, a od wyników z pakietu POLIS o ok. 6%. Średnie porównanie wyników wypada na korzyść pakietu ORION; jest lepszy od wyników z pakietu ESTEREL o ok. 22%, a od wyników z pakietu POLIS o ok. 11%.

Na rys. 5.3 zestawiono zajętość pamięci programu przez kod wynikowy po kompilacji z użyciem kompilatora µVision2 firmy Keil Elektronik GmBH, dla platformy MCS'51.



Rys. 5.3. Charakterystyki porównawcze dla kompilatora firmy Keil

Wyniki tych badań najkorzystniej przedstawiają się dla pakietu ORION. Ze względu na wzmiankowane wcześniej trudności przeprowadzenia syntezy dla tej platformy z wykorzystaniem pakietu POLIS, porównanie obejmuje tylko pakiet ESTEREL. Średnio wyniki z pakietu ORION są lepsze o ok. 200%.

W podsumowaniu można zauważyć, iż uzyskane wyniki syntezy programowej mogą wskazywać na wysoką efektywność opracowanego modelu programowego. Jednakże nie można wyeliminować pewnych czynników mających wpływ na uzyskane wyniki, takich jak: przyjęty model opisu behawioralnego, wykorzystanie szczególnych elementów opisu, optymalizacja modelu, itp.

Gorsze wyniki dla pakietu ORION, uzyskano w zasadzie tylko dla przykładów najprostszych. Jest to rezultatem dużego, stałego nakładu pamięci, wnoszonego przez system decyzyjny. Korzyści takiego rozwiązania są widoczne dla przykładów o nieco większym stopniu skomplikowania.

5.2. Wpływ topologii sieci na wyniki syntezy

Wszystkie przeprowadzone testy w tym podrozdziale wykonane zostały z wykorzystaniem narzędzia µVision2 firmy Keil Elektronik GmBH, dla

standardowego mikrokontrolera przemysłowego Intel8051, taktowanego z częstotliwością 12MHz. Ustawienia opcji kompilatora pozostawiono domyślne.

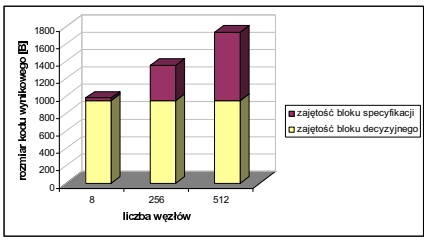
5.2.1. Rozmiar sieci

Celem badań jest określenie wpływu rozmiaru sieci (liczby węzłów), na rozmiar kodu wynikowego. Dla zminimalizowania wpływu innych czynników sieć testowa jest płaska, typu automatowego. Tabela 5.3 zawiera wyniki testów, przedstawiających zależność pomiędzy liczbą miejsc i tranzycji sieci, a zajętością pamięci programu (przez blok specyfikacji oraz blok decyzyjny).

Tab. 5.3. Wpływ rozmiaru sieci

Parametr	test_01	test_02	test_03
liczba miejsc	4	128	256
liczba tranzycji	4	128	256
całkowita zajętość pamięci programu	978 [B]	1350 [B]	1734 [B]
zajętość bloku specyfikacji	28 [B]	400 [B]	784 [B]
zajętość bloku decyzyjnego	950 [B]	950 [B]	950 [B]

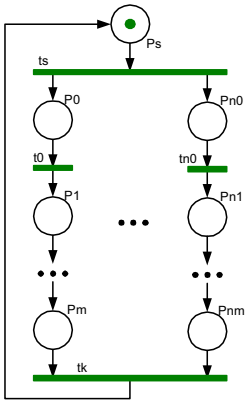
W badanym przedziale zmienności liczby węzłów (do 256 miejsc i 256 tranzycji), zajętość pamięci programu wykazuje liniową zależność od liczby węzłów (tab. 5.3, rys. 5.4).



Rys. 5.4. Zależność rozmiaru kodu wynikowego od liczby węzłów

5.2.2. Współczynnik współbieżności

Celem tych badań jest określenie wpływu współczynnika współbieżności sieci na rozmiar kodu wynikowego. Sieć testową przedstawiono na rys. 5.5, przy czym współczynnik m jest stały i wynosi 9, a w kolejnych testach zmianie ulega liczba gałęzi współbieżnych (współczynnik n) do 10.



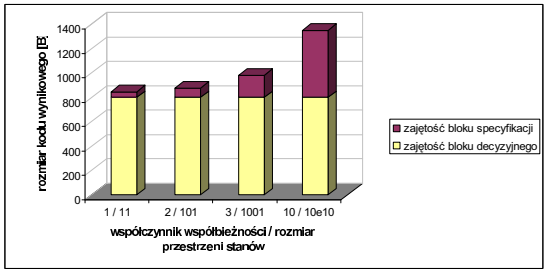
Rys. 5.5. Topologia sieci testowej

Tabela 5.4 zawiera dodatkowo informacje o rozmiarze wewnętrznej przestrzeni stanów sieci. Testowano sieci o rozmiarze wewnętrznej przestrzeni stanów rzędu 10^{10}. Dla badanej topologii jest możliwa synteza sieci o rozmiarze przestrzeni stanów dochodzącej nawet do 10^{38} (dla m=2, n=128 oraz liczby miejsc - 256).

Tab. 5.4. Wpływ współczynnika współbieżności

Parametr	**test_04**	**test_05**	**test_06**	**test_07**
liczba miejsc	11	21	31	101
liczba tranzycji	11	20	29	83
Współczynnik współbieżności	1	2	3	10
przestrzeń stanów	11	101	1001	10^{10}+1
całkowita zajętość pamięci programu	842 [B]	874 [B]	978 [B]	1349 [B]
zajętość bloku specyfikacji	44 [B]	76 [B]	180 [B]	551 [B]
zajętość bloku decyzyjnego	798 [B]	798 [B]	798 [B]	798 [B]

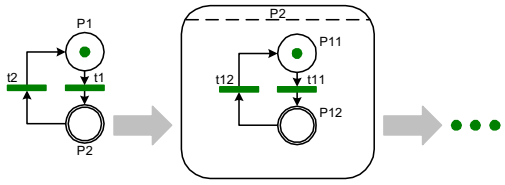
Na podstawie tabel 5.3 i 5.4, rys. 5.4 oraz rys. 5.6 można wnioskować, iż znaczący wpływ na zajętość pamięci programu ma liczba węzłów sieci, współczynnik współbieżności zaś wpływu takiego nie wykazuje.



Rys. 5.6. Zależność rozmiaru kodu wynikowego od współczynnika współbieżności

5.2.3. Hierarchia

Celem tych badań jest określenie wpływu wprowadzenia własności hierarchii na rozmiar kodu wynikowego. Sieć testową przedstawiono na rys. 5.7, przy czym w kolejnych testach zwiększeniu ulega liczba makromiejsc.



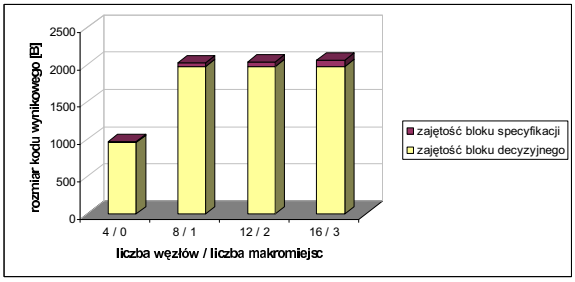
Rys. 5.7. Badanie własności hierarchii

W tabeli 5.5 zawarto wyniki testów dla sieci bez hierarchii (test_08), oraz z jednym, dwoma, oraz trzema dodatkowymi podsieciami (test_09 – test_11).

Tab. 5.5. Wpływ własności hierarchii

Parametr	test_08	test_09	test_10	test_11
liczba miejsc	2	4	6	8
liczba tranzycji	2	4	6	8
liczba makromiejsc	0	1	2	3
całkowita zajętość pamięci programu	972 [B]	2021 [B]	2041 [B]	2061 [B]
zajętość bloku specyfikacji	22 [B]	44 [B]	64 [B]	84 [B]
zajętość bloku decyzyjnego	950 [B]	1977 [B]	1977 [B]	1977 [B]

Zauważyć można prawie dwukrotny wzrost zajętości pamięci przez blok decyzyjny po wprowadzeniu hierarchii (test_09), po czym pozostaje ona stała dla kolejno wprowadzanych podsieci (rys. 5.8). Wynika to ze skomplikowania procedur decyzyjnych dla obsługi własności hierarchii.



Rys. 5.8. Zależność rozmiaru kodu wynikowego od liczby makromiejsc

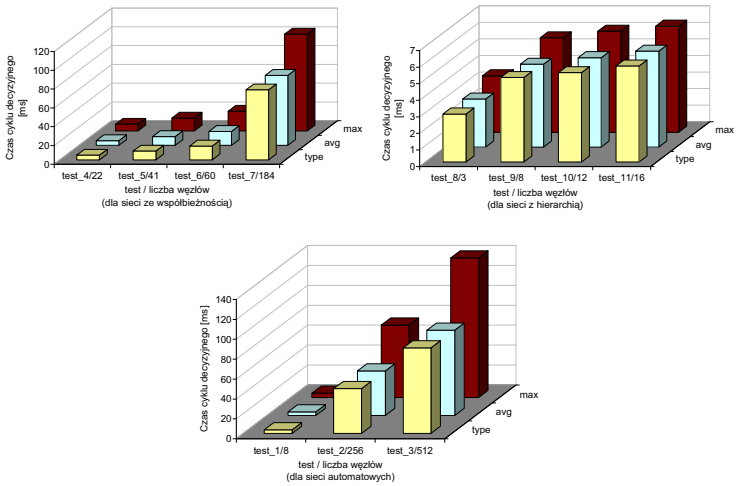
5.2.4. Czas wykonania cyklu decyzyjnego

Celem tych badań jest określenie wpływu rozmiaru sieci na czas wykonania cyklu decyzyjnego. W tabeli 5.6 zebrano wyniki testów dla sieci analizowanych w przykładach test_01 – test_11.

Wszystkie wyniki przedstawiają wartości średnie odpowiednich czasów, obliczonych na podstawie serii 10 testów dla każdej sieci. Przy czym czas t_{type} oznacza czas typowy cyklu decyzyjnego (tzn. najczęściej występujący), t_{avg} oznacza czas średni statystyczny, a t_{max} – czas maksymalny cyklu decyzyjnego. Testy wykonano z wykorzystaniem narzędzia Performance Analyzer, należącego do pakietu µVision2 firmy Keil Elektronik GmBH.

Tab. 5.6. Czasy wykonywania cyklu decyzyjnego

Test	liczba węzłów	t_{type} [ms]	t_{avg} [ms]	t_{max} [ms]
test_01	8	3,6	3,6	4,5
test_02	256	45	44,8	72,8
test_03	512	85,9	85,6	140
test_04	22	5,1	5,1	7,2
test_05	41	9,5	9,5	13,6
test_06	60	14,8	14,8	21
test_07	184	74,6	74,6	103
test_08	3	2,9	2,9	3,4
test_09	8	5,1	5	5,7
test_10	12	5,4	5,4	6,1
test_11	16	5,8	5,8	6,4



Rys. 5.9. Czasy wykonania cyklu decyzyjnego

Na rys. 5.9 przedstawiono wyniki analizy w postaci odpowiednich wykresów, przyjmując dla lepszej wizualizacji problemu ich rozdzielenie zgodnie z wcześniej badanymi klasami sieci. Na ich podstawie można wysunąć wnioski:

- dla sieci płaskich czas cyklu decyzyjnego jest w przybliżeniu liniowo zależny od liczby węzłów sieci,
- znaczące wydłużenie cyklu decyzyjnego następuje w momencie wprowadzenia do opisu elementu hierarchii (zestaw dodatkowych funkcji).

5.3. Podsumowanie i wnioski

W rozdziale niniejszym przedstawiono wyniki badań porównawczych, uzyskanych podczas syntezy programowej wybranych przykładów testowych, z wykorzystaniem pakietów wspomagających projektowanie: POLIS, ESTEREL oraz autorskiego ORION. Jako naczelne kryterium porównawcze przyjęto zajętość pamięci programu przez pliki wynikowe po procesie syntezy. Przeprowadzono także podstawowe badania wprowadzonego modelu, dla określenia zależności zajętości pamięci programu od jego wybranych własności: topologii, współczynnika współbieżności i hierarchii.

Podsumowując można wskazać najważniejsze wnioski praktyczne:

- proponowana metoda programowej realizacji cyfrowych układów sterowania wykazuje istotne korzyści pod względem obciążenia pamięci programu w stosunku do porównywanych, alternatywnych narzędzi akademickich i profesjonalnych, dla każdej z rozpatrywanych platform implementacyjnych,
- najkorzystniej metoda przedstawia się dla struktur mikrokontrolerów przemysłowych (różnice o aż ok. 200% zajętości pamięci programu, w stosunku do pakietu ESTEREL),
- w prostych strukturach mikrosystemów cyfrowych (o pamięci programu nie przekraczającej 64KB) można realizować systemy sterowania o przestrzeni wewnętrznej stanów nawet dochodzącej do rzędu 10^{38}.

Ze względu na fakt, iż w strukturach mikrosystemów cyfrowych najczęściej integrowane są rdzenie mikrokontrolerów, tak więc proponowana metoda realizacyjna może zapewnić wymierne korzyści przy realizacji w nich współbieżnych układów sterowania binarnego.

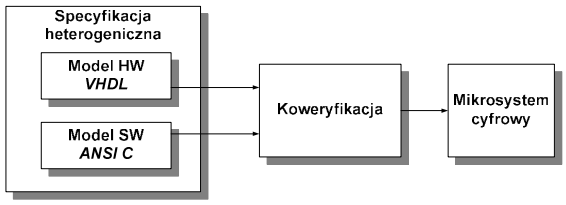
Rozdział 6

ŚRODOWISKO PROJEKTOWE DO SYNTEZY MIKROSYSTEMÓW CYFROWYCH

W niniejszym rozdziale opisane zostały typowe narzędzia wspomagające projektowanie cyfrowych układów sterowania, w kontekście opracowywanego pakietu wspomagającego zintegrowane projektowanie sprzętu i oprogramowania. Szczególną uwagę poświęcono autorskiemu modułowi syntezy programowej ORION.

6.1. Koncepcja środowiska projektowego

Istniejące narzędzia wspomagające projektowanie zintegrowane z wykorzystaniem struktur układowych mikrosystemów cyfrowych, mają w ogólności heterogeniczna strukturę, zbliżoną do przedstawionej na rys. 6.1.

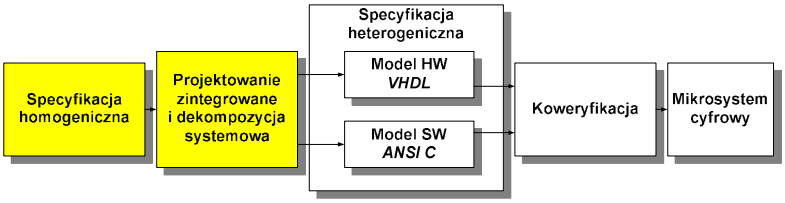


Rys. 6.1. Struktura typowych narzędzi do projektowania mikrosystemów cyfrowych

Część sprzętowa projektu opisywana jest z wykorzystaniem języka opisu sprzętu (najczęściej VHDL), a programowa z wykorzystaniem standardu języka C. Obie części powstają oddzielnie, a współprojektowanie zaczyna się od definiowania interfejsu komunikacji. Następnie prowadzona jest wspólna weryfikacja, nazywana w literaturze koweryfikacją. Jej wyniki mogą zostać użyte do ewentualnych poprawek w opisie projektowanych modułów, a opisywany cykl trwa do momentu spełnienia założonych wymagań projektowych.

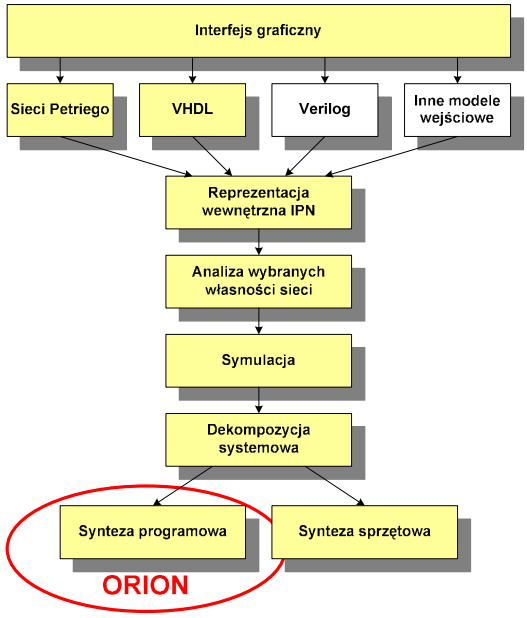
Metodyka taka pozostawia miejsce na rozszerzenie możliwości narzędzi ją stosujących, o opracowanie nakładki, pozwalającej na homogeniczny opis systemu sterowania (wykorzystujący jeden model specyfikacji formalnej), oraz przeprowadzenie dekompozycji systemowej, w wyniku której utworzone zostaną

wymagane wejściowe specyfikacje modułów (a więc VHDL oraz C). Ilustracja znajduje się na rys. 6.2.



Rys. 6.2. Ilustracja rozszerzenia możliwości opisowych systemów heterogenicznych

W ośrodku zielonogórskim trwają prace nad przygotowaniem kompleksowego narzędzia wspomagającego zintegrowane projektowanie sprzętu i oprogramowania. Uproszczony schemat blokowy przedstawiono na rys. 6.3.



Rys. 6.3. Środowisko zintegrowanego projektowania mikrosystemów cyfrowych

W chwili obecnej są zrealizowane (lub w fazie testów) następujące bloki funkcjonalne:

- interfejs graficzny (Łabiak i Andrzejewski 1999, Skowroński 2000),
- konwerter VHDL do interpretowanych sieci Petriego (Skowroński 2000),
- analizator wybranych własności sieci (Andrzejewski i Łabiak 1999, Biliński 1996),
- moduł symulacji (Dzikuć i Skowroński 1998, Skowroński 2000),

- moduł syntezy sprzętowej (Andrzejewski i Łabiak 1999, Biliński 1996, Kozłowski 1996, Wolański 1998).

W ramach prac badawczych wykonano moduł syntezy programowej, o nazwie roboczej ORION.

Pozostałe bloki funkcjonalne, a więc: konwertery innych modeli formalnych do sieci Petriego, oraz moduł dekompozycji systemowej są w chwili obecnej we wstępnym etapie projektowania.

6.2. Moduł syntezy programowej

Moduł syntezy programowej ORION opracowany został jako osobny program pracujący w trybie konsoli. W chwili obecnej umożliwia on generowanie kodu realizacyjnego zgodnego ze standardem ANSI C oraz dla kompilatorów Keil Software, Inc. (platforma MCS'51).
Złożony jest on z trzech podmodułów, realizujących następujące funkcje:

- analiza składni hpn,
- analiza statystyczna sieci,
- synteza programowa.

Na potrzeby tekstowej reprezentacji interpretowanych sieci hierarchicznych Petriego, opracowany został format hpn, szczegółowo opisany w dodatku A („Gramatyka HPN"). Jest on wykorzystywany jako format wejściowy do pakietu ORION, umożliwiając w ten sposób niezależne testowanie i wykorzystywanie tego pakietu.

Moduł analizy statystycznej pozwala na określenie wielu istotnych parametrów sieci. Najważniejsze przedstawiono poniżej:

- liczba miejsc,
- liczba tranyzcji,
- liczba makromiejsc, wyznaczających ilość podsieci,
- liczba sygnałów wejściowych, wyjściowych oraz lokalnych,
- użycie łuków zabraniających lub zezwalających,
- wykorzystanie etykiet action, cond lub abort przypisanych do tranzycji,
- użycie parametrów czasowych, itp.

Na podstawie tych informacji moduł syntezy optymalizuje automatycznie model programowy, eliminując nie wykorzystywane elementy, dopasowuje do niego strukturę funkcyjną bloku decyzyjnego, dobiera właściwe typy zmiennych do przechowywania informacji w strukturach danych oraz wyznacza niezbędne wielkości tych struktur.

Końcowym efektem działania modułu syntezy programowej jest wygenerowanie kompilowalnych plików, zawierających kompletny zestaw funkcji bloku decyzyjnego, funkcję obsługi zmiennych czasowych wraz z informacją o częstotliwości jej wywoływania (o ile jest potrzebna), oraz zainicjowane struktury danych bloku specyfikacji i stanu systemu.

Rozdział 7

PODSUMOWANIE I WNIOSKI

Praca ma charakter teoretyczny, przeglądowy oraz praktyczny. W części teoretycznej opracowano nowy, alternatywny do innych model hierarchicznych sieci Petriego HPN, integrujący w sobie wiele elementów opisu używanych w różnych klasach sieci płaskich (np. łuki zabraniające, zezwalające, parametry czasowe, itp.) oraz wybrane elementy zaczerpnięte ze standardu UML (np. atrybut historii). Elementy te upraszczają opis sytuacji trudnych do zamodelowania z wykorzystaniem tylko sieci płaskich. Jednocześnie przy pominięciu dodatkowych, rzadziej spotykanych elementów model automatycznie sprowadza się do zgodnego z częściej stosowanymi interpretowanymi sieciami płaskimi. W części teoretycznej przedstawiono także nowatorską koncepcję realizacji programowej interpretowanych sieci Petriego, wykorzystującą teoriomnogościową reprezentację sieci, nazywaną modelem programowym PHPN. Model ten definiowany jest w abstrakcyjnym środowisku systemu decyzyjnego, który na podstawie danych specyfikacji systemu oraz jego stanu aktualnego podejmuje decyzje o przeprowadzeniu systemu do stanu kolejnego lub pozostawieniu go w stanie bieżącym. Rozwiązanie to jest znacznym rozszerzeniem propozycji wirtualnego systemu decyzyjnego VFSM, użytego do realizacji automatów FSM, opisywanego w pracy (Wagner 1994).

W części przeglądowej skonfrontowano opracowany model z podobnymi modelami znanymi z literatury. Zebrano także i usystematyzowano znane metody implementacyjne interpretowanych sieci Petriego, oraz najistotniejsze metody realizacji innych modeli. W części praktycznej pracy zrealizowano oprogramowanie, o roboczej nazwie ORION, pozwalające w sposób automatyczny przygotowywać programy realizacji układów sterowania cyfrowego danych w postaci interpretowanych, hierarchicznych sieci Petriego.

Motywacją podjęcia tematyki badań była potrzeba opracowania narzędzia wspomagającego automatyczną syntezę programową sieci w tworzonym na Uniwersytecie Zielonogórskim pakiecie oprogramowania wspomagającym projektowanie zintegrowane sprzętu i oprogramowania.

Najbardziej znaczącym rezultatem prac jest sformułowanie ogólnych, przydatnych w praktyce zasad syntezy programowej interpretowanej sieci Petriego, pozwalających na stosunkowo łatwą realizację praktyczną systemów sterowania binarnego na platformie programowej, z wykorzystaniem języków wysokiego

poziomu. Dla rozszerzenia zakresu stosowalności przyjętego rozwiązania, opracowano algorytm dekompozycji systemowej interpretowanych sieci Petriego, umożliwiający wyodrębnianie z sieci takich fragmentów, które dla spełnienia żądanych kryteriów czasowych wymagają realizacji sprzętowej.

Przeprowadzono szereg testów praktycznych, w których dokonuje się porównania wyników syntezy programowej różnych pakietów wspomagających projektowanie zintegrowane (POLIS, ESTEREL) z pakietem autorskim ORION. Przeprowadzono także badania nad wpływem rozmiaru sieci, jej topologii a także użycia wybranych elementów opisu na rozmiar kodu wykonywalnego oraz czasy trwania cyklu decyzyjnego. Badania przeprowadzono dla różnych platform operacyjnych (Linux, WinNT, MCS'51), z wykorzystaniem różnych narzędzi do kompilacji (Microsoft Visual Studio C++, gcc, Keil Software, IAR). Wyniki badań wskazują, iż użycie programowego modelu interpretowanych sieci Petriego ułatwia procedury syntezy programowej układów sterowania binarnego, a także pozwala na osiągnięcie istotnych korzyści w sensie ograniczenia zajętości zasobów pamięciowych. Pozwalają także sądzić, iż proponowana metoda realizacyjna nie musi ograniczać się do mikrosystemów cyfrowych, ale może być wykorzystana znacznie szerzej, ze względu na jej niezależność od platformy realizacyjnej (dotyczy platformy programowej zdefiniowanej jak w rozdz. 1.4).

Praca ma charakter otwarty. Można wskazać kilka kierunków dalszych prac badawczych:

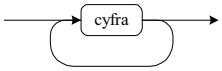
- opracowanie algorytmów analizy wybranych własności hierarchicznych sieci Petriego,
- rozszerzenie możliwości graficznej reprezentacji sieci (np. definiowanie magistral, wsparcie opisu strukturalnego),
- optymalizacja algorytmu decyzyjnego,
- wykorzystanie innych języków wysokiego poziomu do realizacji praktycznej (np. Java),
- wykorzystanie modelu programowego jako jądra symulacyjnego w narzędziach weryfikacji,
- opracowanie metod konwersji pomiędzy sieciami hierarchicznymi a innym modelami formalnymi (np. Statecharts).

Programowa realizacja układów sterowania cyfrowego, z wykorzystaniem modelu sieci hierarchicznych i opracowanego oprogramowania, była tematem zajęć laboratoryjnych z przedmiotu Reaktywne Systemy Cyfrowe. Wyniki ćwiczeń pozwoliły na częściową weryfikację praktyczną metody, potwierdzając jednocześnie efektywność modelowania systemów o różnorodnych wymaganiach opisowych.

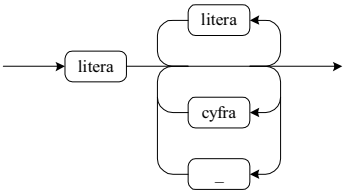
Wyniki badań były częściowo publikowane w materiałach konferencyjnych (Andrzejewski i Łabiak 1999, Łabiak i Andrzejewski 1999, Andrzejewski 2000a, Andrzejewski 2000b, Andrzejewski 2001a, Ciesłowki i Andrzejewski 2001, Andrzejewski 2001b, Andrzejewski 2001c, Andrzejewski 2001d, Andrzejewski 2001e, Andrzejewski 2002).

Dodatek A – Gramatyka hpn – diagramy składni

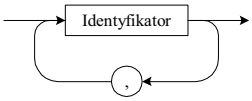
1. Liczba



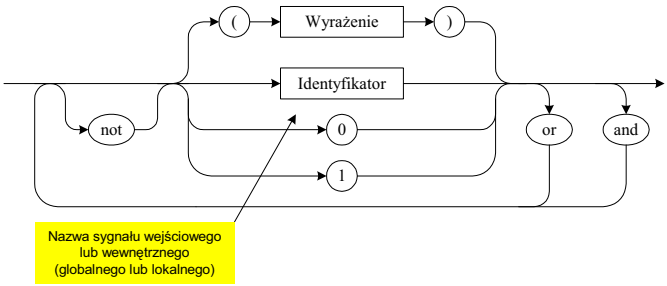
2. Identyfikator



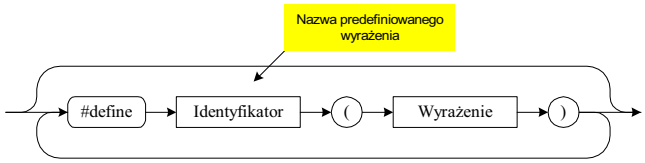
3. Lista identyfikatorów



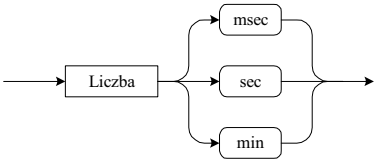
4. Wyrażenie



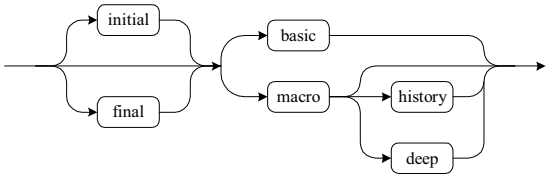
5. Predefinicja



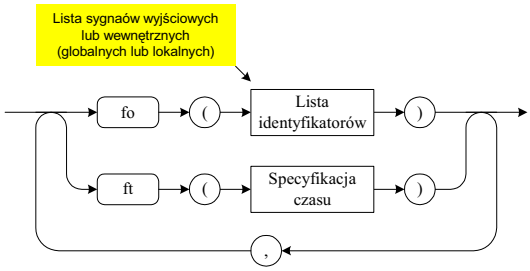
6. Specyfikacja czasu



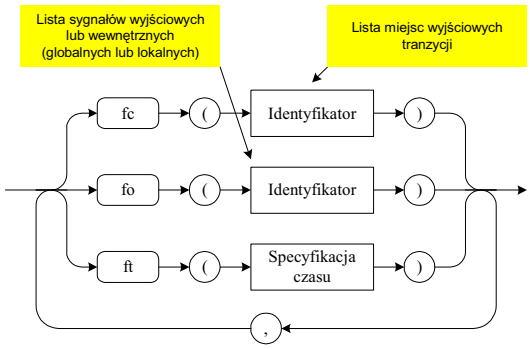
7. Typ miejsca



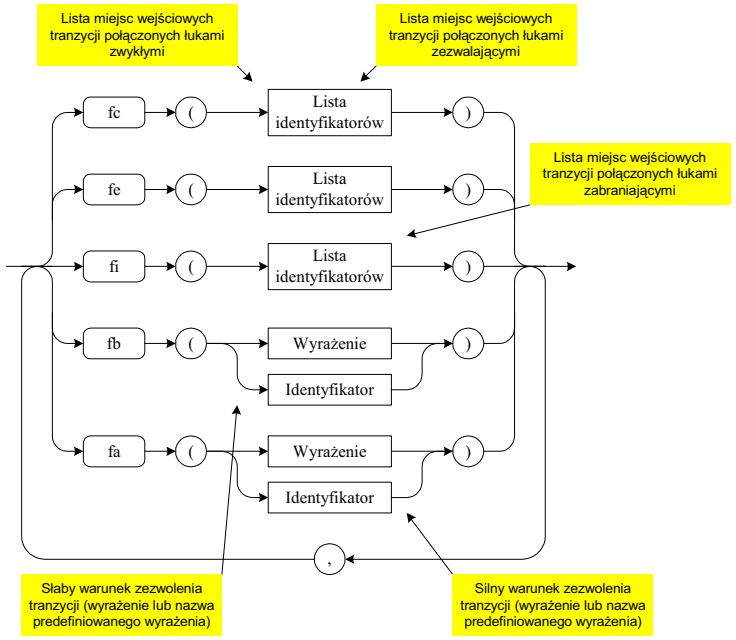
8. Akcja miejsca



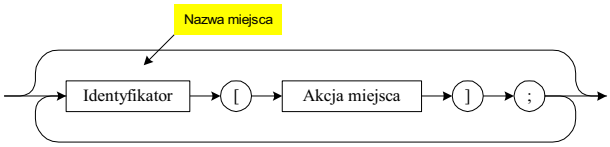
9. Akcja tranzycji



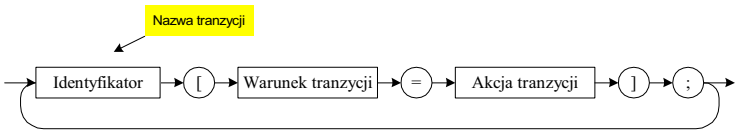
10. Warunek tranzycji



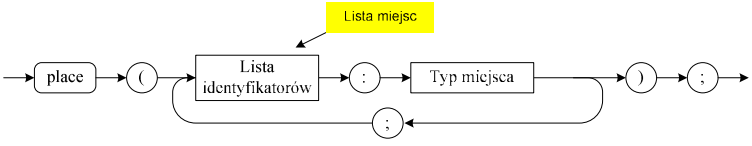
11. Definicja miejsca



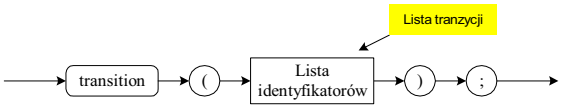
12. Definicja tranzycji



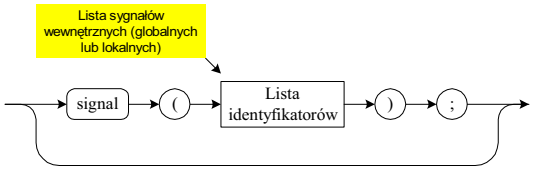
13. Deklaracja miejsca



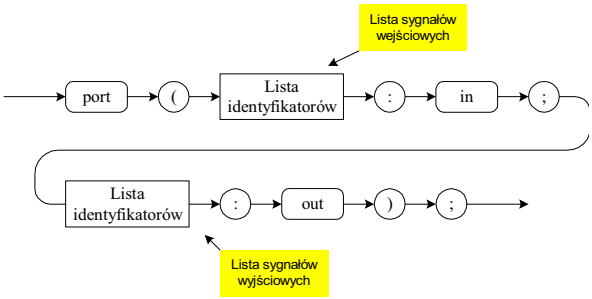
14. Deklaracja tranzycji



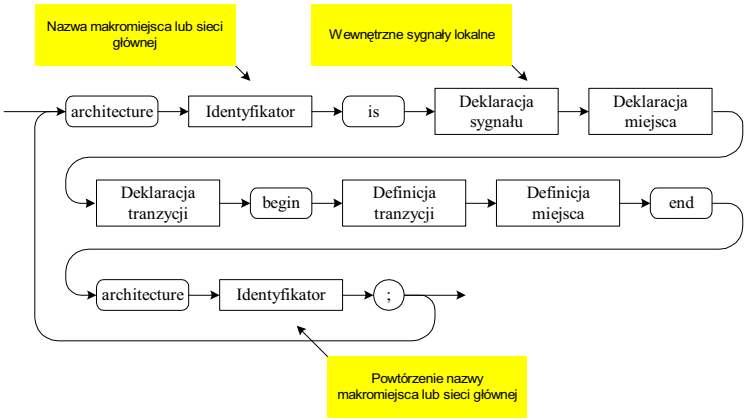
15. Deklaracja sygnału



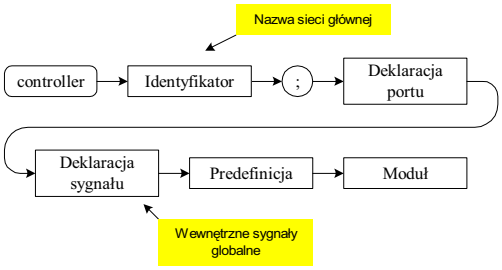
16. Deklaracja portu



17. Moduł



18. Ogólna struktura formatu hpn



Uwaga: Kolejność definiowania poszczególnych modułów wynika z diagramu hierarchii, tzn. aby zdefiniować dowolny moduł muszą być wcześniej zdefiniowane wszystkie moduły podrzędne. Ostatnim modułem w opisie jest moduł reprezentujący sieć podstawową.

Opis sieci hierarchicznej w formacie *hpn* dla przykładu z rys. 1.15:

controller coffee-maker;
port (power-on, power-off, cup-empty, coffee-empty, ready, result, coin, return, make, cup-ready, coffee-ready: **in**; empty, coin-to-return, light-make, light-return, cup-pr, coffe-pr, light-busy, coin-to-bank: **out**);
#define c1 (cup-empty **or** coffee-empty)
#define c2 (ready **and** result)
#define c3 (ready **and not** result)

architecture ON **is**
place (STANDBY: **initial basic**; TEST, READY, CUP-BUSY, COFFE-BUSY, LIGHT-COIN: **basic**);
transition (T1,T2,T3,T4,T5,T6,T7);
begin
T1[**fc**(STANDBY),**fb**(coin)=**fc**(TEST)];
T2[**fc**(TEST),**fb**(c3)=**fc**(STANDBY)];
T3[**fc**(TEST),**fb**(c2)=**fc**(READY)];
T4[**fc**(READY),**fb**(make)=**fc**(CUP-BUSY,LIGHT-COIN)];
T5[**fc**(CUP-BUSY),**fb**(cup-ready)=**fc**(COFFEE-BUSY)];
T6[**fc**(COFFE-BUSY,LIGHT-COIN),**fb**(coffee-ready)=**fc**(STANDBY)];
T7[**fc**(READY),**fb**(return)=**fc**(STANDBY)];
STANDBY[**fo**(coin-to-return)];
READY[**fo**(light-make,light-return)];
CUP-BUSY[**fo**(cup-pr),**ft**(1 sec)];
COFFEE-BUSY[**fo**(coffee-pr),**ft**(5 sec)];
LIGHT-COIN[**fo**(light-busy,coin-to-bank)];
end architecture ON;

architecture coffee-maker **is**
place (OFF: **initial basic**; ON: **macro**; EMPTY: **basic**);
transition (T1,T2,T3,T4);

```
begin
T1[fc(OFF),fb(power-on)=fc(ON)];
T2[fc(ON),fb(power-off)=fc(OFF)];
T3[fc(ON),fb(c1)=fc(EMPTY)];
T4[fc(EMPTY),fb(not c1)=fc(ON)];
EMPTY[fo(empty)];
end architecture coffee-maker;
```

Opis sieci hierarchicznej w formacie *hpn* dla przykładu z rys. 3.3:

```
controller washer;
port (start,L1,L2,TL1,TL2: in; CR,CL,H,V1,V2: out);
signal (s1);
#define c1 (L2 and (not TL1))
#define c2 ((not L2) or TL2)

architecture P3 is
place (P11: initial basic; P12: final basic);
transition (T8);
begin
T8[fc(P11),fb(L1)=fc(P12)];
P11[fo(V1)];
end architecture P3;

architecture P4 is
place (P9: initial basic; P10: basic);
transition (T6,T7);
begin
T6[fc(P9),fb(c1)=fc(P10)];
T7[fc(P10),fb(c2)=fc(P9)];
P10[fo(H)];
end architecture P4;

architecture P5 is
place (P13: initial basic; P14: basic);
transition (T9,T10);
begin
T9[fc(P14)=fc(P13),ft(5 sec)];
T10[fc(P13)=fc(P14),ft(5 sec)];
P13[fo(CR),ft(10 sec)];
P14[fo(CL),ft(10 sec)];
end architecture P5;

architecture P6 is
place (P15: initial basic; P16: final basic);
transition (T11);
begin
T11[fc(P15)=fc(P16)];
P15[ft(280 sec)];
P16[fo(s1)];
end architecture P6;

architecture P2 is
place (P3,P4: initial macro; P5: macro; P6: history macro; P7: basic; P8: final basic);
transition (T3,T4,T5);
```

begin
T3[**fc**(P3)=**fc**(P5,P6)];
T4[**fc**(P4,P5,P6),**fa**(s1)=**fc**(P7)];
T5[**fc**(P7),**fb**(L3)=**fc**(P8)];
P7[**fo**(V2)];
end architecture P2;

architecture washer **is**
place (P1: **initial basic**; P2: **history macro**);
transition (T1,T2);
begin
T1[**fc**(P1),**fb**(start)=**fc**(P2)];
T2[**fc**(P2),**fa**(**not** start)=**fc**(P1)];
end architecture washer;

Dodatek B – Skrypty testowe

1. Skrypt testowy dla pakietu ESTEREL:

```
ESTEREL=/home/user/cads/esterel
PATH=$PATH:$ESTEREL/bin
esterel <nazwa_pilku.strl>
gcc –c <nazwa_pilku.c>
```

Jako rezultat otrzymuje się plik w formacie <nazwa_pilku.c> oraz plik objektowy <nazwa_pilku.o>.

2. Skrypt transformacji formatu strl na oc:

```
ESTEREL=/home/user/cads/esterel
PATH=$PATH:$ESTEREL/bin
esterel –oc <nazwa_pilku.strl>
```

Jako rezultat otrzymuje się plik wsadowy do dalszych konwersji <nazwa_pilku.oc>

3. Skrypt transformacji formatu oc na shift:

```
./oc2shift –o <nazwa_pilku.shift> <nazwa_pilku.oc>
```

Jako rezultat otrzymuje się plik wsadowy do pakietu POLIS <nazwa_pilku.shift>

4. Skrypt uruchomieniowy pakietu POLIS:

```
POLIS=/home/user/cads/polis
PATH=$PATH:$POLIS/bin
polis
```

5. Skrypt testowy pakietu POLIS (po uruchomieniu pakietu):

```
read_shift /home/user/<nazwa_pliku.shift>
propagate_const
set_impl –s
partition
set arch unix
build_sg
sg_to_c –d /home/user
gen_os –D /home/user/cads/polis/polis_lib/os –d /home/user
```

Jako rezultat otrzymuje się pliki w formacie <os.c> (system operacyjny) oraz <z_nazwa_pilku_0.c> (plik pomocniczy).

Literatura

Adamski M. (1990): *Digital system design by formal transformation of specification*. – Prep.35 Int. Wissen. Koll., TH Ilmenau, Germany, Heft 3, pp. 62-65.

Adamski M. (1992): *Projektowanie Układów Cyfrowych Systematyczną Metodą Strukturalną*. – Zielona Góra: Wydawnictwo WSI.

Adamski M. (2000): *Bezpośrednia implementacja sieci Petriego w reprogramowalnych układach cyfrowych*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'2000*, Szczecin, ss. 131-138.

Adamski M. (2001): *Programowa sieć Petriego jako model formalny mikrosystemu cyfrowego*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'2001*, Szczecin, ss. 139-146.

Andre C. (1996): *Examples of Synccharts*. – Technical Report RR.96, Institut National de Recherche en Informatique et en Automatique, I3S, Sophia-Antipolis, France.

Andrzejewski G., Łabiak G. (1999): *Eksperymentalny system do syntezy kontrolerów współbieżnych*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'99*, Szczecin, ss. 43-49.

Andrzejewski G. (2000a): *Programowa implementacja sieci Petriego w sterownikach mikrokomputerowych*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'2000*, Szczecin, ss. 139-145.

Andrzejewski G. (2000b): *Projektowanie kontrolerów współbieżnych z wykorzystaniem programowego modelu interpretowanej sieci Petriego*. – Mat. Kraj. Konf. Nauk. *Ogólnopolskie Warsztaty Doktoranckie: OWD'2000*, Istebna-Zaolzie, ss. 63-68.

Andrzejewski G. (2001a): *Dekompozycja systemowa w zintegrowanym projektowaniu sprzętu i oprogramowania*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'2001*, Szczecin, ss. 117-124.

Andrzejewski G. (2001b): *Timed Petri nets for software applications*. – Proc. Int. Work. *Discrete - Event System Design: DESDes'01*, Przytok k/Zielonej Góry, Polska, pp. 73-78.

Andrzejewski G. (2001c): *Hierarchical Petri Net as a Representation of Reactive Behaviors*. – Proc. Int. Conf. *Advanced Computer Systems: ACS'2001*, Szczecin, Polska, Part 2, pp. 145-154.

Andrzejewski G. (2001d): *Synteza programowa w kontekście zintergowanego projektowania systemow sterowania*. – Mat. Kraj. Konf. Nauk. *Ogólnopolskie Warsztaty Doktoranckie: OWD 2001*, Istebna-Zaolzie, 2001, ss. 81-86.

Andrzejewski G. (2001e): *Program model of Petri net*. – Proc. Int. Conf. *Computer - Aided Design of Discrete Devices: CAD DD 2001*, Minsk, Belarus, 2001, Vol. 1, pp. 87-92.

Andrzejewski G. (2002): *Synchronizacja procesów sterujących zdekomponowanych w hybrydowych strukturach mikrosystemów cyfrowych*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'2002*, Szczecin, ss. 137-143.

Augin M., Boeri F., Andre C. (1980): *Systematic method of realization of interpreted Petri nets*. – Digital Processes, Vol.6, pp. 55-68.

Balarin F. (Ed) (1997): *Hardware-Software Co-Design of Embedded Systems. The POLIS Approach*. – Kluwer Academic Publishers.

Banaszak Z., Kuś J., Adamski M. (1993): *Sieci Petriego. Modelowanie, Sterowanie i Synteza Procesów Dyskretnych*. – Zielona Góra: Wydawnictwo WSI.

Baranowski J. (1981): *Metody syntezy układów cyfrowych opisanych siecią Petri*. – rozprawa doktorska, Gliwice, Politechnika Śląska.

Belhadj H., Gerbaux L., Bertrand M., Saucier G. (1993): *Specification and Synthesis of Communicating Finite State Machine*. – Synthesis for Control Dominated Circuits, Elsevier Science Publishers B.V., North-Holland.

Berry G. (1991): *Programming a Digital Wristwatch in ESTEREL v3.2*. – Technical Report RR.8, Centre de Mathématiques Appliquées, Ecole des Mines de Paris.

Berry G. (1998): *The Foundation of Esterel*. – To appear in Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, MIT Press.

Berthomieu B., Diaz M. (1991): *Modeling and Verification of Time Dependent Systems Using Time Petri Nets*. – IEEE Transaction on Software Engineering, Vol. 17, No. 3.

Biliński K. (1996): *Application of Petri nets in parallel controler design*. – PhD thesis, University of Bristol, UK.

Bolton M. (1990): *Digital Systems Design with Programmable Logic*. – Addison-Wesley Publishing Company.

Brener F., Gajski D. (1990): *Chippe: A System for Constraint Driven Behavioral Synthesis*. – IEEE Trans. CAD, Vol. 9, No. 7.

Ciesłowski P., Andrzejewski G. (2001): *Dynamiczna realizacja sieci Petriego*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'2001*, Szczecin, ss. 125-130.

David R., Alla H. (1992): *Petri Nets & Grafcet. Tools for Modelling Discrete Event Systems*. – New York: Prentice Hall.

Dzikuć T., Skowroński Z. (1998): *Symulator czasowo-funkcjonalny do specyfikacji systemów cyfrowych zadanych w postaci interpretowanych sieci Petriego*. – Mat. Kraj. Konf. Nauk. *Międzynarodowe Sympozjum Naukowe Studentów i Młodych Pracowników Nauki: MSN'20*, Zielona Góra, Polska, T.3: Informatyka, ss. 295-301.

Edwards S., Lavagno L., Lee E.A., Sangiovanni-Vincentelli A (1997): *Design of Embedded Systems: Formal Models, Validation, and Synthesis*. – Proc. IEEE, Vol. 85, No. 3, pp. 366-390.

Esser R. (1996): *An object oriented Petri net approach to embedded systems design*. – PhD thesis, Swiss Federal Institute of Technology, Zurich.

Fernandes J.M., Adamski M., Proenca A.J.: *VHDL Generation from Hierarchical Petri Net Specifications of Parallel Controllers*. – Proc IEEE: Computers and Digital Techniques, Vol. 144, No. 2, pp. 127-137.

Fornaciari W., Gubian P., Sciuto D., Silvano C. (1998): *Power estimation of embedded systems: a hardware/software codesign approach*. – IEEE Transaction on VLSI Systems, Vol. 6, No. 2, pp. 266-275.

Gajski D. (1997): *Principles of Digital Design*. – Prentice Hall.

Gajski D. (2001): *SpecC: Specification Language and Methodology*. – Kluwer Academic Publishers.

Gallier J.H. (1986): *Logic for Computer Science*. – New York: Foundations of Automatic Theorem proving. Harper & Row Publishers.

Harel D. (1987): *Statecharts: A Visual Formalism for Complex Systems*. – Science of Computer Programming, North-Holland, no 8, pp. 231-274.

Heiner M. (1998): *Petri net based system analysis without state explosion*. – Proc. Int. Conf. *High Performance Computing'98*, Boston, session "Petri net Applications and HPC", pp. 394-403.

Hermann D., Henkel J., Ernst R. (1994): *An approach to the adaptation of estimated cost parameters in the Cosyma system*. – CASHE, Grenoble, IEEE CS Press, September, pp. 100-107.

Holvoet T., Verbaeten P. (1995): *Petri Charts: an Alternative Technique For Hierarchical Net Construction*. – Proc. IEEE Conf. on Systems, *Man and Cybernetics*.

Hong J.E., Bae D.H. (1998): *HOONets: Hierarchical Object-Oriented Petri Nets for System Modeling and Analysis*. – KAIST Technical Report CS/TR-98-132, November.

Hurk J., Jess J. (1998): *System Level Hardware/software Co-design. An Industrial Approach*. – Kluwer Academic Publishers.

Jensen K. (1997): *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. – Monographs in Theoretical Computer Science, Springer-Verlag.

Jerraya A.A., Mermet J. (Ed) (1999): *System-Level Synthesis*. – Kluwer Academic Publishers.

Johnsson C., Årzen K.E. (1994): *High-Level Grafcet and Batch Control*. – Proc. Int. Conf. *Automation of Mixed Processes: ADPM'94, Dynamical Hybrid Systems*, Bryssel.

Kalinowski J. (1984): *Wykorzystanie sieci Petriego do projektowania systemów cyfrowych*. – Rozprawa doktorska, Warszawa, Politechnika Warszawska.

Karatkievich A, Andrzejewski G. (2002): *Analiza wybranych własności interpretowanych sieci Petriego metodą optymalnej symulacji*. – Mat. Kraj. Konf. *Krajowa Konferencja Elektroniki: KKE'2002*, Kołobrzeg, t. II, ss. 685-690.

Kernighan B., Ritchie D. (1988): *The C Programming Language*. – Prentice Hall, Inc.

Kirovski D., Potkonjak M. (1997): *System-level synthesis of low-power hard real- time systems*. – Proc. Int. Conf. Design Automation Conference'1997, ACM Press, pp. 697-702.

Kozłowski T. (1996): *Petri-Net-Based CAD Tools for Parallel Controller Synthesis*. – M.Sc. Thesis, Bristol, University of Bristol.

Ku D., Micheli G. (1988): *Hardware C - a language for hardware design*. – Technical Report CSL-TR-88-362, Computer Systems Laboratory, Stanford University.

Kyeyune Y. (2000): *Developing Concepts and Methods for Module and Integration Tests for Models of Reactive Systems*. – Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften der Universität Dortmund am Fachbereich Informatik, Dortmund.

Li Y., Henkel J. (1998): *A framework for estimating and minimizing energy dissipation of embedded HW/SW systems*. – Proc. Int. Conf. Design Automation Conference'1998, ACM Press, pp. 188-193.

Łabiak G., Andrzejewski G. (1999): *Edytor graficzny sieci Petriego*. – Mat. Kraj. KOnf. Nauk. *Reprogramowalne układy Cyfrowe: RUC'99*, Szczecin, ss. 57-64.

Łabiak G. (2001): *Zastosowanie diagramów Statecharts w specyfikacji funkcjonalnej układów sterowania binarnego*. – Mat. Kraj. KOnf. Nauk. *Reprogramowalne układy Cyfrowe: RUC '2001*, Szczecin, ss. 55-60.

Magiollo-Schettini A., Merro M. (1996): *Priorities in Statecharts*. – Proc. Conf. LOMPAS'96, Stockholm, Sweden, Vol. 1192, pp. 404-429.

Majewski W. (1999): *Układy Logiczne*. – Wydawnictwa Naukowo-Techniczne.

Merlin P. (1974): *A study of the Recoverability of Computer Systems*. – Thesis, Department of Computer Science, University of California, Irvine.

Micheli G. (1998): *Synteza i Optymalizacja Układów Cyfrowych*. – Warszawa: WNT.

Miczulski P. (2001): *State space calculation algorithm of hierarchical Petri nets with application of decision diagrams*. – Proc. Int. Work. *Discrete - Event System Design: DESDes'01*, Przytok k/Zielonej Góry, Polska, ss. 67-72.

Minato S.I. (1996): *Binary Decision Diagrams and Applications for VLSI CAD*, – Boston: Kluwer Academic Publishers.

Mirkowski J., Skowroński Z. (1997): *Interpreted Petri nets as a formal representation of hardware/software systems*. – Proc. Int. Conf. MIXDES'97, Poznań, Polska, pp. 173-178.

Misiurewicz P. (1980): *Zagadnienia projektowania cyfrowych układów sterowania binarnego*. – Mat. Konf. Krajowej Konferencji Automatyki'80, Szczecin, ss. 664-670.

Murata T. (1989): *Petri Nets: Properties, Analysis and Applications*. – Proc. of the IEEE, Vol. 77, no 4, pp. 548-580.

Napieralski A. (Ed) (1998): *Mixed Design of Integrated Circuits and Systems*. – Kluwer Academic Publishers.

Orchad H.J. (1990): *Adjusting the Parameters in Elliptic-Function Filters*. – Proc. of the IEEE Trans. CAS, Vol. 37, No. 5.

Peterson J. (1981): *Petri Net Theory and Modeling of Systems*. - Prentice-Hall, Englewood Cliffs, NJ.

Petri C.A. (1962): *Kommunikation mit Automaten*. – PhD Thesis, Schriften des IIM Nr 3, Institut für Instrumentelle Mathematik, Bonn, Germany.

Ramamoorthy C.V., Ho G.S. (1980): *Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets*. – IEEE Transaction on Software Engineering, SE-6(5).

Ramchandani C. (1974): *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. – Massachusetts Institute of Technology, Project MAC, Technical Report 120.

Ratajczak J., Pabiś N. (1999): *CXL, a XML-based Language for the Description of Hierarchical Concurrent State Machines*. – ICS WUT Research Report 15/99.

Reisig W. (1988): *Sieci Petriego. Wprowadzenie*. – Warszawa: Wydawnictwa Naukowo-Techniczne.

Sami M., Courvoisier M. (1981): *A Petri net based programmable logic controller with on line testing capabilities*. – Proc. Int. Conf. IFAC'81.

Schof S., Sonnenschein M., Wieting R. (1995): *High-level Modeling with THORNs*. – Proc. International Congress on Cybernetics, Namur, Belgium.

Silberschatz A., Galvin P., Gagne G. (2001): *Operating System Concepts*. – John Willey & Sons, Inc.

Silva M., David R. (1979): *Synthése programmée des automatismes logiques décrits par réseaux de Petri: une méthode de mise en ceuvre sur microcalculateur*. – R.A.I.R.O. Automatique/Systems Analysis and Control, Vol. 13, No. 3, pp. 369-393.

Silva M., Velilla S. (1982): *Programmable logic controllers and Petri nets: a comparative study*. – Proc Conf. IFAC'82, Software for Computer Control, Madrid, Spain, pp. 83-88.

Silva M., Valette R. (1990): *Petri nets and flexible manufacturing*. – Lecture Notes in Computer Science 424, Springer Verlag, pp. 374-417.

Skowroński Z. (1998): *Dekompozycja systemów mikroprocesorowych współpracujących ze specjalizowanymi układami cyfrowymi na część sprzętową i programową*. – Mat. Kraj. Konf. Nauk. *Reprogramowalne Układy Cyfrowe: RUC'98*, Szczecin, ss. 75-82.

Skowroński Z. (2000): *Translacja specyfikacji funkcjonalnej układów cyfrowych na sieć Petriego dla potrzeb syntezy systemowej*. – Rozprawa doktorska, Szczecin.

Starke P.H. (1987): *Sieci Petri - Podstawy, Zastosowania, Teoria*. – Warszawa: PWN.

Staunstrup J., Wolf W. (Ed) (1997)*: Hardware/Software Co-Design Principles and Practice*. – Kluwer Academic Publishers.

Stoy E. (1995): *A Petri Net Based Unified Representation for Hardware/Software Co-Design*. – Licenciate Thesis No. LiU-Tek-Lic 1995:21, Linköping University, Linköping, Sweden.

Suraj Z., Szpyrka M. (1999): *Sieci Petriego i PN-Tools. Narzędzia do modelowania i analizy systemów współbieżnych*. – Rzeszów: Wydawnictwo Wyższej Szkoły Pedagogicznej.

Tal A.A, Yuditskiy S.A. (1982): *Hierarchy and parallelism in Petri networks. II. Complex automaton Petri networks with parallelism*. – Automation and Remote Control, No. 9, pp. 83-88.

Trung N.Q. (2001) - *Konfigurowlany model klasy mikrokontrolerów w języku VHDL*. – Rozprawa Doktorska, Warszawa.

Valette R. (1995): *Petri nets for control and monitoring: specification, verification, implementation*. – Proc. Conf. ADEDOPS'95, London, UK.

Varadharajan V., Baker K. (1987): *Directed graph based representation for software system design*. – Software Engineering Journal, pp. 21-28.

Vojnar T. (1997): *Hierarchical and Time Extensions of Pure Object Oriented Petri Nets*. – Proc. Conf. ASIS'97, Krnov, Ostrava.

Wagner F. (1994): *The Virtual State Machine: Executable Control Flow Specification*. – Rosa Fischer-Löw Verlag.

Węgrzyn M. (1998): *Hierarchiczna implementacja współbieżnych kontrolerów cyfrowych z wykorzystaniem FPGA*. –Rozprawa doktorska, Warszawa.

Węgrzyn M., Adamski M. (1999): *Hierarchical Approach for Design of Application Specific Logic Controller*. – Proc. Conf. ISIE'99, Bled, Słowenia, Vol. 3, pp. 1389-1394.

Wieting R., Sonnenschein M. (1995): *Extending High-level Petri Nets for Modeling Hybrid Systems*. – Proc. of the IMACS Symposium on Systems Analysis and Simulation, Berlin, Germany, Vol. 18/19, pp. 259-262.

Wolański P. (1998): *Modelowanie układów cyfrowych na poziomie RTL z wykorzystaniem sieci Petriego i podzbioru języka VHDL*. – Rozprawa doktorska, Warszawa, Politechnika Warszawska.

Abstract

This work presents a methodology of design and implementation of binary control systems which are used in digital microsystems. A modular and hierarchical structure is a characteristic attribute of such systems. It enables implementation of many concurrent processes of a sequential character. The realization of these processes is based on the global state analysis of a discrete controller described by an interpreted Petri net.

Modern chips of digital microsystems integrate in their structures a reprogrammable hardware matrix (e.g. FPGA, CPLD) with a microprocessor core (such as AVR or 8051). The implementation of control systems in these structures requires an approach to design process other than the classical. In many academic centers it has been shown that the Hardware/Software Co-Design methodology is more effective in the above mentioned cases. In this methodology, software and hardware parts are designed simultaneously. A commercial software for digital microsystems is heterogeneous as it is more flexible. Both parts are usually described in high level languages such as for example C (for the software) and VHDL (for the hardware), and it constitutes the main disadvantage of this approach, especially in cases of design of relatively simple control systems.

There is a need for developing a software package which has a homogeneous input interface (only one model of formal specification) and which renders it possible to carry out the hardware/software co-synthesis to VHDL and ANSI C. Research on design of a system supporting co-synthesis is conducted at the University of Zielona Góra. In this system, Petri net has been chosen as the model of formal description of controllers. An effective implementation of Petri nets (by means of digital microsystems) requires a considerable extension of the classical model of control Petri nets known from the literature. The author in his studies on software implementation of interpreted Petri nets has taken into account some elements taken from timed Petri nets and he has added hierarchy elements typical for hierarchical state diagrams such as Statecharts. There are many kinds of hierarchical nets described in the literature on the subject but most of them are only formulated as mathematical models. Their fitness for the purpose is limited as they lack some effective elements of description which are required in engineering practice, e.g. time dependencies, the history of macrostates, preemption. An alternative model of hierarchical Petri nets is a result of theoretical work. This model is called HPN (*Hierarchical Petri Net*) and it integrates most important elements of a formal description which are available in other kinds of nets, e.g. interpretation, time dependencies, enabling and inhibiting arcs, and the above mentioned history and preemption mechanism. It has allowed the elaboration of an

efficient model of specification of control systems with high flexibility of description methods and a clear graphic interface comparable to Statecharts.

Software implementation of Petri net can be realized in various ways. One of them involves defining an abstract program environment making efficient implementation of control systems possible. The proposed program environment is defined by names of net places, names of input/output signals and the program decision system called the Virtual Decision System – VDS. The system makes a decision about changing or not the state of the controller on the basis of information included in blocks storing the names of active input signals, the names of active local signals, information about the system state and the contents of specification block. There are also external input and output blocks which are assigned the task of collecting information about the values of external input signals and driving the external output signals.

In the work, a new program model of an HPN net is described in the software environment which together with the virtual decision system creates a realization kernel. So described model and environment can be relatively easily implemented in the software. It is enough to create suitable data structures and functions executing the information flow for making decisions.

A software synthesis module (called ORION) has been prepared which creates files in ANSI C standard on the basis of text description of nets in an *hpn* format. The *hpn* format has also been developed by the author. It provides a convenient way for input specification of design system to CAD software.

A series of comparative tests have been shown, revealing considerable benefits in the sense of occupied program memory size. In general, program memory occupation is the most important parameter because there is only 64 kB memory in modern digital microsytems (e.g. FPSLIC produced by ATMEL). Another parameter is the response time and it has been tested only for HPN. A comparison of the response times of this system and other systems (like POLIS worked out in Berkeley University or ESTEREL worked out in cooperation with the Institut National de Recherche en Informatique et en Automatique and Centre de Mathématiques Appliquées w Sophia-Antipolis) is a very difficult task because of too many differences in practical solutions of software synthesis of these systems. The benefits result from using the HPN program model and they refer to the results obtained from other tools (described above) for software synthesis. Additionally, the tests have been prepared on nets with various topology and size in order to demonstrate their influence on the quality of results.

The book was developed partially within the grants 7 T11C 010 20 and 4 T11C 006 24 of the State Committee for Scientific Research in Poland.