ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1996
Seria: AUTOMATYKA z.118 Nr kol. 1338

Lech GRODZKI
Politechnika Białostocka

## ZASTOSOWANIE TRANSLATORA JODA DO PROGRAMOWANIA ALGORYTMÓW STEROWANIA BINARNEGO

**Streszczenie.** Artykuł dotyczy zagadnień programowania sterowników binarnych w niestandardowych zastosowaniach (jednostkowe aplikacje, systemy wbudowane). Do ich programowania używa się zwykle klasycznych języków symbolicznych lub wysokopoziomowych. Praca przedstawia pokrótce proponowaną metodę usprawniającą stosowanie tych języków. Opiera się ona na traktowaniu sterownika binarnego jako automatu skończonego. Algorytm sterowania zapisuje się w specjalnym języku opisu. Praca zawiera opis: składni tego języka, możliwości jego translatora i planowanych rozszerzeń środowiska programowego języka JODA.

## APPLICATION OF JODA TRANSLATOR FOR BINARY CONTROL ALGORITHM PROGRAMMING

**Summary.** The article touches the problems of discrete controller programming in non-standard applications (singular or built-in systems). For programming of such problems classical assembly or high-level languages are used. The paper presents proposed methods, facilitating programming process. It is based on considering discrete controller as finite automaton. Control algorithm is described in special language. Paper contains description of: language syntax, abilities of its translator and planned extensions of programming environment for JODA language.

### 1. Wstęp

Metody programowania sterowników przemysłowych są obiektem zainteresowania wielu ludzi: producentów samych sterowników, projektantów systemów automatyki i wreszcie użytkowników tych systemów. O wadze problemu świadczy fakt opracowania międzynarodowej normy IEC 1131-3 [6] opisującej standardy programowania tych sterowników. Wśród metod opisu algorytmu sterowania norma ta wymienia: logikę drabinkową, język strukturalny i schematy funkcyjne. Standardy te są przyjmowane przez światowych producentów sterowników, dostosowujących swoje wyroby do zaleceń wymienionej normy.

Poza zasięgiem normy pozostaje jednak pewna grupa urządzeń przeznaczonych do realizacji sterowania binarnego i binarno-ciągłego. Są to konstrukcje jednostkowe, realizowane na specjalne zamówienie, lub mikrosterowniki będące systemami wbudowanymi do wnętrza innych urządzeń produkowanych seryjnie. Programowanie tych sterowników polega zwykle na wykorzystaniu, dostępnych dla stosowanego mikroprocesora, języków programowania (makroasemblery, kroskompilatory). Zapis przy użyciu tych metod całego algorytmu sterowania jest, oczywiście, możliwy, ale może nastręczać sporo trudności podczas jego częstych modyfikacji, które są nieodłącznym elementem procesu uruchamiania sterownika.

Niniejsza praca przedstawia metodę programowania mikroprocesorowych układów sterowania binarnego, częściowo uniezależniającą ten proces od docelowego mikroprocesora sterownika. Metoda ta zdała egzamin w kilku zastosowaniach, różniących się wielkością postawionego zadania, platformą sprzętową i programową. Narzędziem wspomagającym stosowanie tej metody jest wymieniony w tytule translator JODA.

## 2. Metoda opisu układu sterowania binarnego

Metoda JODA programowania sterowników binarnych wykorzystuje koncepcję traktowania tych urządzeń jako automatów. Istotnie, rozważając sterownik logiczny, można wskazać grupy: sygnałów wejściowych, składających się na słowo wejściowe $x_i$, oraz wyjściowych - tworzących słowo wyjściowe $y_j$. Działanie sterownika polega na reagowaniu na słowa wejściowe odpowiednimi zmianami słów wyjściowych. Uwzględniając ponadto fakt wpływu „historii" procesu na zachowanie się sterownika, dochodzi się do wniosku, że sterownik binarny może być traktowany jako automat skończony. Zakłada się przy tym, że słowo wyjściowe $y_j$ jest wyłącznie funkcją bieżącego stanu automatu. Biorąc także pod uwagę właściwy systemom komputerowym synchronizm operacji, otrzymuje się model sterownika jako układu synchronicznego typu Moore'a.

**Definicja 1**

Synchronicznym sekwencyjnym modelem sterownika binarnego w metodzie JODA nazywamy uporządkowaną piątkę:

$$MS = < S, X, Y, \delta, \lambda >$$

gdzie: $S = \{s_1, s_2, ..., s_N\}$ jest zbiorem stanów automatu, takim że $||S|| = N$ i $1 < N < \infty$;

$X = \{x_1, x_2, ..., x_K\}$ jest zbiorem słów wejściowych;

$Y = \{y_1, y_2, ..., y_M\}$ jest zbiorem słów wyjściowych;

δ jest funkcją przejść przyporządkowującą parze $(s_t, x_t)$ w chwili t nowy stan $s_{t+1}$ w chwili następnej t+1: $\delta(s_t, x_t) = s_{t+1}$;

λ jest funkcją wyjść określoną dla danego stanu $s_t$ wzorem: $y_t = \lambda(s_t)$ □

Słowa wejściowe $x_t$ składają się z liter, którymi są wejściowe zmienne logiczne. Zmienne te, to przede wszystkim sygnały dwustanowe przychodzące do sterownika z obiektu, ale także wewnętrzne flagi sterownika (np. od timerów) lub zmienne binarne pochodzące od innych układów sterowania, a służące koordynacji pracy różnych sterowników. Słowa wejściowe są argumentami wyrażeń logicznych warunkujących przejścia między stanami automatu. Wyrażenia te koduje się tak, by mogły być wartościowane opisaną w [1,3] metodą maska-wzór. Zmiana stanu automatu wiąże się ze zmianą jego słowa wyjściowego $y_j$. Na słowo to składają się, oprócz sygnałów wyjściowych sterujących obiektem, także flagi przesyłane do innych sterowników i zakodowane binarnie zmienne wielowartościowe. Te ostatnie mogą służyć do: inicjowania timerów, wskazywania na dodatkowe operacje charakterystyczne dla poszczególnych stanów itp. Słowa wyjściowe automatu wraz ze wskaźnikami do tablic przejść tworzą tzw. bloki opisu stanów.

Pierwsze praktyczne zastosowanie metody, opisane w [7], wykazało, że dzięki niej:

- potrzebna jest tylko jedna, wspólna dla wszystkich automatów, procedura wartościująca wyrażenia logiczne i wyznaczająca kod następnego stanu;
- nie trzeba pisać dla każdego stanu i automatu indywidualnych procedur obliczających wartości odpowiednich wyrażeń logicznych;
- uzyskuje się możliwość łatwej modyfikacji algorytmu sterowania, polegającej na wymianie tylko fragmentów kodu programu sterownika.

Spostrzeżenia te skłoniły do dalszych prac nad rozszerzeniem skromnych początkowo możliwości metody i do opracowania odpowiedniego oprogramowania wspomagającego. Jest nim wymieniony w tytule translator JODA, przetwarzający język deklaracji i wyrażeń logicznych na potrzebne struktury danych opisujących pracę sterownika.

## 3. Translator języka programowania automatów - JODA

Pierwsza wersja translatora powstała w 1990 roku i miała bardzo ograniczone możliwości: tylko generację tablic przejść w języku mikroprocesora Z80. Rok później możliwości programu uległy poszerzeniu o generację tablic wyjść i mikrokomputery rodziny

MCS51 [1]. Kolejne rozszerzenia wersji podstawowej, zachodzące w następnych latach, były częściowo wymuszane kolejnymi aplikacjami układów sterowania [2,4]. Także aktualna wersja translatora powinna być traktowana jako etap pośredni na drodze do kompletnego systemu programowania nietypowych implementacji sterowania binarnego.

Język JODA służy do opisu działania automatu skończonego, za jaki można uważać sterownik binarny podłączony do obiektu. Składnia tego języka umożliwia:

- wybór platformy programowej;
- zdefiniowanie struktury słowa wejściowego $x_i$;
- zdefiniowania struktury słowa wyjściowego $y_j$;
- zadeklarowanie zbioru $S = \{s_1, s_2, ..., s_N\}$ stanów automatu oraz wartości słów wyjściowych $y_j$ dla każdego $s_j \in S$;
- zdefiniowanie warunków przejść pomiędzy dowolnymi dwoma stanami $s_i, s_j \in S$ w formie wyrażeń logicznych o składni typowej dla języków wysokiego poziomu;
- stosowanie mnemotechnicznych nazw stanów i zmiennych logicznych;
- stosowanie rozbudowanych komentarzy objaśniających poszczególne obiekty w programie;
- zawarcie w jednym pliku źródłowym opisu więcej niż jednego automatu.

Składnię języka JODA w zmodyfikowanej notacji Backusa-Naura (MBNF), której opis można znaleźć np. w [5], przedstawiono poniżej.

Składnia języka JODA

*program* = "**jezyk**" *język_docelowy* ";" *opis_automatu* { ";" *opis_automatu* } "." .
*język_docelowy* = ( "**az80**" | "**a8051**"| "**c16**"| "**c32**" | "**pascal**" | "**c**" ).
*opis_automatu* = *tekst_źródłowy* | *tekst_zakodowany* .
*tekst_źródłowy* = *nagłówek deklaracja_wejść* [ *deklaracja_wyjść* ] [ *deklaracja_flag* ]
*deklaracja_stanów opis_przejść* "**koniec**" .
*nagłówek* = "**automat**" *tekst* ";" .
*tekst* = *ciąg_znaków* .
*ciąg_znaków* = {"**znak ASCII**"} .
*deklaracja_wejść* = "**bity**" *bajt_wejść* ";" { *bajt_wejść* ";" } .
*bajt_wejść* = *bajt* .
*bajt* = *bit* "," *bit* "," *bit* "," *bit* "," *bit* "," *bit* "," *bit* "," *bit* . *bit* .
*bit* = ( "-" | *identyfikator* ) .
*deklaracja_wyjść* = "**wyjscia**" { *bajt_wyjść* ";" }
[ *deklaracja_stałej* { "," *deklaracja_stałej* } ";" ] .
*bajt_wyjść* = *bajt* .
*deklaracja_stałej* = *deklaracja_rozmiaru_stałej* [ *deklaracja_nazwy_stałej* ].
*deklaracja_rozmiaru_stałej* = *rozmiar_bajtu* | *rozmiar_słowa* | *rozmiar_czterobajtowy* .
*rozmiar_bajtu* = "#" .
*rozmiar_słowa* = "##" .
*rozmiar_czterobajtowy* = "####" .

*deklaracja_nazwy_stałej* = *identyfikator* .
*deklaracja_flag* = "**flagi**" *bajt_flag* ";" { *bajt_flag* ";" } .
*bajt_flag* = *bajt* .
*identyfikator* = *litera* { *litera* | *cyfra* } .
*litera* = ( "**a**" | .. | "**z**" | "**A**" | .. | "**Z**" ) .
*cyfra* = ( "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**" ) .
*deklaracja_stanów* = "**stany**" *opis_stanu* ";" { *opis_stanu* ";" } .
*opis_stanu* = *identyfikator_stanu* ":" [ *bit* { "," *bit* } { "," *stała* }] | [ *stała* { "," *stała* }] ";" .
*identyfikator_stanu* = *identyfikator* .
*stała* = *liczba* | *identyfikator* .
*liczba* = *cyfra* { *cyfra* } .
*opis_przejść* = "**funkcje**" *przejście* { *przejście* } .
*przejście* = *identyfikator_stanu* "-" *identyfikator_stanu* ":" *wyrażenie_logiczne* ";" .
*wyrażenie_logiczne* = *składnik* { "+" *składnik* } .
*składnik* = *czynnik* { "*" *czynnik* } .
*czynnik* = *identyfikator* | "/" *czynnik* | "(" *wyrażenie_logiczne* ")" .
*tekst_zakodowany* = *nagłówek_kodu* [ *deklaracja_wejść* ] *program_zakodowany* "**koniec**" .
*nagłówek_kodu* = "**dekoduj**" *tekst* ";" .
*program_zakodowany* = "**treść zakodowanego uprzednio programu**" .
*dyrektywa* = *nazwa_dyrektywy parametr_dyrektywy* .
*nazwa_dyrektywy* = ( "**list**" | "**opt**" ) .
*parametr_dyrektywy* = ( "**on**" | "**off**" ) .
*komentarz* = "{" *tekst* "}" .

Uwagi dodatkowe:

- bity w bajtach wejściowych i wyjściowych podaje się w kolejności od najstarszego do najmłodszego, a bajty opisuje się w kolejności od najmłodszego do najstarszego;
- znak "-" przy deklarowaniu tych bitów oznacza nie używaną przez dany automat pozycję w bajcie;
- deklaracja stałej w słowie wyjściowym określa przede wszystkim jej rozmiar w bajtach ('#' - 1 bajt, '##' - 2 bajty, '####' - 4 bajty), opcjonalnie po zdefiniowaniu rozmiaru można podać nazwę tej stałej - będzie ona miała znaczenie przy deklaracjach typów rekordowych na potrzeby zapisu w języku wysokopoziomowym (Pascal, C);
- znaki "/", "*" i "+" to operatory odpowiednio: negacji, iloczynu i sumy logicznej;
- translator dopuszcza także następujące uproszczenia w zapisie wyrażeń logicznych:

  ")" *czynnik* ≡ ")*" *czynnik*

  *czynnik* "(" ≡ *czynnik* "*("

  *czynnik* „/" ≡ *czynnik* "*/"

  "/(" *czynnik* ≡ *czynnik*

przedstawiona notacją MBNF składnia wyrażeń, w celu uzyskania prostego jej zapisu, uwzględnia tylko nieuproszczone wersje wyrażeń;

- stałe charakterystyczne stanów w deklaracjach stanów mogą być określane poprzez nazwy, znaczenie tych nazw (reprezentowana przez nie wartość) musi być wtedy zdefiniowane w treści programu docelowego;
- komentarz może rozpoczynać się w dowolnym miejscu opisu z wyjątkiem „wnętrza" identyfikatorów i słów kluczowych;
- dwie dostępne dyrektywy służą sterowaniu opcjami: przedruku tłumaczonego opisu automatu do pliku zawierającego raport z działania translatora (**list**) oraz optymalizacji wyrażeń logicznych (**opt**).

Możliwości leksykalne translatora są następujące:

- długość identyfikatorów stanów i bitów (zmiennych logicznych) oraz nazw stałych jest ograniczona do 10 znaków;
- maksymalna liczba stanów w jednym automacie: 256;
- maksymalny rozmiar słowa wejściowego: 32 bajty (256 zmiennych wejściowych);
- maksymalny rozmiar słowa wyjściowego: 32 bajty zmiennych wyjściowych (256 bitów), 32 bajty flag wyjściowych (256 bitów), 10 jedno- , dwu- lub czterobajtowych stałych charakterystycznych stanu.

Translator ma rozbudowaną diagnostykę danych wejściowych, ze szczególnym uwzględnieniem kontroli wyrażeń logicznych, wskazywaniem miejsca i rodzaju wykrytych błędów. Przetwarzanie wyrażeń logicznych polega między innymi na: wyeliminowaniu nawiasów, przekształceniu do postaci APN, minimalizacji liczby implikantów. Wynikiem jego pracy może być raport z translacji, zawierający przedruk tekstu źródłowego z zaznaczeniem ewentualnych błędów. Przy braku błędów w danych wejściowych generowany jest plik tekstowy w formacie zgodnym z zadeklarowanym językiem docelowym translacji. Zawiera on zapisane w składni języka docelowego: listę stanów automatu, bloki opisu stanów, zawierające dla każdego ze stanów wskaźniki do tablic przejść i słowa wyjściowe, tablicę flag i tablicę zakodowanych funkcji przejść. Plik ten może być następnie dołączony do właściwego tekstu programu sterownika zapisanego w asemblerze lub języku wysokiego poziomu.

O dużej elastyczności metody świadczą zrealizowane za jej pomocą aplikacje, różniące się między sobą:

- bazą sprzętową: od systemów modułowych z mikroprocesorami Z80, poprzez mikrokomputery jednoukładowe 8051 do IBM 486DX;

- stosowanym językiem programowania: od asemblera symbolicznego po język wysokiego poziomu;
- skalą rozwiązywanych problemów: od prostego układu reagującego na klawisze i upływ czasu, poprzez sterowania dwustanowe urządzeniami wykonawczymi do realizacji zadań automatyki kompleksowej.

Przykładowymi zastosowaniami translatora JODA są: modułowy system automatyki kompleksowej doświadczalnej linii uzysku białka [7], mikroprocesorowy sterownik formatyzerki [2], sumator hematologiczny i oparty na IBM PC system automatyki oczyszczalni ścieków [4]. Przykład użycia translatora w ostatniej z wymienionych aplikacji, ze względu na jego obszerność, nie może być zamieszczony w treści artykułu.

Ponieważ w praktyce programowania zdarzają się, niestety, przypadki utraty archiwalnych wersji programów, aby umożliwić odtworzenie opisów pracy automatu w języku JODA, translator został wyposażony w mechanizm translacji odwrotnej. Polega on na tym, że jako plik źródłowy podawany jest odpowiednio przygotowany fragment wygenerowanego wcześniej pliku docelowego. Dane do odwrotnej translacji wymagają przede wszystkim określenia języka programowania, z jakiego ma być ona przeprowadzona. Fragment przetwarzanego programu poprzedza się rozkazem **'dekoduj'**, a następnie, zależnie od potrzeb i możliwości, używa się:

- deklaracji słowa wejściowego **'bity'**, w której można zdefiniować nazwy zmiennych logicznych będących argumentami funkcji przejść lub przynajmniej określić długość słowa wejściowego;
- słowa kluczowego **'funkcje'** do wskazania fragmentu programu zawierającego zakodowane funkcje przejść.

Czytając tak przygotowany plik wejściowy, translator tworzy listę stanów i odtwarza wyrażenia logiczne w zapisie symbolicznym. Wynik translacji odwrotnej jest zapisywany do wskazanego pliku wyjściowego w formacie zgodnym ze składnią języka JODA. Dzięki temu możliwa jest szybka modyfikacja tego pliku i przygotowanie go do dalszego wykorzystywania już jako zwykłego zbioru tekstowego zawierającego opis automatu w języku JODA.

## 4. Perspektywy rozszerzenia funkcji translatora JODA

Dostępne środowisko programowe dla metody JODA, obejmujące aktualnie sam translator, będzie ulegać dalszej rozbudowie. Jednym z dodatkowych środków

uruchomieniowych będzie debuging automatu w trybie on-line. Specjalizowane oprogramowanie będzie za pośrednictwem łącza szeregowego odczytywać informację o stanie automatu. Otrzymana informacja ma być przedstawiana w syntetycznej postaci tekstowej na ekranie monitora. Po wprowadzeniu przewidywanych graficznych metod opisu automatu, jego praca podczas debugingu on-line mogłaby być także ilustrowana na tle grafu przejść.

Oprogramowanie uruchomieniowe pozwoli jednocześnie na modyfikację wybranych fragmentów lub też całych struktur danych opisujących automatu. Operacja ta, polegająca na transmisji tablic binarnych nie będących kodem maszynowym mikroprocesora, nie wymaga stosowania dodatkowego oprogramowania narzędziowego. W tym miejscu ujawnia się jedna z zalet metody - niezależność opisu automatu od procesora sterownika.

Bardzo wygodnym środkiem przyśpieszającym powstawanie kolejnych modyfikacji algorytmu sterowania byłaby możliwość wymiany opisujących automat tablic: przejść i wyjść w trakcie pracy sterownika. Wykorzystywany byłby do tego wyróżniony stan spoczynkowy automatu. Uwzględniając fakt, iż przepisanie nowych struktur danych opisujących automat do pamięci sterownika jest operacją krótkotrwałą, możliwe jest zmodyfikowanie pracy automatu w chwilach, gdy przebywa on w takim stanie.

Dodatkowym programem narzędziowym wspomagającym uruchamianie automatu ma być również prosty symulator jego działania. Wykorzystując charakterystyczną dla metody JODA niezależność opisu od sprzętu, umożliwi sprawdzenie zachowania się sterownika przy różnych słowach wejściowych. Wpłynie to na przyśpieszenie realizacji zadania i zmniejszenie kosztów prac uruchomieniowych poprzez wyeliminowanie części błędów w algorytmie sterowania jeszcze przed użyciem rzeczywistego sterownika.

Wymienione tu planowane kierunki rozbudowy narzędzi wspomagających, stosowanie metody JODA do programowania sterowników binarnych stworzą razem zintegrowane środowisko uruchomieniowe, obsługujące wszystkie fazy realizacji zadania: projekt wstępny, uruchamianie na sterowniku i późniejsze modyfikacje.

## LITERATURA

1. Grodzki L.: Metoda oprogramowania sterowników binarno-ciągłych procesów przemysłowych na poziomie asemblera. VIII Krajowa Konferencja Automatyzacji Dyskretnych Procesów Przemysłowych 1992 - Zeszyty Naukowe Politechniki Śląskiej, Automatyka z.109, Gliwice 1992, s.67-76.

2. Grodzki L., Kierus K.: Mikroprocesorowy sterownik formatyzerki - sprawozdanie z realizacji pracy umownej dla Zakładów Produkcji Sklejek w Białymstoku (49/90-RNN/U/187/90), Białystok 1991.
3. Grodzki L.: Porównanie różnych metod wartościowania funkcji logicznych w mikroprocesorowych sterownikach PLC. Zeszyty Naukowe Politechniki Białostockiej, Elektryka z.13, Białystok 1994.
4. Grodzki L.: System automatyki komunalnej oczyszczalni ścieków. V Krajowa Konferencja Automatyzacja i Eksploatacja Systemów Sterowania. Akademia Marynarki Wojennej, Gdynia 1995.
5. Iglewski M., Madey J., Matwin S.: Pascal. WNT, Warszawa 1984.
6. Internationale Standard IEC 1131-3. Programmable controllers - part 3: Programming languages. IEC,Genewa 1993.
7. Leszczyński J., Grodzki L.: Oprogramowanie systemu sterowania doświadczalną linią produkcji kazeiny. Materiały XI Krajowej Konferencji Automatyki, Białystok 1991.

Recenzent: Dr inż. Tadeusz Legierski

Wpłynęło do Redakcji do 30.06.1996 r.

**Abstract**

The article presents the method of binary control algorithm programming and abilities of utility program - JODA language translator. Those language and its translator support programming in classical assembly or high-level languages for non-standard applications (singular or built-in systems).

Discrete controller is treated as Moore's automaton, which reacts on changes of the input word. This input word consists of several components: object binary variables, internal controller variables (for example timer flags) and binary flags from other control units, used to operate together with them in the large control systems. The input word bits are used in logical expressions, which are conditioning the transitions between automaton states. Automaton output word related to each state consists of some binary variables acting on object and output flags transmitted to other controllers. Output word can also contain some multiple-valued variables used to executing special tasks.

JODA language is used for description of automaton - its structure and operation. The article presents syntax of this language and abilities of its translator used for generating data structures, coded in selected controller programming language. Possible target languages are: assembly languages, Pascal and C. Logical expression translation consist of: syntax analysis, conversion to disjunctive form, reduction of terms, minimization of number of implicants and coding. Translator can also realize the reverse translation to restore automaton description in JODA language.

The method and translator described were used successfully in several applications, with different controller hardware, scale of problems and target programming languages. The above incline the author to plan the extensions of the programming environment, which are marked in the last part of the paper.