mgr inż. Jan Sadolewski
Politechnika Rzeszowska, Katedra Informatyki i Automatyki

ASERCYJNE ROZSZERZENIE JĘZYKA ST NORMY IEC 61131-3 DO DYNAMICZNEJ WERYFIKACJI SYSTEMÓW STEROWANIA[1]

W pracy przedstawiono propozycję asercyjnego rozszerzenia języka ST (Structured Text) normy IEC 61131-3, nawiązującego do reguł projektowania kontraktowego i języka JML (Java Modeling Language). Zapisane asercje można przekształcić do kodu podczas kompilacji w celu uzyskania możliwości dynamicznej weryfikacji programów sterowania oraz detekcji błędów czujników. Przykłady dotyczą sterowania układem grzałek w zależności od temperatury oraz urządzenia do sortowania drewna.

ASSERTIONAL EXTENSION IN ST LANGUAGE OF IEC 61131-3 STANDARD FOR CONTROL SYSTEMS DYNAMIC VERIFICATION

The paper presents a proposition of assertional extension in Structured Text language from IEC 61131-3 standard, according to design by contract rules and JML (Java Modeling Language). Stored assertions could be converted to the code at compile time to obtain possibility of dynamic verification and for sensors failure detection. Heater control system and wood sorter machine are examples.

1. WPROWADZENIE

Współczesne standardy tworzenia oprogramowania, wykorzystują projektowanie kontraktowe [7] (ang. *design by contract*). Zaletą takiego podejścia jest precyzyjne ustalenie zakresu funkcjonalności pomiędzy autorem specyfikacji warunków końcowych, a autorem implementacji i specyfikacji warunków początkowych. Ustalenia te nazywa się *kontraktem*, który powinien zostać zapisany w sposób formalny w pobliżu kodu, który opisuje. Języki programowania takie jak ANSI C, C++ czy Java mają własne narzędzia do formalnego zapisu specyfikacji (kontraktu) w postaci języków ACSL, Larch, i JML [1, 5, 6], ponadto języki Eiffel i Why [8, 3] mają wbudowaną w składnię możliwość zapisania specyfikacji bezpośrednio w ich kodzie. Tak zapisaną specyfikację można poddać weryfikacji statycznej z jej implementacją w celu wychwycenia i usunięcia błędów w implementacji oraz do przedstawienia formalnego dowodu zgodności. Niniejsza praca przedstawia propozycję asercyjnego rozszerzenia języka ST normy IEC 61131-3 do programowania systemów sterowania. Rozszerzenie to oparto na standardzie JML, wprowadzając jednocześnie możliwość weryfikacji dynamicznej programu w czasie jego wykonywania (*run-time checking*), a także podjęcia *akcji ratunkowej* w sytuacjach nie spełniających założonego kryterium.

Praca podzielona jest następująco. W punkcie 2 omówiono cechy i zastosowanie asercji w językach programowania w tym standardu JML. Punkt 3 zawiera semantykę asercyjnego rozszerzenia języka ST oraz jego uzupełnienie do obsługi sytuacji krytycznych. Punkt 4 opisuje znaczenie klauzul podczas weryfikacji dynamicznej. W punkcie 5 przedstawiono dwa proste przykłady systemów sterowania wykorzystujące asercyjne rozszerzenie do weryfikacji dynamicznej oraz wykrywania defektów czujników.

[1] Praca naukowa finansowana ze środków na naukę w latach 2010 – 2011 jako projekt badawczy nr N N516 41538.

2. ASERCJE W JĘZYKACH PROGRAMOWANIA

Asercją nazywamy fragment kodu składającego się z co najmniej wyrażenia logicznego, którego każdorazowe obliczenie w miejscu i kolejności jego wstąpienia ma zwrócić wartość pozytywną. Standardowe asercje w językach programowania takich jak ANSI C, Delphi (Pascal), czy C# przedstawiono w tab. 1. Typowo mają zastosowanie tylko do celów testowych, a ich kod „znika" w czasie kompilacji do postaci finalnej opracowywanego modułu. Po zmianie odpowiednich ustawień w kompilatorze postać finalna może również zawierać asercje. Problem pojawia się dopiero w sytuacjach, kiedy sprawdzenie warunku asercji zwraca wartość negatywną. W zależności od implementacji niespełnienie asercji kończy się oknem komunikatu, utworzeniem wyjątku, bądź przerwaniem programu. Drugim utrudnieniem przy korzystaniu ze standardowych asercji są komunikaty pomocnicze, w najlepszym przypadku definiowane przez użytkownika, w których nie ma informacji o przyczynach ich niespełnienia. Projektowanie kontraktowe wykorzystuje dwie dodatkowe asercje `requires` do określenia warunków wstępnych i `ensures` do warunków końcowych projektowanej, a następnie formalnie weryfikowanej jednostki. Asercje te zapisywane są w postaci adnotacji do kodu poprzez odpowiednie klauzule języka lub komentarze specjalne. Języki Eiffel i Why wykorzystują wbudowane klauzule, natomiast JML i ACSL używają komentarzy rozpoczynających się od znaku '@'. Ta cecha przyczyniła się do powstania standardu dla języków określanych mianem BISL (*Behavioral Interface Specification, Language*), których przykład stanowi standard JML. Wykorzystanie komentarzy jako miejsca do adnotacji pozwala na zapewnienie przenaszalności kodu źródłowego pomiędzy kompilatorami różnych producentów.

Tab. 1. Asercje w popularnych językach programowania

ANSI C, POSIX	*Delphi, Pascal*	*C# (Platforma .NET)*

`#include <assert.h>`
`void additem(struct ITEM *itemptr)`
`{`
`assert(itemptr != NULL);`
`... }`
	`procedure ModifyStorage (AStorage: TStorage; const s: string); begin`
	`Assert(AStorage <> nil, '');`
	`AStorage.Data := s; end;`
		`{`
		`int index;`
		`...`
		`System.Diagnostics.Debug .Assert(index > -1);`
		`... }`

Standard JML zawiera notację do zapisu formalnej specyfikacji zachowania klas i metod języka Java. Zubożone obiekty języka Java przypominają jednostki oprogramowania (POU) zdefiniowane w normie 61131-3, dlatego wykorzystanie JML jako bazy dla asercyjnego rozszerzenia języka ST wydaje się uzasadnione. Naturalnym wydaje się fakt, że tylko pewien podzbiór standardu JML będzie miał zastosowanie w systemach sterowania, dlatego aspekty typowe dla Java, nie mające odzwierciedlenia w ST, zostały pominięte.

3. SEMANTYKA ASERCYJNEGO ROZSZERZENIA JĘZYKA ST

Adaptację asercyjnego rozszerzenia dla języka ST przedstawiono w tab. 2. Klauzule języka zostały pogrupowane ze względu na ich typ. Każda z klauzul ma zakres swojej widoczności. Zakres instrukcji może pojawić się w dowolnym miejscu programu, gdzie mogą znaleźć się instrukcje lub wyrażenia. Zakres lokalny odnosi się do całej instancji obiektu POU, natomiast zakres globalny dotyczy całego projektu (konfiguracji wg normy). Zakres mieszany sygnalizuje możliwość wystąpienia klauzuli w zależności od kontekstu.

Z powodu konfliktów słów kluczowych asercyjnego rozszerzenia z identyfikatorami języka ST, a także w celu ułatwienia analizy kodu przez programistów, wprowadzono następujące nieznaczne modyfikacje:

- przeniesiono adnotacje opisujące zachowanie metody do wnętrza obiektu POU, bezpośrednio po nazwie obiektu występującej w jego definicji[2],
- wprowadzono jednowierszowe komentarze w specyfikacji od znaków '//' do najbliższego znaku końca linii lub znaków '*)' zamykających adnotację,
- wprowadzono znak dwukropka po słowie kluczowym adnotacji rozpoczynającym klauzule i średnik po wyrażeniu występującym jako argument klauzuli.

Klauzule umieszcza się wewnątrz opisywanej jednostki, z uwzględnieniem jej zakresu widoczności. Przeznaczeniem lokalnej klauzuli `ensures` jest wyrażenie warunku końcowego jednostki oprogramowania, jaki musi zostać spełniony po jej zakończeniu wykonywania. Zakres widoczności tej klauzuli odnosi się do jednostki, w której została zadeklarowana. Podobnie klauzula `requires` wyraża warunki początkowe, jakie muszą zostać spełnione, aby jednostka wykonała się poprawnie przy spełnionych warunkach początkowych, a otrzymany rezultat działań spełniał warunki końcowe tej jednostki. Klauzula `assert` służy do definicji warunku, który powinien zostać spełniony w dowolnym miejscu gdzie mogą pojawić się instrukcje. Klauzule `requires` i `ensures` są specjalnymi przypadkami klauzuli `assert`. W adnotacji funkcji klauzula `ensures` może wykorzystać specjalną wartość `\result` dla oznaczenia wartości zwróconej przez funkcję (zgodność z JML), alternatywnie nazwę weryfikowanej funkcji (zgodność z językiem ST).

Tab. 2. Adaptacja klauzul JML dla języka ST

Typ	***Standard JML***	***Adaptacja w ST***	***Zakres***
Asercje	`assert`	`assert`	instrukcji
	`ensures`	`ensures:`	lokalny
	`requires`	`requires:`	lokalny
Modyfikatory lokalizacji	`\at`	`\at` lub `at`	instrukcji
	`\old`	`\old`	instrukcji
Kwantyfikatory	`\exists`	`\exists`	mieszany
	`\forall`	`\forall`	mieszany
Niezmiennik	`invariant`	`invariant:`	instrukcji
Deklaracje	`label`	`label:`	instrukcji
	`logic`	`logic:`	globalny
	`ghost`	`ghost:`	lokalny
	`predicate`	`predicate:`	globalny
	`axiom`	`axiom:`	globalny
Wartość zwracana funkcji	`\result`	`\result` lub *nazwa_funkcji*	lokalny
Operacje	`set`	`set:`	instrukcji
	`assigns`	`assigns:`	lokalny
Poprawność iteracji	`variant`	`variant:`	instrukcji

Weryfikacja jednostki oparta jest na *stanach pamięci* zawierających wartości zmiennych w określonych momentach wykonywania jednostki [2]. Klauzula `\old(`*zm*`)` reprezentuje wartość zmiennej o nazwie *zm* w chwili rozpoczęcia jednostki. Dla programów i bloków funkcjonalnych jest to również wartość zmiennej otrzymana na zakończenie poprzedniego cyklu.

[2] Dla funkcji adnotacje umieszcza się po deklaracji typu wartości zwracanej.

Stany wykonywania jednostki można nazywać za pomocą klauzuli `label`. Dostęp do wartości zmiennej z zadeklarowanego stanu uzyskuje się za pomocą klauzuli `\at(`*`zm`*`,`*`etyk`*`)`, gdzie *`etyk`* jest wcześniej zadeklarowaną etykietą stanu. Klauzula `assigns` wskazuje które zmienne globalne mogą ulec zmianie podczas wykonywania kodu jednostki.

W zapisie adnotacji mogą pomóc funkcje abstrakcyjne deklarowane za pomocą klauzuli `logic`, często wykonywane operacje na zmiennych używanych w asercjach mogą zostać zgrupowane w formie funkcji – predykatu poprzez wykorzystanie klauzuli `predicate`. Aksjomaty pomocnicze deklaruje się poprzez klauzulę `axiom`. Dodatkowe zmienne lokalne deklarowane w celu uproszczenia obliczeń w wyrażeniach asercji, nie występujące w kodzie jednostki, deklaruje się za pomocą klauzuli `ghost`, a pomocnicze operacje na tych zmiennych można definiować w klauzuli `set`. Adnotacje do pętli w postaci niezmiennika umieszcza się w klauzuli `invariant`, a wyrażenie określające skończoność pętli w klauzuli `variant`. Adnotacje te mogą wykorzystywać dwa kwantyfikatory – ogólny umieszczany w klauzuli `\forall` i szczegółowy umieszczany w `\exists`.

Tab. 3. Rozszerzenie JML dla języka ST

Typ	***Zapis w ST***	***Zakres***
Stany bezpieczne	`safe_behaviour`	lokalny
Obsługa wyjątków	`general_failure:`	lokalny
	`requires_failure:`	lokalny
	`ensures_failure:`	lokalny
Punkty wznowienia	`resume_after`	mieszany
	`resume_after_unit`	mieszany
	`resume_from_unit`	mieszany
	`resume_from_program`	mieszany
Punkt zatrzymania	`terminate_execution`	mieszany

Jak już wspomniano, standard JML został przewidziany do formalnego zapisu specyfikacji w języku Java, i nie zdefiniowano w nim elementów typowych dla systemów sterowania. Propozycję rozszerzenia o nowe elementy dla języka ST przedstawiono w tab. 3. Pierwszym rozszerzeniem jest definicja stanu bezpiecznego. Stan bezpieczny to taki, w którym nie występuje zagrożenie, spowodowane awarią komponentów systemu sterującego. Definicja takiego stanu dla każdego POU wykorzystuje klauzulę `safe_behaviour`. Ponieważ stany bezpieczne mogą być różne w zależności od fazy procesu, dlatego proponuje się aby każdy stan bezpieczny miał nadaną nazwę. Definicja stanu bezpiecznego zawiera kod, który przełączy wartości zmiennych na odpowiednie wartości, a ostatnią instrukcją tej definicji jest jeden z punktów wznowienia albo zatrzymania.

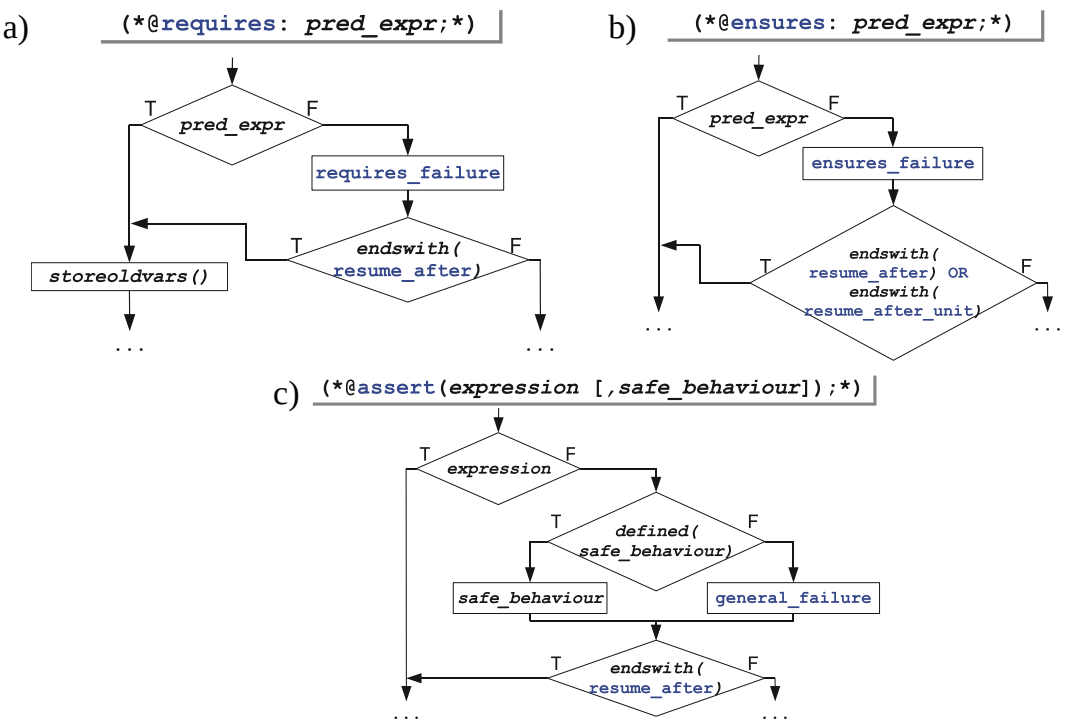
Punkty wznowienia określają miejsca od których przetwarzanie zadania sterownika będzie kontynuowane. Punkt `resume_after` określa wznowienie pracy w miejscu bezpośrednio po sprawdzeniu wyrażenia asercji. Oznaczenie `resume_after_unit` wznowi pracę w miejscu następującym po zakończeniu przetwarzania tego POU, natomiast `resume_from_unit` wznowi pracę sterownika od początku bieżącego POU. Punkt `resume_from_program` ponowi pracę sterownika od początku aktualnie wykonywanego programu. W przeciwieństwie do punktów wznowienia punkt zatrzymania (`terminate_execution`) przerywa pracę sterownika po ustaleniu wartości wyjść. W tej sytuacji wznowienie pracy nastąpi dopiero po ręcznym zresetowaniu sterownika.

Przechwytywanie niespełnionych standardowych asercji `requires` i `ensures` powierzono klauzulom `requires_failure` i `ensures_failure`, w których określa się nazwy stanów bezpiecznych. Podobnie `general_failure` zawiera stan bezpieczny dla klauzul `assert`, którym nie określono indywidualnego stanu bezpiecznego (`assert` z jednym argumentem). Rozszerzenie to stanowi dozór nad wykonywanym programem, którego realizację opisano w następnym punkcie.

4. REALIZACJA WERYFIKACJI DYNAMICZNEJ

Zadaniem weryfikacji dynamicznej jest sprawowanie dozoru nad wykonywaniem programu. Polega to na sprawdzaniu spełnienia warunków początkowych, asercji i warunków końcowych jednostek. W praktyce realizowane jest to przez odpowiednie przekształcenia adnotacji do postaci instrukcji operujących na aktualnie wykonywanym kodzie jednostki.

Klauzula `(*@ghost: zm : typ; *)` przekształcana jest podczas kompilacji do postaci `VAR zm : typ; END_VAR`. Klauzula `(*@label: etyk;*)` wywołuje wewnętrzną funkcję `storeatvars(etyk)`, której zadaniem jest wykonanie kopii wartości zmiennych z modyfikatorem `\at` występujących w adnotacjach. Wartości tych zmiennych zostaną oznaczone nazwą etykiety *etyk* przekazanej w klauzuli `label`. Klauzula `(*@set: ghost_var := expr;*)` zostanie przetłumaczona do postaci *ghost_var* `:=` *expr*.



Rys. 1. Sieci działań weryfikacji dynamicznej dla klauzul: a) requires, b) ensures, c) assert.

Asercje tłumaczone są do postaci sieci działań. Sieć dla klauzuli `requires` została przedstawiona na rys. 1a. Zapis klauzuli tłumaczony jest do postaci wyrażenia warunkowego, którego niespełnienie wywołuje obsługę stanu bezpiecznego skojarzonego w klauzuli `requires_failure`. W przypadku, gdy stan bezpieczny zakończony jest punktem powrotu `resume_after`, to przetwarzanie programu jest wznawiane w taki sposób jakby wyrażenie warunkowe zostało spełnione. Ostatnią instrukcją poprawnego zakończenia klauzuli `requires` jest wywołanie wewnętrznej metody `storeoldvars()`, której zadaniem jest zapamiętanie obecnej wartości zmiennych, które występują w adnotacjach z modyfikatorem `\old`.

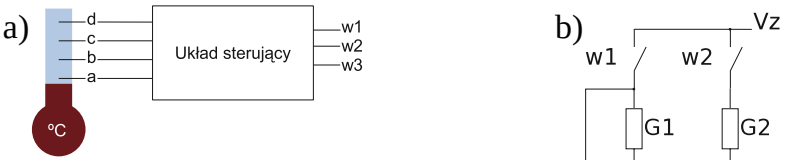
Sieć działań dla klauzuli `ensures` przedstawiono na rys. 1b. W przypadku niespełnienia wyrażenia warunkowego tej klauzuli wywoływana jest obsługa stanu bezpiecznego skojarzonego w klauzuli `ensures_failure`. Gdy definicja stanu bezpiecznego kończy się punktem powrotu `resume_after` lub `resume_after_unit`, to przetwarzanie wznawiane jest od punktu zakończenia aktualnie sprawdzanej jednostki.

Sieć działań dla klauzuli `assert` przedstawiono na rys. 1c. Klauzula ta ma dwa warianty. Wariant pierwszy (jednoargumentowy), w przypadku niespełnienia wyrażenia, obsługuje stan bezpieczny skojarzony z klauzulą `general_failure`. Wariant drugi (dwuargumentowy) obsługuje stan bezpieczny, którego identyfikator został przekazany jako drugi argument. W obu wariantach, gdy obsługa stanu bezpiecznego kończy się punktem `resume_after`, to wznawiana jest praca sterownika w miejscu następującym po asercji.

Zastosowanie weryfikacji dynamicznej pozwala zwiększyć bezpieczeństwo wykonywanego oprogramowania, poprzez ochronę przed niezamierzonym lub celowym wprowadzeniem błędnych parametrów, czy poprzez wykrycie uszkodzeń czujników. Omawiane w następnym punkcie przykłady wykorzystują asercyjne rozszerzenia i weryfikację dynamicznej do detekcji nieprawidłowych wartości wejść układów.

5. PRZYKŁADY

Przykład 1 pochodzi z [9]. Termometr kontaktowy (rys. 2a) generuje sygnały `a`, `b`, `c`, `d`, gdy temperatura przekroczy odpowiednie wartości. Układ sterujący powinien tak generować sygnały dla przełączników `w1`, `w2`, `w3`, (rys. 2b), aby spełnić wymagane warunki włączeń w zależności od temperatury *t*, wymienione na rys. 2a. Styki na rys. 2b narysowano w pozycji 0.



$t < ta$ – G1 i G2 włączone
$ta \le t < tb$ – tylko G1 włączony,
$tb \le t < tc$ – tylko G2 włączony,
$tc \le t < td$ – G1 i G2 połączone szeregowo,
$t \ge td$ – G1 i G2 wyłączone.

Rys. 2. Grzałki sterowane termometrem, a) sygnały układu, b) schemat połączeń grzałek.

Przekroczenie każdego progu temperatury jest sygnalizowane stanem 1 odpowiednich czujników (a – d). Po utworzeniu pierwotnej tablicy stanów wejść i odpowiadającym im stanów wyjść, którą przedstawiono w tab. 4, przystąpiono do otrzymania równań specyfikujących zachowanie programu poprzez minimalizację za pomocą tablic Karnaugha. Tablice te wraz z otrzymanymi wzorami zastosowanymi w specyfikacji przedstawiono na rys. 3.

Tab 4. Tablica stanów wejść i wyjść przykładu 1

Wejścia				Wyjścia		
a	b	c	d	w1	w2	w3
0	0	0	0	1	1	0
1	0	0	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	1	1
1	1	1	1	0	0	-

ab \ cd	00	01	11	10
00	1	-	-	-
01	-	-	-	-
11	0	-	0	0
10	1	-	-	-

$w1 = \bar{b}$

ab \ cd	00	01	11	10
00	1	-	-	-
01	-	-	-	-
11	1	-	0	1
10	0	-	-	-

$w2 = \bar{a} + b\bar{d}$

ab \ cd	00	01	11	10
00	0	-	-	-
01	-	-	-	-
11	0	-	-	1
10	0	-	-	-

$w3 = c$

Rys. 3. Tablice Karnaguha oraz wzory specyfikacji przykładu 1

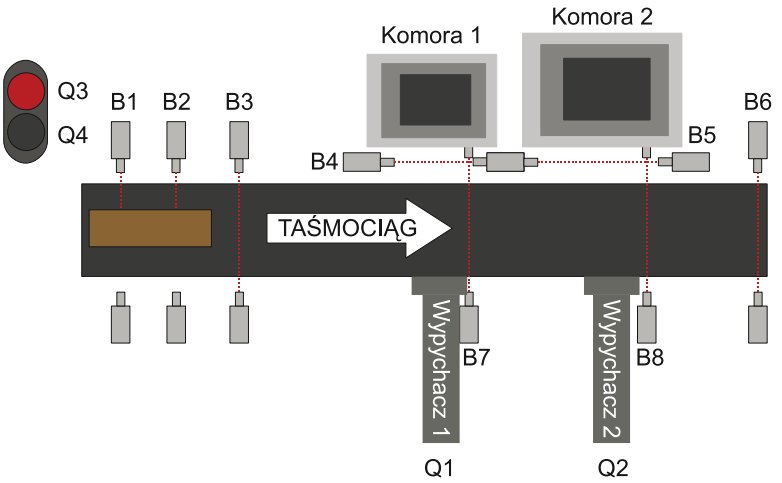
Kompletny program sterowania wraz z asercyjnym rozszerzeniem został umieszczony na listingu 1. Specyfikacja składa się z dwóch części – zgodnej ze standardem JML oraz z części spoza JML. Część zgodna zawiera klauzulę `requires` w której umieszczono dopuszczalne kombinacje wartości wejść, dla których zostały określone wartości wyjść. Klauzula `ensures` zawiera wzory specyfikujące zapisane w postaci równoważności (`<==>` w 8 linii listingu), z wyjątkiem wyjścia `w3`, które nie zostało określone dla temperatury $t \geq td$ (w tab. 4 i na rys. 3 oznaczone szarym tłem). Dla tego przypadku zastosowano w klauzuli `ensures` implikację (`==>` w 9 linii listingu). Część spoza standardu JML zawiera deklarację stanu bezpiecznego `stan_bezp` (l. 10-11), którego zadaniem jest przełączenie wyjść `w1`, `w2`, `w3` w stan 0 logicznego. Punkt wznowienia pracy (l. 12) ustalony został w punkcie końca kodu jednostki. Niespełnienie warunków początkowych lub końcowych (l. 12-13) wywoła obsługę stanu bezpiecznego `stan_bezp`.

Listing. 1. Kod programu sterującego grzałkami

```
01  PROGRAM termometr; VAR_EXTERNAL (*$AUTO*) END_VAR;
02  (*@REQUIRES: (a = FALSE & b = FALSE & c = FALSE & d = FALSE) OR
03               (a = TRUE  & b = FALSE & c = FALSE & d = FALSE) OR
04               (a = TRUE  & b = TRUE  & c = FALSE & d = FALSE) OR
05               (a = TRUE  & b = TRUE  & c = TRUE  & d = FALSE) OR
06               (a = TRUE  & b = TRUE  & c = TRUE  & d = TRUE);
07  ASSIGNS: w1, w2, w3;
08  ENSURES: (w1 <==> NOT b) & (w2 <==> (NOT a OR (b & NOT d))) &
09           (w3 ==> c);
10  SAFE_BEHAVIOUR stan_bezp: w1 := FALSE; w2 := FALSE; w3 := FALSE;
11                            resume_after_unit;
12  REQUIRES_FAILURE: stan_bezp;
13  ENSURES_FAILURE: stan_bezp;                                    *)
14
15   IF NOT a THEN    w1 := TRUE;      w2 := TRUE;       w3 := FALSE;
16   ELSIF NOT b THEN w1 := TRUE;      w2 := FALSE;      w3 := FALSE;
17   ELSIF NOT c THEN w1 := FALSE;     w2 := TRUE;       w3 := FALSE;
18   ELSIF NOT d THEN w1 := FALSE;     w2 := TRUE;       w3 := TRUE;
19   ELSE             w1 := FALSE;     w2 := FALSE;      w3 := FALSE;
20   END_IF
21
22  END_PROGRAM
```

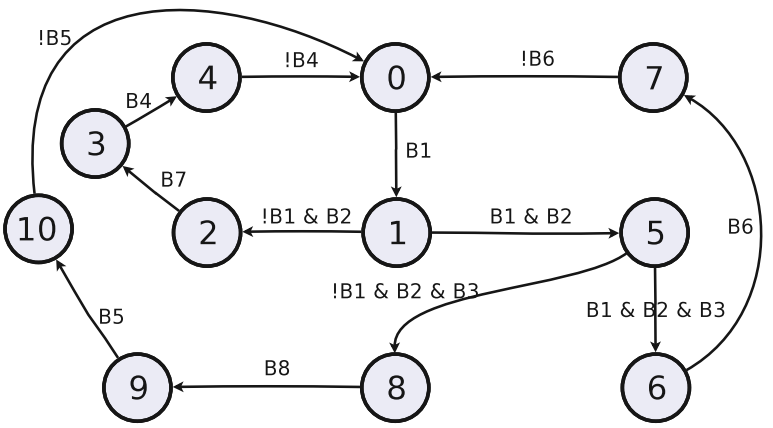
Tak adnotowany przykład skompilowany do weryfikacji dynamicznej, zapewnia elementarną ochronę przed niedozwolonymi wartościami czujników w trakcie wykonania, a także automatycznie przełączy wyjścia do stanu bezpiecznego w przypadku wykrycia niespójności pomiędzy specyfikacją układu sterowana, a jego realizacją. Jednocześnie została zachowana przejrzystość kodu pomiędzy częścią sterującą, a częścią odpowiedzialną za sprawdzenie poprawności wejść.

Przykład 2 zaczerpnięto z [4]. Przenośnik taśmowy transportuje kłody drewna (rys. 4). Na taśmociągu dokonuje się pomiaru długości kłody za pomocą czujników B1, B2, B3. Przysłonięcie jednego z czujników generuje na jego wyjściu logiczną jedynkę. Kłody krótkie, to takie których długość jest mniejsza od rozstawu czujników pomiaru długości (zakłada się jednakowy rozstaw czujników do pomiaru długości). Średnie mają długość od rozstawu czujników do podwojonej tej długości. Kłody długie mają długość większą niż podwojony rozstaw czujników. Zadaniem układu do sortowania drewna jest skierowanie kłód krótkich do komory 1, średnich do komory 2, a długich przetransportowanie do końca taśmociągu w celu dalszej obróbki. Do kierowania kłodami wykorzystuje się wypychacze, sterowane wyjściami Q1 i Q2. Pozycję kłody na taśmie wskazują czujniki B7 i B8, a opuszczenie przez kłodę taśmy czujniki B4, B5 i B6. Wprowadzenie następnej kłody na taśmociąg możliwe jest tylko wtedy, gdy poprzednia opuści go. Sygnalizowane jest to za pomocą świateł Q3 i Q4.



Rys. 4. Układ czujników w urządzeniu do sortowania drewna

Automat Moore'a sterujący urządzeniem do sortowania drewna wraz z tablicą wartości wyjść stanów przedstawiono na rys. 5. Znaczenie stanów jest następujące: **0** – taśmociąg pusty, można wprowadzić kłodę; **1** – początek pomiaru długości kłody; **2** – kłoda krótka, oczekiwanie na dojazd kłody do wypychacza 1; **3** – wypychanie kłody krótkiej; **4** – oczekiwanie na opuszczenie taśmociągu przez krótką kłodę; **5** – pomiar długości kłody; **6** – kłoda długa, oczekiwanie na dojazd kłody do końca taśmociągu; **7** – oczekiwanie na opuszczenie taśmociągu przez kłodę; **8** – kłoda średnia, oczekiwanie na dojazd do wypychacza 2; **9** – wypychanie kłody średniej; **10** – oczekiwanie na opuszczenie taśmociągu przez kłodę średnią.



	Q1	Q2	Q3	Q4
0	0	0	0	1
1	0	0	1	0
2	0	0	1	0
3	1	0	1	0
4	1	0	1	0
5	0	0	1	0
6	0	0	1	0
7	0	0	1	0
8	0	0	1	0
9	0	1	1	0
10	0	1	1	0

Rys. 5. Automat Moore'a i tablica wyjść stanów sortownicy

Automat zaimplementowano w języku ST jako dwie instrukcje wyboru (*case*), przedstawione na listingu 2. Pierwsza z nich (linia 25) odpowiada za przejścia pomiędzy stanami, a druga (l. 43) za ustawienie wartości wyjść otrzymanego stanu. Do programu dodano również adnotacje specyfikacji w postaci warunków wstępnych (l. 3), modyfikowanych zmiennych globalnych (l. 4), warunków końcowych (l. 5 – 17) i dwóch stanów bezpiecznych (l. 18 – 21). Niespełnienie warunku początkowego lub końcowego skojarzono ze stanem bezpiecznym `system_blad`, którego obsługa przełącza wszystkie wyjścia w stan 0 i powoduje zatrzymanie pracy sterownika. Warunek końcowy powstał na podstawie automatu i zawiera implikacje połączone koniunkcją. Każda z implikacji odpowiada jednemu przejściu pomiędzy stanami. Poprzednik implikacji składa się z wartości zmiennej `stan` z modyfikatorem `\old` wskazującym stan automatu w poprzednim cyklu, oraz warunku przejścia. Następnik implikacji składa się z nowego stanu automatu oraz wymaganym stanem wyjść. Wykrywanie niesprawności czujników odbywa się poprzez osobne asercje oznaczone w listingu szarym tłem. W przypadku wykrycia błędnych wskazań czujników zostanie wywołana obsługa stanu bezpiecznego `czujnik_blad`, którego zadaniem jest wyłączenie wypychaczy i zaświecenie obu lamp sygnalizujących niezdatność pracy układu.

Listing. 2. Kod programu realizujący układ sterowania sortownicą

```
01 PROGRAM sortownica; VAR_EXTERNAL (*$AUTO*) END_VAR;
02 VAR stan:INT; END_VAR
03 (*@REQUIRES: stan>=0 & stan<=10;
04 ASSIGNS: Q1, Q2, Q3, Q4;
05 ENSURES: (\old(stan)=0 & B1 ==> stan=1 & Q1=0 & Q2=0 & Q3=1 & Q4=0)
06 & (\old(stan)=1 & NOT B1 & B2 ==> stan=2 & Q1=0 & Q2=0 & Q3=1 & Q4=0)
07 & (\old(stan)=1 & B1 & B2 ==> stan=5 & Q1=0 & Q2=0 & Q3=1 & Q4=0)
08 & (\old(stan)=2 & B7 ==> stan=3 & Q1=1 & Q2=0 & Q3=1 & Q4=0)
09 & (\old(stan)=3 & B4 ==> stan=4 & Q1=1 & Q2=0 & Q3=1 & Q4=0)
10 & (\old(stan)=4 & NOT B4 ==> stan=0 & Q1=0 & Q2=0 & Q3=0 & Q4=1)
11 & (\old(stan)=5 & B1 & B2 & B3 ==> stan=6 & Q1=0 & Q2=0 & Q3=1 & Q4=0)
12 & (\old(stan)=5 & NOT B1 & B2 & B3 ==> stan=8 & Q1=0 & Q2=0 & Q3=1 &
Q4=0)
13 & (\old(stan)=6 & B6 ==> stan=7 & Q1=0 & Q2=0 & Q3=1 & Q4=0)
14 & (\old(stan)=7 & NOT B6 ==> stan=0 & Q1=0 & Q2=0 & Q3=0 & Q4=1)
15 & (\old(stan)=8 & B8 ==> stan=9 & Q1=0 & Q2=1 & Q3=1 & Q4=0)
16 & (\old(stan)=9 & B5 ==> stan=10 & Q1=0 & Q2=1 & Q3=1 & Q4=0)
17 & (\old(stan)=10 & NOT B5 ==> stan=0 & Q1=0 & Q2=0 & Q3=0 & Q4=1);
18 SAFE_BEHAVIOUR czujnik_blad: Q1:=0; Q2:=0; Q3:=1; Q4:=1;
19                              resume_after_unit;
20 SAFE_BEHAVIOUR system_blad: Q1:=0; Q2:=0; Q3:=0; Q4:=0;
21                              terminate_execution;
22 REQUIRES_FAILURE: system_blad;
23 ENSURES_FAILURE: system_blad;                                    *)
24
25 CASE stan OF
26 0: IF B1 THEN stan:=1; END_IF
27 1: (*@ASSERT(NOT (B1 AND NOT B2 AND B3), czujnik_blad); *)
28    IF B1 & B2 THEN stan:=5; ELSIF NOT B1 & B2 THEN stan:=2; END_IF
29 2: (*@ASSERT(NOT (B8 OR B6 OR B5), czujnik_blad); *)
30    IF B7 THEN stan:=3; END_IF
31 3: IF B4 THEN stan:=4; END_IF
32 4: IF NOT B4 THEN stan:=0; END_IF
33 5: (*@ASSERT(NOT (B1 AND NOT B2 AND B3), czujnik_blad); *)
34    IF B1 & B2 & B3 THEN stan:=6; ELSIF NOT B1 & B2 & B3 THEN stan:=8; END_IF
35 6: (*@ASSERT(NOT (B7 OR B8 OR B5 OR B4), czujnik_blad); *)
36    IF B6 THEN stan:=8; END_IF
37 7: IF NOT B6 THEN stan:=0; END_IF
```

```
38 | 8: (*@ASSERT(NOT (B7 OR B6 OR B4), czujnik_blad); *)
39 |    IF B8 THEN stan:=9; END_IF
40 | 9: IF B5 THEN stan:=10; END_IF
41 | 10: IF NOT B5 THEN stan:=0; END_IF
42 | END_CASE
43 | CASE stan OF
44 | 0: Q1:=0; Q2:=0; Q3:=0; Q4:=1;
45 | 1..2,5..8: Q1:=0; Q2:=0; Q3:=1; Q4:=0;
46 | 3,4: Q1:=1; Q2:=0; Q3:=1; Q4:=0;
47 | 9,10: Q1:=0; Q2:=1; Q3:=1; Q4:=0;
48 | END_CASE END_PROGRAM
```

W drugim przykładzie sprawdzenie poprawności wyjść czujników odbywa się w tych stanach, w których możliwe jest wykrycie błędów. Przekształcenie wykorzystanych asercji do pojedynczej klauzuli `requires` znacznie skomplikowałoby jej zapis, co uczyniłoby ją trudną do odczytania przez inne osoby.

6. PODSUMOWANIE

Przedstawiono asercyjne rozszerzenie języka ST do projektowania kontraktowego oraz weryfikacji dynamicznej systemów sterowania. Cechą charakterystyczną prezentowanego rozszerzenia jest podobieństwo do innych języków BISL, wykorzystywanych we współczesnym projektowaniu oprogramowania. Uzupełniając popularne klauzule standardu JML o deklaracje stanów bezpiecznych i punktów wznowienia, można przeprowadzić weryfikację dynamiczną opracowanego rozwiązania. Sprawdzanie asercji podczas wykonywania programu zwiększa bezpieczeństwo wykonywanego programu poprzez diagnostykę wejść układu i chroni przed niezamierzonym lub celowym wprowadzeniem błędnych parametrów. Zapis wykorzystujący adnotacje oddziela właściwy kod systemu sterowania od jego diagnostyki, co czyni go przejrzystym i lżejszym do analizy przez inne osoby.

Dalsze prace będą dotyczyły opracowania kompilatora do transformacji kodu języka ST z adnotacjami do ANSI C, oraz implementacji asercyjnego rozszerzenia do weryfikacji dynamicznej w pakiecie CPDev.

BIBLIOGRAFIA

1. Baudin P., Cuoq P., Filliâtre J.Ch., Marché C., Monate B., Moy Y., Prevosto V.: ACSL: ANSI/ISO C Specification Language. http://frama-c.cea.fr.
2. Bornat R.: Proving pointer programs in Hoare logic. Mathematics of Program Construction. MPC '00 Proceedings, pp. 102–126. Springer-Verlag, London, 2000.
3. Filliâtre J.Ch. The WHY verification tool. Tutorial and Reference Manual. http://www.lri.fr.
4. Kasprzyk J., Jegierski T., Wyrwał J., Hajda J.: Programowanie sterowników PLC. Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice, 1998.
5. Leavens G.: An Overview of Larch/C++: Behavioral Specifications for C++ Modules. [w:] Kilov H., Harvey W. (red): Specification of Behavioral Semantics in Object-Oriented Information Modeling. Kluwer Academic Publishers, 1996.
6. Leavens G., Baker A., Ruby C.: JML: a Notation for Detailed Design. [w:] Kilov H., Rumpe B., Simmonds I. (red): Behavioral Specifications of Businesses and Systems, 99.
7. Meyer B.: Applying “design by contract”. Computer, 25(10), pp. 40–51, 1992.
8. Meyer B.: Eiffel: The Language. Object-Oriented Series. Prentice Hall New York, 1992.
9. Świder Z. (red.): Sterowniki mikroprocesorowe. Oficyna Wydawnicza Politechniki Rzeszowskiej, Rzeszów, 2002.