RAPORT TECHNICZNY

Wprowadzenie do weryfikacji oprogramowania systemów automatyki z wykorzystaniem Frama C

Autor: Jan SADOLEWSKI

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Praca finansowana ze środków Unii Europejskiej, budżetu państwa oraz budżetu województwa podkarpackiego w ramach projektu: „Podkarpacki fundusz stypendialny dla doktorantów".

Rzeszów 2010

Wszelkie uwagi dotyczące raportu, pytania, propozycje systemów do weryfikacji, są mile widziane.
Można je przesyłać na adres js@prz-rzeszow.pl.

Słowa kluczowe: Frama C, Jessie, Coq, weryfikacja oprogramowania, lematy poprawności, warunki wstępne i końcowe Dijkstry, bezpieczeństwo przepełniena wartości przez zmienne.

Dokument opracowano wykorzystując system składu tekstu LaTeX (pdfTeXk), oraz inne ogólnodostępne programy m.in.: Scribus, Dia, OpenOffice.org, Gimp – działające w systemie Linux.

1 Wprowadzenie

Systemy sterowania przed wdrożeniem muszą być sprawdzone pod względem poprawności implementacji. Małe systemy wbudowane programowane są przeważnie w ANSI C. Pomimo tego że programy często pisane są przez zaawansowanych programistów, nie ma pewności że docelowy kod jest poprawny i nie zawiera efektów ubocznych. Wyszukiwanie błędów poprzez ręczną analizę kodu może być uciążliwe, a testy oprogramowania nigdy nie świadczą o braku błędów. Formalna weryfikacja zgodności pomiędzy specyfikacją a implementacją może pomóc w znalezieniu błędów oraz efektów ubocznych oprogramowania.

Niniejszy raport przedstawia proces weryfikacji dla prostych systemów sterowania zapisanych w ANSI C. Proces ten wykorzystuje ogólnodostępne oprogramowanie do analizy kodu: Frama C [5] z modułem Jessie [9] oraz Coq [3]. Frama C wspiera analizę kodu języka C zapisanego z adnotacjami specyfikacji w języku ACSL [2]. Frama zawiera wewnętrzny moduł do statycznej analizy wartości całego programu (lub jego znacznej części), lecz jest ona zbyt słaba nawet dla prostych programów. Słabość ta tłumaczy użycie modułu Jessie, który generuje lematy poprawności bazujące na metodzie warunków wstępnych Dijkstry. Lematy należą do dwóch grup – poprawności (ang. *ensures proof obligations*) oraz bezpieczeństwa zakresów (ang. *safety proof obligations*). Dowód lematów pierwszej grupy gwarantuje poprawność programu. Dowód lematów drugiej grupy wyraża zapewnienie że przekroczenie zakresów przez zmienne nie wystąpi podczas wykonywania programu. Dowody lematów będą przeprowadzone półautomatycznie w Coq.

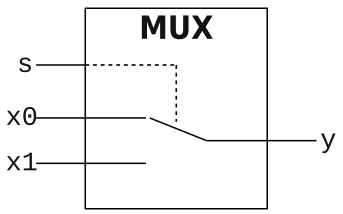
Raport przedstawia sposób wykorzystania Coq jako weryfikatora lematów zamiast narzędzia do modelowania. Do prezentacji wykorzystano łącznie sześć przykładów – programów realizujących układy kombinacyjne (multiplekser binarny oraz sterowanie grzałkami zależne od temperatury), programów układów sekwencyjnych (przerzutnik RS oraz wykrywanie ruchu wózka) oraz układów sekwencyjnych z uzależnieniami czasowymi (mrugający LED i winda towarowa). W raporcie przedstawiono metodę postępowania w celu przeprowadzenia weryfikacji podanych przykładów.

2 Układy kombinacyjne

Specyfikacja układów kombinacyjnych może być podana w następujących postaciach: za pomocą wzorów, w postaci słownej oraz tablicy wartości wyjść. Przykłady zaprezentowane tutaj używają pierwszej i drugiej postaci. Trzecia postać może być sprowadzona do wzorów za pomocą tablic Karnaugha [6].

2.1 Multiplekser binarny

Na rys. 1 przedstawiono multiplekser binarny z trzema wejściami oraz jednym wyjściem. Wejście `s` jest wykorzystywane do przełączania wyjścia pomiędzy wejściami `x0` oraz `x1`. Wzór specyfikujący pokazano w (1), kod realizujący funkcję multipleksera binarnego zamieszczono w listingu 1. Multiplekser zrealizowano jako funkcję języka C. W celu weryfikacji kodu analizatorem Frama C zamieszczony kod uzupełnia się o adnotacje specyfikacji zawarte w listingu 2. Andotacje te poprzedzają kod funkcji w procesie weryfikacji.



Rys. 1. Symbol multipleksera binarnego

$$y = (s \wedge x_0) \vee (!s \wedge x_1) \tag{1}$$

Specyfikacja wykorzystuje dwie klauzule `requires` oraz `ensures`. Pierwsza wyraża ograniczenie na wartości parametrów podczas wywołania funkcji (warunki wstępne - (ang. *preconditions*)). Druga wyraża ograniczenie jakie musi zostać spełnione po zwróceniu wartości przez funkcję (warunki końcowe - (ang. *postconditions*)). Zmienna `\result` reprezentuje wartość jaka została zwrócona przez wywołanie funkcji, tutaj wyjście multipleksera. Wzór (1) został umieszczony bezpośrednio w klauzuli `ensures`.

Listing 1.
```
char mux(char s, char x0, char x1)
{
  char y;
  if(s == 1)
    y = x0;
  else
    y = x1;
  return y;
}
```

Listing 2.
```
requires (s==0 || s==1) && (x0==0 || x0==1) && (x1==0 || x1==1);
ensures (\result <==> (s && x0) || (!s && x1));
```

Program Frama C uruchamia się z następującymi parametrami:

```
frama-c -jessie mux.c
```

W przypadku gdy tryb graficzny nie jest poprawnie obsługiwany, wygenerowany przez Frama C skrypt zakończy się błędem. Należy wtedy wykonać dodatkowe polecenia do przygotowania skryptu weryfikacyjnego:

```
cd mux.jessie
make -f mux.makefile coq
cd coq
```

Graficzne narzędzie Coq do weryfikacji uruchamia się za pomocą polecenia:

```
coqide mux_why.v
```

Domyślną taktyką dowodu jest `intuition`. Automatyczną weryfikację uaktywnia się za pomocą menu: 'Navigation | End'. Weryfikator zatrzyma się na pierwszym lemacie, który nie został automatycznie udowodniony. W tym przykładzie będzie to `mux_ensures_default_po_1` przedstawiony na rys. 2. W prawej części rysunku znajdują się aktualne podcele do udowodnienia, hipotezy - umieszczone nad kreską oraz szczegóły podcelu pod kreską. Taktyka `intuition` przekształciła lemat do prostszej postaci agregując w kontekście dodatkowe hipotezy. Obecny podcel jest alternatywą, dla której stosujemy taktyki `left` lub `right` zostawiające lewą lub prawą część alternatywy do dalszego dowodu. Tutaj będzie to `left`. Po jej zastosowaniu bieżący podcel będzie koniunkcją. Stosujemy w tym celu taktykę `split` aby rodzielić cel na dwa nowe podcele. Obecny podcel `integer_of_int8 s = 0 -> False` jest implikacją o fałszywym następniku. Poprzednik tej implikacji jest sprzeczny z założeniami kontekstu `H0 : integer_of_int8 s = 1`. Ponowne zasosowanie taktyki `intuition` znajdzie sprzeczność i podcel zostanie udowodniony. Biężacym podcelem stanie się `integer_of_int8 x0 = 0 -> False`. Wykonując podstawienie hipotezy będącej równością (`HW_6`) za pomocą taktyki `rewrite <- HW_6`, a następnie zostanie analogicznie użyta równość `HW_7`, to bieżący cel będzie zgodny z hipotezą `HW_8`. Zgodność celu z hipotezą dowodzi się taktyką `assumption`. Powtarzając te same operacje dla drugiego podcelu lemat `mux_ensures_default_po_1` zostanie udowodniony.

Lemat `mux_ensures_default_po_2` podobnie dzieli się na dwa podcele `False`. Hipotezy kontekstu `H4` i `H5` są implikacjami o następniku zgodnym z podcelem, dlatego możliwe jest zastosowanie taktyki `apply H5`. Taktyka ta zmieni cel na poprzednik przekazanej jako argument implikacji – `integer_of_int8 x0 = 0`. Pozwoli to na wykorzystanie taktyk z poprzedniego lematu:

```
rewrite <- HW_6.
rewrite <- HW_7.
assumption.
```

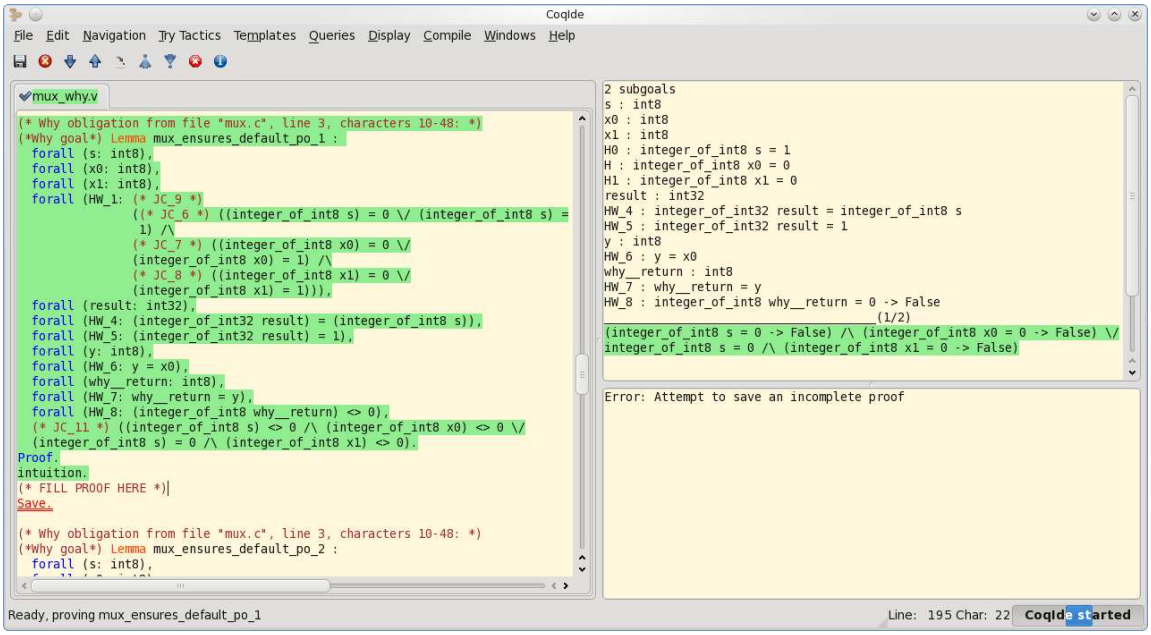
Aplikując ten ciąg dla pozostałego podcelu lemat `mux_ensures_default_po_2` zostaje udowodniony.

Dowody pozostałych lematów przebiegają podobnie, umieszczono je w listingu 3.

Listing 3.

```
Lemma mux_ensures_default_po_3 :
  forall (s: int8),
  forall (x0: int8),
  forall (x1: int8),
  forall (HW_1: (((integer_of_int8 s) = 0 \/
         (integer_of_int8 s) = 1) /\
         ((integer_of_int8 x0) = 0 \/
         (integer_of_int8 x0) = 1) /\
         ((integer_of_int8 x1) = 0 \/
         (integer_of_int8 x1) = 1))),
  forall (result: int32),
  forall (HW_4: (integer_of_int32 result) =
                      (integer_of_int8 s)),
  forall (HW_10: (integer_of_int32 result)
                                     <> 1),
  forall (y: int8),
  forall (HW_11: y = x1),
  forall (why__return: int8),
  forall (HW_12: why__return = y),
  forall (HW_13: (integer_of_int8
         why__return) <> 0),
         ((integer_of_int8 s) <> 0 /\
         (integer_of_int8 x0) <> 0 \/
         (integer_of_int8 s) = 0 /\
         (integer_of_int8 x1) <> 0).
Proof.
intuition.
(* FILL PROOF HERE *)
right. split. assumption.
rewrite <- HW_11.
rewrite <- HW_12.
assumption.
right. split. assumption.
rewrite <- HW_11.
rewrite <- HW_12.
assumption.
Save.
```

```
Lemma mux_ensures_default_po_4 :
  forall (s: int8),
  forall (x0: int8),
  forall (x1: int8),
  forall (HW_1: (((integer_of_int8 s) = 0 \/
         (integer_of_int8 s) = 1) /\
         ((integer_of_int8 x0) = 0 \/
         (integer_of_int8 x0) = 1) /\
         ((integer_of_int8 x1) = 0 \/
         (integer_of_int8 x1) = 1))),
  forall (result: int32),
  forall (HW_4: (integer_of_int32 result) =
                      (integer_of_int8 s))
  forall (HW_10: (integer_of_int32 result)
                                     <> 1),
  forall (y: int8),
  forall (HW_11: y = x1),
  forall (why__return: int8),
  forall (HW_12: why__return = y),
  forall (HW_14: (integer_of_int8 s) <> 0 /\
         (integer_of_int8 x0) <> 0 \/
         (integer_of_int8 s) = 0 /\
         (integer_of_int8 x1) <> 0),
         (integer_of_int8 why__return) <> 0.
Proof.
intuition.
(* FILL PROOF HERE *)
apply H5.
rewrite <- HW_11.
rewrite <- HW_12.
assumption.
apply H5.
rewrite <- HW_11.
rewrite <- HW_12.
assumption.
Save.
```



Rys. 2. Środowisko graficzne weryfikatora Coq

2.2 Sterowanie grzałkami

Termometr kontaktowy (rys. 3a) generuje sygnały `a`, `b`, `c`, `d`, gdy temperatura przekroczy odpowiednie wartości. Układ sterujący powinien tak generować sygnały dla przełączników `w1`, `w2`, `w3`, aby spełnić wymagane warunki włączeń w zależności od temperatury t:

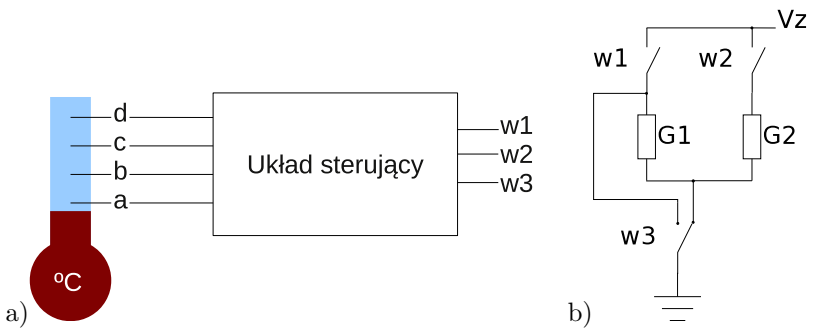
- $t < ta$ – oba grzejniki włączone równolegle,
- $ta \leqslant t < ta$ – włączony tylko grzejnik G1,
- $tb \leqslant t < tc$ – włączony tylko grzejnik G2,
- $tc \leqslant t < td$ – oba grzejniki włączone szeregowo,
- $t \geqslant td$ – oba grzejniki wyłączone.

Styki wyłączników na rys. 3b narysowano w pozycji 0. Przekroczenie każdego progu temperatury jest sygnalizowane stanem 1 odpowiednich czujników (`a` – `d`). Po utworzeniu pierwotnej tablicy stanów wejść i odpowiadającym im stanów wyjść przystąpiono do otrzymania równań specyfikujących zachowanie programu poprzez minimalizację za pomocą tablic Karnaugha. Równania te podano w (2). Kod źródłowy programu realizującego sterowanie przełącznikami przedstawiono w listingu 4. Wszystkie zmienne w nim występujące zostały zadeklarowane globalnie typem `char`. Prawidłowe działanie termometru generuje 5 zestawów sygnałów, które połączono alternatywą i umieszczono w klauzuli `requires` adnotacji specyfikacji pokazanej na listingu 5. Klauzula `assigns` wskazuje modyfikowane zmienne globalne, a klauzula `ensures` zawiera specyfikację ze wzoru (2) zapisaną w postaci dwóch równoważności oraz jednej implikacji ponieważ dla tego wyjścia zakreślona ósemka w tabeli Karnaugha zawierała stany nieistotne.

$$w_1 = !b \qquad w_2 = !a \vee (b \wedge !d) \qquad w_3 = c \tag{2}$$

Weryfikacja programu za pomocą modułu Jessie wygeneruje 22 lematy poprawności. Lematy bezpieczeństwa przekroczenia zakresów nie zostaną wygenerowane, ponieważ w programie występują tylko podstawienia do zmiennych o wartościach stałych. Powstałe lematy dowodzą się jednym z trzech poniższych sposobów:

1. Tylko taktyką `intuition`.
2. Sekwencją taktyk `intros`, `rewrite _ in _`, `contradiction`.
3. Sekwencją taktyk `intros`, `rewrite`, `omega`.



Rys. 3. Układ sterowania grzałkami a) schemat poglądowy, b) połączenie grzałek

Listing 4.

```
void termometr()
{
   if(!a) {          w1 = 1;        w2 = 1;        w3 = 0;    }
   else {
      if(!b) {       w1 = 1;        w2 = 0;        w3 = 0;    }
      else {
         if(!c) {    w1 = 0;        w2 = 1;        w3 = 0;    }
         else {
            if(!d) { w1 = 0;        w2 = 1;        w3 = 1;    }
            else {   w1 = 0;        w2 = 0;        w3 = 0;    }
              }
         }
      }
}
```

W niektórych przypadkach sposoby o numerach 2 i 3 mogą wymagać dodatkowych taktyk takich jak `left`, `right` i `decompose`. Taktykę `decompose` wykorzystuje się w przypadku gdy hipoteza jest koniunkcją (`[and]`) lub alternatywą (`[or]`). Dekompozycja hipotezy będącej koniunkcją wyrażeń generuje dodatkowe hipotezy z części składowych, a dekompozycja hipotezy będącej alternatywą wyrażeń generuje dodatkowe podcele do udowodnienia.

Inne przypadki mogą być udowodnione podobnie przy zachowaniu prezentowanej metody postępowania.

3 Układy sekwencyjne

W układach sekwencyjnych wyjście zależy od bieżącego stanu wejść oraz od wewnętrznego stanu z poprzedniego cyklu obliczeń. Jedno z wejść (zegarowe) jest dostępne dla wszystkich elementów i tylko na jego zboczu jest możliwa zmiana stanów wewnętrznych oraz wejść. Implementacja takich progamów jako funkcje języka C wymaga dodatkowych parametrów zazwyczaj wskaźników. Ich dereferencja służy do odczytania stanu z poprzedniego cykl i zapisania nowego stanu. Specyfikacja układów może być w postaci tekstowej, wzorów lub utworzona na podstawie przebiegów czasowych czy grafu automatu.

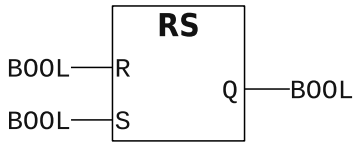
3.1 Przerzutnik RS

Specyfikacja przerzutnika (rys. 4) podana została słownie:
1 logiczne na wejściu `R` *przełącza wyjście* `Q` *do wartości 0, 1 na wejściu* `S` *ustawia* `Q` *na wartość 1, lecz*

Listing 5.

```
requires (a == 0 && b == 0 && c == 0 && d == 0) || (a == 1 && b == 0 && c == 0 && d == 0) ||
         (a == 1 && b == 1 && c == 0 && d == 0) || (a == 1 && b == 1 && c == 1 && d == 0) ||
         (a == 1 && b == 1 && c == 1 && d == 1);
assigns w1, w2, w3;
ensures (!b <==> w1) && ((!a || (b && !d)) <==> w2) && (w3 ==> c);
```

tylko wtedy gdy `R` *jest 0, gdy oba wejścia* `R` *i* `S` *mają wartość 0 wtedy wartość* `Q` *nie zmienia się, natomiast gdy oba wejścia mają wartość 1 to wejście* `R` *jest dominujące i przełącza* `Q` *do 0.*



Rys. 4. Symbol przerzutnika RS

Specyfikacja zapisana w postaci kodu została umieszczona w listingu 6. Klauzula `requires` zawiera modyfikator `\valid`, który wskazuje, że zmienna `state` nie będzie miała wartości `NULL` w czasie wykonania funkcji. Klauzula `assigns` musi zawierać zmienną `state` nawet wtedy gdy jest zadeklarowana jako lokalny wskaźnik (gdyż taki wskaźnik może modyfikować zmienne globalne). Modyfikator `\old` występujący w klauzuli `ensures` wskazuje na wartość zmiennej z początku wykonywania funkcji (w większości przypadków jest to zmienna mająca wartość z poprzedniego cyklu).

Listing 6.
```
requires \valid(state) && (R==0||R==1) && (S==0||S==1) && (*state==0||*state==1);
assigns *state;
ensures ((\old(*state)==0) && (\old(S)==0) ==> (\result==0) && (*state==0)) &&
        ((\old(*state)==0) && (\old(S)==1) && (\old(R)==0) ==>
                                              (\result==1) && (*state==1)) &&
        ((\old(*state)==1) && (\old(R)==0) ==> (\result==1) && (*state==1)) &&
        ((\old(*state)==1) && (\old(R)==1) ==> (\result==0) && (*state==0));
```

Listing 7.
```
char rs_ff_aut(char R, char S, char* state)
{
  switch(*state)
  {
    case 0:
      if(S == 1 && R == 0) { *state = 1; return 1; }
      return 0;
    break;
    case 1:
      if(R == 1) { *state = 0; return 0; }
      return 1;
    break;
  }
}
```

Na listingu 7 pokazano implementację przerzutnika RS. W wyniku analizy Frama C i Jessie otrzymuje się 46 lematów weryfikujących poprawność oraz 9 weryfikujących bezpieczeństwo przepełnienia zmiennych w trakcie wykonywania programu. Spośród lematów poprawności większość można udowodnić za pomocą taktyki `intuition`, albo metod pokazanych wcześniej. Problemy pojawiają się gdy należy udowodnić lematy operujące na wskaźnikach. Przykładowy lemat wraz z dowodem przedstawiono na listingu 8. Po zastosowaniu domyślnej taktyki cel zmienia się do postaci `integer_of_int8 (select char_P_char_M_stan_1_0 stan) = 1`. W tej sytuacji należy zastąpić modyfikowaną zmienną `char_P_char_M_stan_1_0` jej wartością z kontekstu. Dokonuje się tego za pomocą taktyki `subst char_P_char_M_stan_1_0`. Zapis `select _ _` oznacza ekstrakcję z modelu pamięci będącym pierwszym argumentem wartości zmiennej będącej drugim argumentem. Podobnie zapis `store _ _ _` oznacza aktualizację modelu pamięci będącego pierwszym argumentem pod adresem zmiennej (drugi argument) wartością przekazaną jako trzeci argument. Zapis `select (store _ A V) A` można uprościć do postaci `V` za pomocą taktyki `rewrite select_store_eq`, która dodatkowo wygeneruje trywialny podcel `A = A`.

Nieco inna sytuacja występuje w przypadku dowodu lematu, który odpowiada za modyfikację wspólnej pamięci przez dwa wskaźniki. Przykład takiego lematu zaprezentowano na listingu 9. Zastosowanie taktyki `intuition` powoduje powstanie 8 identycznych podcelów, co niekoniecznie jest właściwym podejściem do przeprowadzenia dowodu. Wykorzystanie taktyki `intros` da w tym przypadku podobne efekty

Listing 8.

```
Lemma rs_ff_aut_ensures_default_po_4 :
  forall (R: int8),
  forall (S: int8),
  forall (stan: (pointer char_P)),
  forall (char_P_stan_1_alloc_table: (alloc_table char_P)),
  forall (char_P_char_M_stan_1: (memory char_P int8)),
  forall (HW_1: ((offset_min char_P_stan_1_alloc_table stan) <= 0 /\
                 (offset_max char_P_stan_1_alloc_table stan) >= 0 /\
                 ((integer_of_int8 R) = 0 \/ (integer_of_int8 R) = 1) /\
                 ((integer_of_int8 S) = 0 \/ (integer_of_int8 S) = 1) /\
                 ((integer_of_int8 (select char_P_char_M_stan_1 stan)) = 0 \/
                 (integer_of_int8 (select char_P_char_M_stan_1 stan)) = 1))),
  forall (result: int8),
  forall (HW_4: result = (select char_P_char_M_stan_1 stan)),
  forall (result0: int32),
  forall (HW_5: (integer_of_int32 result0) = (integer_of_int8 result)),
  forall (HW_6: (integer_of_int32 result0) = 0),
  forall (result1: int32),
  forall (HW_7: (integer_of_int32 result1) = (integer_of_int8 S)),
  forall (HW_8: (integer_of_int32 result1) = 1),
  forall (result2: int32),
  forall (HW_9: (integer_of_int32 result2) = (integer_of_int8 R)),
  forall (HW_10: (integer_of_int32 result2) = 0),
  forall (result3: int8),
  forall (HW_11: (integer_of_int8 result3) = 1),
  forall (char_P_char_M_stan_1_0: (memory char_P int8)),
  forall (HW_12: char_P_char_M_stan_1_0 =
                 (store char_P_char_M_stan_1 stan result3)),
  forall (result4: int8),
  forall (HW_13: (integer_of_int8 result4) = 1),
  forall (__retres: int8),
  forall (HW_14: __retres = result4),
  forall (why__return: int8),
  forall (HW_15: why__return = __retres),
  forall (HW_17: (integer_of_int8 (select char_P_char_M_stan_1 stan)) = 0 /\
                 (integer_of_int8 S) = 1 /\ (integer_of_int8 R) = 0),
  (integer_of_int8 (select char_P_char_M_stan_1_0 stan)) = 1.
Proof.
intuition.
(* FILL PROOF HERE *)
subst char_P_char_M_stan_1_0.
rewrite select_store_eq.
assumption.
trivial.
Save.
```

jak `intuition` bez dodatkowych podcelów. Kolejnym krokiem jest eliminacja modyfikowanej zmiennej przez `subst` (jak poprzednio). Jako pierwsza w celu występuje klauzula `not_assigns`, której przeznaczeniem jest wykazanie, że dowolna zmienna (tutaj oznaczona jako `A`) spoza jednolementowego zbioru zmiennych nie zmodyfikuje zawartości pamięci funkcji. Można to udowodnić następująco:
`intros A B` – wprowadzenie hipotez `A` i `B` do kontekstu,
`decompose [and] B` – rozdzielenie hipotezy `B` na dwie składowe koniunkcji.
Rozdzieleinie hipotezy `B` dodaje dwie nowe hipotezy `H: valid char_P_stan_1_alloc_table A` i `H0: ~in_pset A (pset_singleton stan)`. Za pomocą taktyki `rewrite in_pset_singleton in H0` hipoteza H0 zmienia się do postaci `A <> stan`. Taktyka `rewrite select_store_neq` zmieni cel do prostej równości osiągalnej taktyką `trivial`. Pozostały podcel `stan <> A` zbliżony do hipotezy `H0` można udowodnić taktyką `auto`.

Listing 9.

```
Lemma rs_ff_aut_ensures_default_po_9 :
  forall (R: int8),
  forall (S: int8),
  forall (stan: (pointer char_P)),
  forall (char_P_stan_1_alloc_table: (alloc_table char_P)),
  forall (char_P_char_M_stan_1: (memory char_P int8)),
  forall (HW_1: ((offset_min char_P_stan_1_alloc_table stan) <= 0 /\
                 (offset_max char_P_stan_1_alloc_table stan) >= 0 /\
                 ((integer_of_int8 R) = 0 \/ (integer_of_int8 R) = 1) /\
                 ((integer_of_int8 S) = 0 \/ (integer_of_int8 S) = 1) /\
                 ((integer_of_int8 (select char_P_char_M_stan_1 stan)) = 0 \/
                 (integer_of_int8 (select char_P_char_M_stan_1 stan)) = 1))),
  forall (result: int8),
  forall (HW_4: result = (select char_P_char_M_stan_1 stan)),
  forall (result0: int32),
  forall (HW_5: (integer_of_int32 result0) = (integer_of_int8 result)),
  forall (HW_6: (integer_of_int32 result0) = 0),
  forall (result1: int32),
  forall (HW_7: (integer_of_int32 result1) = (integer_of_int8 S)),
  forall (HW_8: (integer_of_int32 result1) = 1),
  forall (result2: int32),
  forall (HW_9: (integer_of_int32 result2) = (integer_of_int8 R)),
  forall (HW_10: (integer_of_int32 result2) = 0),
  forall (result3: int8),
  forall (HW_11: (integer_of_int8 result3) = 1),
  forall (char_P_char_M_stan_1_0: (memory char_P int8)),
  forall (HW_12: char_P_char_M_stan_1_0 =
                 (store char_P_char_M_stan_1 stan result3)),
  forall (result4: int8),
  forall (HW_13: (integer_of_int8 result4) = 1),
  forall (__retres: int8),
  forall (HW_14: __retres = result4),
  forall (why__return: int8),
  forall (HW_15: why__return = __retres),
  (not_assigns
   char_P_stan_1_alloc_table char_P_char_M_stan_1 char_P_char_M_stan_1_0 (
   pset_singleton stan)).
Proof.
intros.
(* FILL PROOF HERE *)
subst char_P_char_M_stan_1_0.
intros A B.
decompose [and] B.
rewrite in_pset_singleton in H0.
rewrite select_store_neq.
trivial.
auto.
Save.
```

Następnym przykładem dowodzenia lematów jest jawne wykazanie sprzeczości hipotez. Przedstawiony na listingu 10 lemat, po zastosowaniu taktyki `intuition` generuje hipotezy które nie są zgodne z celem dowodu. Widoczne jest to po odpowiednim przekształceniu celu za pomocą taktyk `rewrite HW_31` oraz `rewrite HW_30`. W tej sytuacji hipoteza `HW_29` zakłada, że `integer_of_int8 result2 = 0` a celem do udowodnienia jest `integer_of_int8 result2 = 1`, czego dowód nie jest możliwy. W tej sytuacji jedynym rozwiązaniem jest wykazanie że hipotezy kontekstu są sprzeczne.

Listing 10.

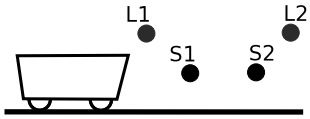
```
Lemma rs_ff_aut_ensures_default_po_20 :
  forall (R: int8),
  forall (S: int8),
  forall (stan: (pointer char_P)),
  forall (char_P_stan_1_alloc_table: (alloc_table char_P)),
  forall (char_P_char_M_stan_1: (memory char_P int8)),
  forall (HW_1: ((offset_min char_P_stan_1_alloc_table stan) <= 0 /\
                 (offset_max char_P_stan_1_alloc_table stan) >= 0 /\
                 ((integer_of_int8 R) = 0 \/ (integer_of_int8 R) = 1) /\
                 ((integer_of_int8 S) = 0 \/ (integer_of_int8 S) = 1) /\
                 ((integer_of_int8 (select char_P_char_M_stan_1 stan)) = 0 \/
                 (integer_of_int8 (select char_P_char_M_stan_1 stan)) = 1))),
  forall (result: int8),
  forall (HW_4: result = (select char_P_char_M_stan_1 stan)),
  forall (result0: int32),
  forall (HW_5: (integer_of_int32 result0) = (integer_of_int8 result)),
  forall (HW_6: (integer_of_int32 result0) = 0),
  forall (result1: int32),
  forall (HW_7: (integer_of_int32 result1) = (integer_of_int8 S)),
  forall (HW_28: (integer_of_int32 result1) <> 1),
  forall (result2: int8),
  forall (HW_29: (integer_of_int8 result2) = 0),
  forall (__retres: int8),
  forall (HW_30: __retres = result2),
  forall (why__return: int8),
  forall (HW_31: why__return = __retres),
  forall (HW_34: (integer_of_int8 (select char_P_char_M_stan_1 stan)) = 1 /\
                  (integer_of_int8 R) = 0),
  (integer_of_int8 why__return) = 1.
Proof.
intros.
(* FILL PROOF HERE *)
decompose [and] HW_34.
rewrite <- HW_4 in H.
rewrite <- HW_5 in H.
rewrite HW_6 in H.
absurd (0=1).
omega.
assumption.
Save.
```

Po wycofaniu użytych taktyk, stosuje się `intros`, a następnie dekomponuje hipotezę `HW_34`. Zastosowanie `rewrite <- HW_4 in H` zastąpi odwołanie do pamięci nazwą zmiennej. Podobnie `rewrite <- HW_5 in H` przekształci wartości liczbowe o różnych zakresach (ilości bitów). Ostatnie przekształcenie `rewrite HW_6 in H` zostawi wartości liczbowe zamiast symboli generując hipotezę `H: 0 = 1`. Sprzeczność hipotezy określana jest za pomocą taktyki `absurd`, a dowód przeprowadzi taktyka `omega`. Pozostały cel jest efektem działania taktyki `absurd`, a jego dowód sprowadza się do `assumption`.

Lematy bezpieczeństwa przekroczenia zakresów przez zmienne są proste. Siedem spośród nich udowadnia się automatycznie za pomocą taktyki `intuition`. Pozostałe dwa mogą być udowodnione sekwencją `intros`, `rewrite _`, `omega`.

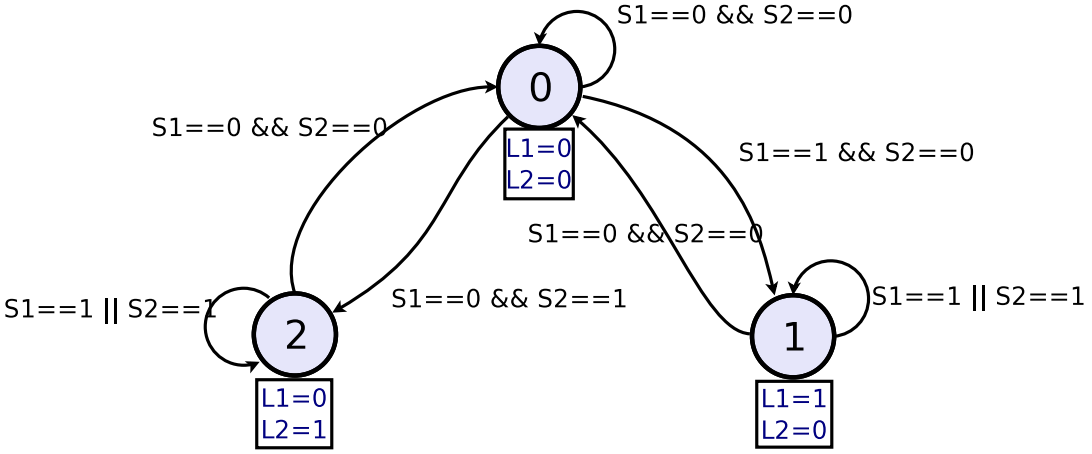
3.2 Wykrywanie kierunku ruchu wózka

Przemieszczanie się wózka w lewo lub w prawo jest wykrywane przez dwa czujniki optyzne `S1`, `S2` oraz sygnalizowane przez diody `L1`, `L2` (rys. 5. Jeżeli wózek przemieszcza się w lewo to świeci dioda `L1`. Odległość pomiędzy czujnikami jest mniejsza niż długość wózka. Przyjmuje się że wózek może zatrzymać się (i zawrócić) tylko poza zasięgiem czujników.



Rys. 5. Wykrywanie kierunku ruchu wózka

Specyfikację układu pokazano jako automat Moorea na rys. 6. Reprezentację w postaci adnotacji do kodu przedstawiono na listingu 11. Podobnie jak poprzednio, przyjęto, że wszystkie zmienne są typu `char` oraz zostały zadeklarowane globalnie, w celu uproszczenia dowodów lematów poprawności. Na listingu 12 przedstawiono implementację programu sterowania.



Rys. 6. Wykrywanie kierunku ruchu wózka

Listing 11.

```
requires (state >= 0) && (state <= 2) && (cl==0 || cl==1) && (cp==0 || cp==1);
assigns state, l1, l2;
ensures ((\old(state)==0 && (\old(cl)==0) && (\old(cp)==0)) ==> (state==0)) &&
        ((\old(state)==0 && (\old(cl)==1) && (\old(cp)==0)) ==> (state==1)) &&
        ((\old(state)==0 && (\old(cl)==0) && (\old(cp)==1)) ==> (state==2)) &&
        ((\old(state)==1 && (\old(cl)==1 || \old(cp)==1)) ==> (state==1)) &&
        ((\old(state)==1 && (\old(cl)==0 && \old(cp)==0)) ==> (state==0)) &&
        ((\old(state)==2 && (\old(cl)==1 || \old(cp)==1)) ==> (state==2)) &&
        ((\old(state)==2 && (\old(cl)==0 && \old(cp)==0)) ==> (state==0));
```

Analiza modułem Jessie generuje 97 lematów poprawności oraz dwa lematy bezpieczeństwa zakresów zmiennych. Ponieważ zostały użyte zmienne goblane, to 89 lematów z pierwszej grupy zostaje udowodnione automatycznie taktyką `intuition`. Pozostałe 8 lematów poprawności dowodzą się po zastosowaniu dodatkowych taktyk `rewrite _` i `assumption`. Lematy bezpieczeństwa zakresów dowodzi taktyka `intuition`.

Weryfikacja układów sekwencyjnych za pomocą Frama C z modułem Jessie nie różni się znacznie od układów kombinacyjnych. Różnicę stanowi wykorzystanie wskaźników w deklaracji funkcji. Prowadzi to do nieco skomplikowanych lematów weryfikujących poprawność, które wymagają wiedzy z dowodzenia arytmetyki wskaźników Jessie za pomocą Coq.

Listing 12.

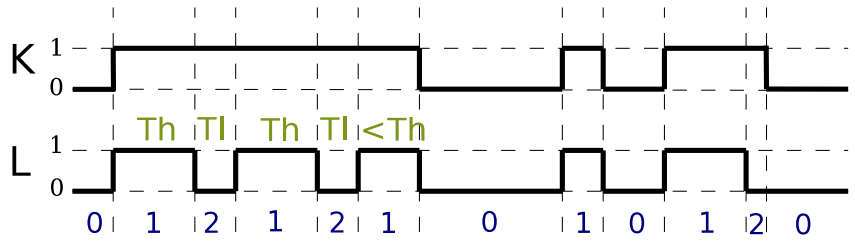
```
void truck_movement()
{
  switch(state)
  {
     case 0:
            l1 = 0;      l2 = 0;
            if(cl && !cp)     state = 1;
            if(!cl && cp)     state = 2;
            if(!cl && !cp)    state = 0;
            break;
     case 1:
            l1 = 1;      l2 = 0;
            if(cl || cp)      state = 1;
            else              state = 0;
            break;
     case 2:
            l1 = 0;      l2 = 1;
            if(cl || cp)      state = 2;
            else              state = 0;
            break;
  }
}
```

4 Układy sekwencyjne z uzależnieniami czasowymi

Implementacja sprzętowa układów sekwencyjnych z uzależnieniami czasowymi wykorzystuje przedziały czasowe zwane *czasem cyklu*. Pomiary czasu mogą być tylko całkowitą wielokrotnością czasu cyklu. Specyfikacja tych układów nie różni się znacząco od poprzednich przykładów z wyjątkiem określania czasu na przejściach pomiędzy stanami automatu. Warunki przejścia zapisywane są nad kreską ułamkową, a operacje na czasomierzach poniżej kreski.

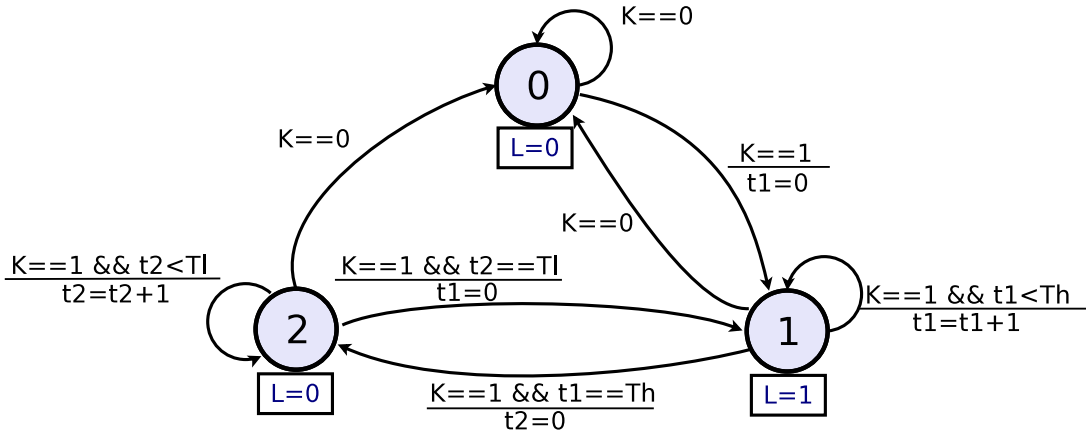
4.1 Błyskający LED

Dioda `L` błyska kiedy klawisz `K` jest przyciśnięty (rys. 7). `Th` i `Tl` są odmierzanymi długościami czasu kiedy LED jest włączony lub wyłączony. Ponadto dioda wyłącza się natychmiast po puszczeniu klawisza. Automat przedstawiony na rys. 8 wyprowadzono bezpośrednio z przebiegów czasowych. Specyfikację układu umieszczono w listingu 13. Można zauważyć, że wyrażenie `K==0` opisuje każde przejście do stanu 0. Pozwala to na zapisanie tylko jednej części w klauzuli `ensures` (dla `K==0`). Niektóre z klauzul specyfikacji wykonują rówież operacje na timerach.



Rys. 7. Przebiegi czasowe sygnałów błyskającej diody LED

Analiza modułem Jessie geneuje 209 lematów poprawności oraz 28 lematów bezpieczeństwa wartości zmiennych. Tylko kilka z lematów poprawności nie może być udowodnione taktyką `intuition`. Te wyjątki można udowodnić w podobny sposób jak w przykładzie z multiplekserem oraz wykrywaniem ruchu wózka. Lematy bezpieczeństwa nie są już takie trywialne. Cztery z nich nie może być udowodnionych ponieważ klauzula `requires` jest zbyt słaba. Również problemem jest sytuacja, że Jessie traktuje stałe `Th` oraz `Tl` jako zmienne, dla których nie ma hipotezy `t1 <= Th` i `t2 < Tl`, ani `t1 <= 126`, `t2 <= 126`. Hipotezy te gwarantują bezpieczeństwo wartości zmiennych typu `char` dla rozważanego programu. Wzmocnienie klauzuli `requires` wspomnianymi hipotezami pozwoli dowieść, że przekroczenie zakresów nie wystąpi.



Rys. 8. Automat stanów błyskającej diody LED

Listing 13.

```
requires ((K==0) || (K==1)) && ((state>=0) && (state<=2)) && (t1>=0) && (t2>=0)
            && (t1<=T1) && (t2<=T2);
assigns L, state, t1, t2;
ensures ((K==0) ==> (state==0)) &&
((K==1) && (\old(state)==0) ==> (state==1) && (t1==0)) &&
((K==1) && (\old(state)==1) && (\old(t1)<Th) ==> (state==1) && (t1==\old(t1)+1)) &&
((K==1) && (\old(state)==1) && (\old(t1)==Th) ==> (state==2) && (t2==0)) &&
((K==1) && (\old(state)==2) && (\old(t2)<Tl) ==> (state==2) && (t2==\old(t2)+1)) &&
((K==1) && (\old(state)==2) && (\old(t2)==Tl) ==> (state==1) && (t1==0));
```

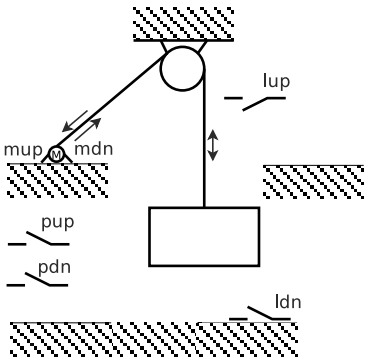
Listing 14.

```
void flashingled()
{
 switch(state)
 {
   case 0:
     L = 0;
     if(K == 1) {    state = 1;      t1 = 0; }
     break;
   case 1:
     L = 1;
     if(K == 0) {    state = 0; } else
     if(K==1 && t1<Th) {    state = 1; t1++; } else
     if(K==1 && t1==Th) {   state = 2; t2 = 0; }
     break;
   case 2:
     L = 0;
     if(K == 0) {    state = 0; } else
     if(K==1 && t2<Tl) {    state = 2; t2++; } else
     if(K==1 && t2==Tl) {   state = 1; t1 = 0; }
     break;
  }
}
```

4.2 Winda towarowa

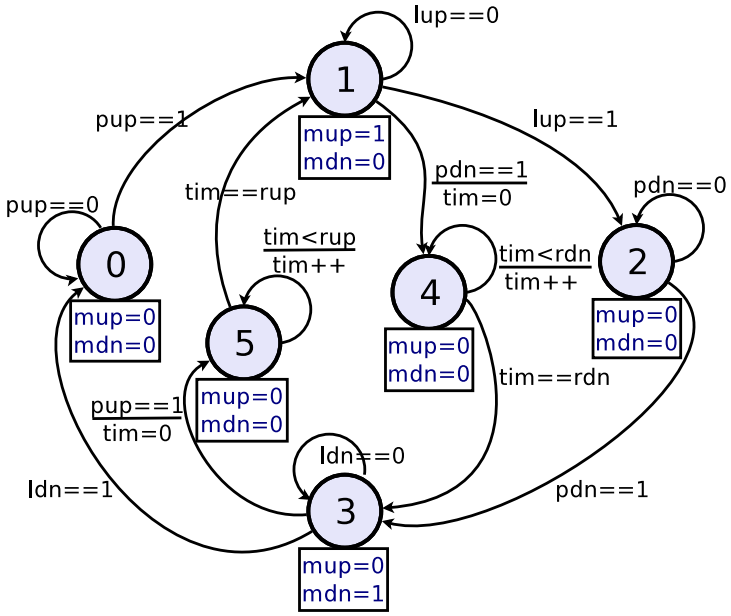
Winda towarowa może przemieszczać się w górę lub w dół (rys. 9). Jeżeli zostaje zawrócona podczas przemieszczania się, najpierw zatrzymuje się na dwie sekundy a następnie zawraca. Sygnały zdefiniowano następująco:

Wejścia:	`pup`, `pdn`	–	przyciski (w górę i w dół)
	`lup`, `ldn`	–	krańcówki (górna i dolna)
Wyjścia:	`mup`, `mdn`	–	silnik



Rys. 9. Winda towarowa

Automat Moorea reprezentujący algorytm sterowania pokazano na rys. 10. Listing 15 zawiera specyfikację w postaci adnotacji do kodu (wszystkie zmienne zadeklarowano globalnie z typem `char`). Zmienne `rdn` oraz `rup` dodatkowo mają modyfikator `const` oraz zostały zainicjowane wartością 10. Kod programu sterującego windą przedstawiono na listingu 16.



Rys. 10. Automat windy towarowej z postojem

Analiza programem Frama C z modułem Jessie generuje 196 lematów poprawności oraz 26 lematów bezpieczeństwa przekroczenia zakresów zmiennych. Większość z nich dowodzi się za pomocą taktyki `intuition`. Pozostałe lematy wymagają jednej z dwóch sekwencji: `intros`, `rewrite _`, `assumption` lub `intros`, `rewrite _`, `rewrite _ in _`, `assumption`. Tylko dwa lematy bezpieczeństwa nie mogą zostać udowodnione z powodu zbyt słabej klauzuli `requires`. Rozwiązaniem jest wzmocnienie jej podobnie jak w poprzednim przykładzie.

Weryfikacja układów sekwencyjnych z ograniczeniami czasowymi za pomocą Frama C i Jessie nie różni się znacząco od omawianych typów układów. Deklaracje zmiennych globalnych mogą uprościć dowody

Listing 15.

```
requires (state>=0 && state<=5) && (mup==0 || mup==1) && (mdn==0 || mup==1) &&
  (lup==0 || lup==1) && (ldn==0 || ldn==1) && (pup==0 || pup==1) &&
  (pdn==0 || pdn==1);
 assigns state,mup,mdn,tim;
 ensures ((\old(state)==0 && pup==1) ==> state==1) &&
  ((\old(state)==1 && pdn==1 && lup==0) ==> ((state==4) && (tim==0))) &&
  ((\old(state)==1 && pdn==0 && lup==1) ==> state==2) &&
  ((\old(state)==2 && pdn==1) ==> state==3) &&
  ((\old(state)==3 && pdn==1 && ldn==0) ==> ((state==5) && (tim==0))) &&
  ((\old(state)==3 && ldn==1 && pup==0) ==> state==0) &&
  ((\old(state)==4 && (\old(tim)<rdn)) ==> ((state==4) && (tim==\old(tim)+1))) &&
  ((\old(state)==4 && (\old(tim)==rdn)) ==> state==3) &&
  ((\old(state)==5 && (\old(tim)<rup)) ==> ((state==5) && (tim==\old(tim)+1))) &&
  ((\old(state)==5 && (\old(tim)==rup)) ==> (state==1));
```

Listing 16.

```
void liftcontrol()
{
  switch(state)
  {
    case 0:
        mup = 0;      mdn = 0;
        if(pup == 1) state = 1;
        break;
    case 1:
        mup = 1;      mdn = 0;
        if(pdn == 1) { state = 4; tim = 0; } else
        if(lup == 1) { state = 2; }
        break;
    case 2:
        mup = 0;      mdm = 0;
        if(pdn == 1) { state = 3; }
        break;
    case 3:
        mup = 0;      mdn = 1;
        if(pup == 1) { state = 5; tim = 0; } else
        if(ldn == 1) { state = 0; }
        break;
    case 4:
        mup = 0;      mdn = 0;
        if(tim < rdn) { tim++; } else
        state = 3;
        break;
    case 5:
        mup = 0;      mdn = 0;
        if(tim < rup) { tim++; } else
        { state = 1; }
        break;
  }
}
```

większości lematów do jednej taktyki `intuition`. Ponadto mogą uwolnić od arytmetyki wskaźników. Jeżeli klauzula `requires` nie zawiera ograniczeń na wartości timerów, to bezpieczeństwo nieprzekroczenia zakresów nie może być udowodnione.

5 Podsumowanie

Przedstawiono proces weryfikacji dla prostych programów sterowania napisanych w ANSI C. Programy realizują zadania układów kombinacyjnych, sekwencyjnych oraz sekwencyjnych z uzależnieniami czasowymi. Specyfikacja układów może być podana w postaci wzoru, słownie, jako automat lub przebiegi czasowe. Implementacja jest sprawdzana pod kątem zgodności ze specyfikacją poprzez ogólnodostępne narzędzia takie jak Frama C, Jessie i Coq. Oprócz poprawności narzędzia te również sprawdzają możliwość przepełniena wartości zmiennych.

Proces weryfikacji daje formalne uzasadnienie poprawności programu, lecz jest procesem skomplikowanym. Aby uczynić go bardziej praktycznym przyszłe prace będą dotyczyły weryfikacji dynamicznej, gdzie w trakcie działania programu będzą sprawdzane wartości pośrednie.

Literatura

[1] Affeldt R., Kobayashi N.: *Formalization and Verification of a Mail Server in Coq*. Lecture Notes in Computer Science, v. 2609, Springer, 2003.

[2] Baudin P., Cuoq P., Filliâtre J.Ch., Marché C., Monate B., Moy Y., Prevosto V. *ACSL: ANSI/ISO C Specification Language*. INRIA 2009.

[3] Coq homepage [online] `http://coq.inria.fr`.

[4] Dijkstra E.W. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs NJ, 1976.

[5] Frama C homepage [online] `http://frama-c.ceq.fr`.

[6] Kalisz J.: *Podstawy elektroniki cyfrowej*. WKŁ Warszawa 1993.

[7] Kerbœuf M., Novak D., Talpin J. P.: *Specification and Verification of a Steam-Boiler with Signal-Coq*, University of Oxford, 2000.

[8] *Sterowniki mikroprocesorowe*. Z. Świder(red.). Oficyna Wydawnicza Politechniki Rzeszowskiej 2002.

[9] Moy Y., Marché C. *Jessie Plugin Tutorial* [online] `http://why.lri.fr`.

[10] Sadolewski J: Wprowadzenie do weryfikacji prostych programów w języku ST za pomocą narzędzi Coq, Why i Caduceus. *Metody Informatyki Stosowanej*. Nr 2 (19) 2009.