STUDIA INFORMATICA 2011
Volume 32 Number 3B (99)

Jan SADOLEWSKI, Zbigniew ŚWIDER
Rzeszów University of Technology, Department of Computer and Control Engineering

SPECIFICATION AND VERIFICATION OF SIMPLE LOGIC CONTROL PROGRAMS USING FRAMA C[1]

Summary. The paper presents an approach to verification process for programs of simple logic controls written in ANSI C. The software is verified with open source tools like Frama C, Jessie and Coq. Process of specification determination and verification whether implementation conforms with specification is demonstrated by several examples, involving combinatorial logic, sequential logic and sequential logic with time constraints.

Keywords: software verification, Frama C, control programs

SPECYFIKACJA I WERYFIKACJA PROSTYCH PROGRAMÓW STEROWANIA LOGICZNEGO Z WYKORZYSTANIEM FRAMA C

Streszczenie. W artykule zaproponowano proces weryfikacji prostych programów sterowania logicznego napisanych w ANSI C. Programy są weryfikowane przez ogólnodostępne narzędzia, jak Frama C, Jessie i Coq. Proces określania specyfikacji i weryfikacji zgodności specyfikacji z implementacją przedstawiono na kilku przykładach układów kombinacyjnych, sekwencyjnych oraz sekwencyjnych z uzależnieniami czasowymi.

Słowa kluczowe: weryfikacja programów, Frama C, systemy sterowania

1. Introduction

Control systems must be reliable and equipped with correct software. Small embedded systems are usually programmed in ANSI C. Despite that the programs are usually written by

[1] This research has been partially supported by MNiSzW under the grant N N516 415638 (2010-2011), and partially by the European Union under the European Social Fund.

experienced programmers, final code may contain mistakes and side effects. Software tests can reveal most of mistakes, but never give assurance that the code is completely error free. Formal verification of compliance between specification and implementation can help to find mistakes and side effects.

The paper presents verification process for simple control programs and function blocks written in ANSI C. It employs open source tools for code analysis like Frama C [5] with Jessie plug-in [9] and Coq [3]. Frama C supports C code analysis with specification annotations written in ACSL language [2]. It contains an internal module for static value analysis of the whole program (or its main part), but it is too weak to prove correctness even simple programs. That weakness motivates the use of Jessie plug-in, which generates condition lemmas based on Dijkstra preconditions [4]. The lemmas consists of ensures proof obligations and safety proof obligations. Ensures proof obligations guarantee program correctness, while safety proof obligations assure that variable overflow does not occur in runtime. The lemmas can be proved half-automatically with Coq.

The research is an extension of previous work [10], by replacing somewhat obsolete Caduceus with Frama C. It is also related to [1, 7] but uses Coq as a backend prover instead of modeling tool. The paper is divided into three parts each of which focuses on examples, i.e. combinatorial logic – binary multiplexer and temperature control, sequential logic – RS flip flop and detection of truck movement, and sequential logic with time constraints – blinking LED and cargo lift. The examples demonstrate, that even for simple programs serious effort is required to prove correctness formally.

2. Combinatorial logic

Specification of combinatorial logic can be provided by output formulas, a text, and value tables. Examples presented here use the first and the second form. The third one can be converted to minimal form using Karnaugh maps [6].

2.1. Binary multiplexer

The first example involves a binary multiplexer of Fig. 1 with three inputs and one output. The `s` input toggles the output `y` between `x0` and `x1` inputs. Specification formula is shown in (1), followed by C program listing 1 (function). The specification written as code annotation for Frama C is in listing 2.

$$y = (s \wedge x0) \vee (!s \wedge x1) \tag{1}$$

```
// ----- 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));
```

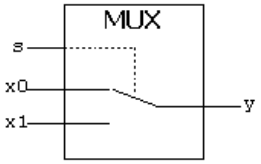

Fig. 1. Binary multiplexer
Rys. 1. Multiplekser binarny

The specification involves two clauses `requires` and `ensures`. The first one expresses constraints on input values at function call (preconditions). The other expresses constraint which must be satisfied upon the function return (postconditions). Variable `\result` represents value returned by the function, i.e. multiplexer output. The formula (1) is inserted into the `ensures` clause for implementation.

By using Jessie plug-in available in Frama C, one can check whether program satisfies postconditions when preconditions are met. Jessie module generates a file with obligation lemmas, which can be proved using Coq. For the multiplexer, four ensures proof obligations and two safety proof obligations are obtained.

a)
```
s: int8
H4: integer_of_int8
s = 0 -> False
-----------------
False
```

b)
```
x0: int8
y: int8
HW_6: y = x0
-----------------
integer_of_int8 x0 = 0
```

c)
```
y: int8
H2: integer_of_int8 y
= 0
-----------------
integer_of_int8 y = 0
```

Fig. 2. Examples of goals and contexts
Rys. 2. Przykłady celów oraz kontekstów

All Coq proofs implemented here begin with `intuition` tactic, which splits given lemma into simpler form. The splitting generates additional hypotheses collected into a space called context. In the next step, the user has to choose which hypotheses to apply for further proof using either `apply _`, `rewrite _` or `assumption` tactic. The _ character (underscore) must be replaced by the user with one of the hypothesis from the context. `Apply` is used when conclusion of the hypothesis is equal to the current goal (Fig. 2a), `rewrite` when hypothesis and the goal are both equalities (Fig. 2b), and `assumption` when the whole goal is equal to hypothesis (Fig. 2c). `Assumption` is a part of `intuition`, so the use of `assumption` immediately after `intuition` is incorrect. The presented tactics prove all lemmas belonging to the ensures

proof obligations group. Safety proof obligations are proved instantly after the initial `intuition` tactic.

2.2. Heater control

Temperature is monitored by a thermometer with contacts `a`, `b`, `c`, `d` (Fig. 3a). Heaters `H1`, `H2` are turned on by the relays `r1`, `r2`, `r3` (Fig. 3b) according to the following rules:

1. $t < ta$ – `H1`, `H2` in parallel,
2. $ta \le t < tb$ – `H1` only,
3. $tb \le t < tc$ – `H2` only,
4. $tc \le t < td$ – `H1` and `H2` in series,
5. $t \ge td$ – both switched off.

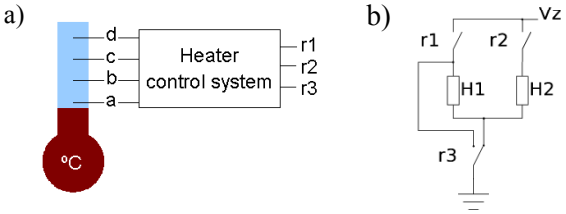

Fig. 3. Heater control system
Rys. 3. System sterowania grzejnikami

Inputs				Outputs		
a	b	c	d	r1	r2	r3
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	-

Fig. 4. Input and output table
Rys. 4. Tablica wejść i wyjść

$$r1 = !b \qquad r2 = !a \vee (b \wedge !d) \qquad r3 = c \tag{2}$$

Control program can be written using either truth table (Fig. 4) or explicit formulae (2) derived by Karnaugh maps. Here the truth table is used for implementation (listing 3) and (2) for specification (listing 4). The `requires` clause specifies admissible sets of signal values (none of the contacts faulty). `Assigns` indicates global variables modified by function body (necessary for Jessie), `ensures` contains formulas (2) with the last one written as implication. It is necessary because Karnaugh map leaves some undetermined values.

```
// ----- Listing  3 -----
void termometr()
{
   if(!a) {          r1 = 1;  r2 = 1;  r3 = 0;  }
   else {
      if(!b) {       r1 = 1;  r2 = 0;  r3 = 0;  }
      else {
```

```
            if(!c) {    r1 = 0;  r2 = 1;  r3 = 0;  }
            else {
               if(!d) { r1 = 0;  r2 = 1;  r3 = 1;  }
               else {   r1 = 0;  r2 = 0;  r3 = 0;  }
                 }
            }
         }
}

// ----- Listing  4 -----
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 r1, r2, r3;
ensures (!b <==> r1) && ((!a || (b && !d)) <==> r2) && (r3 ==> c);
```

Here, the file generated by Jessie module contains twenty two (22) ensures proof obligation lemmas. No safety proof obligations are generated because all assignments in the code are constant. Lemmas can be proved by one of the following tactics:

1. `intuition`,
2. Sequence of `intros`, `rewrite _ in _`, `contradiction`,
3. Sequence of `intros`, `rewrite _`, `omega`.

In some cases 2nd and 3rd tactic would require additional `left`, `right` and `decompose [and] in _`. The first two are used when goal is disjunction and `decompose` when context hypothesis is conjunction.

Tactics presented here can be generalized for other combinatorial examples.

3. Sequential logic

In sequential logic the output depends on current input and on internal state from previous calculation cycle. Implementations as C functions require additional parameters, usually stored as pointers. Dereferencing those pointers at the beginning means retrieving the previous state. Specification can be in text form, formula form, or created from signal-time plots and automaton graphs.

3.1. RS flip flop

Suppose the RS flip flop (Fig. 5) is specified by the following text: *1 (logic) at input* `R` *resets output* `Q` *into 0, 1 at input* `S` *sets* `Q` *into 1 but only when* `R` *is 0, if* `R` *and* `S` *are both 0 then value of* `Q` *is preserved (state), if* `R` *and* `S` *are 1 then* `R` *is predominant and* `Q` *is reset to 0*. Such specification is written as code annotation in listing 5.

```
// ----- Listing  5 -----
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));
```

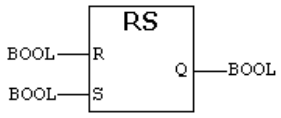

Fig. 5. RS flip flop symbol
Rys. 5. Symbol przerzutnika RS

The `requires` clause contains `\valid` modifier which indicates that the `state` variable will not be `NULL` at the function call. The `assigns` clause must contain `state` variable since it is modified by the function (even when it is pointer dereferencing). The modifier `\old` in the `ensures` clause indicates value from previous execution (previous state).

```
// ----- Listing  6 -----
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;
  }
}
```

Listing 6 shows implementation of the RS flip flop function as an automaton. Jessie analysis generates 46 proof obligation lemmas and 9 safety proof obligations. Most of those 46 lemmas can be proved similarly as before. The other lemmas require operations on pointer arithmetic, which involve `subst _`, `rewrite select_store_eq`, `rewrite pset_singleton in _` tactics, and some combinations of the tactics mentioned before. Moreover, some lemmas involve contradictory hypotheses, so additional tactics like `absurd _`, `omega` and `contradiction` are needed. The safety proof obligations are simpler, and 7 of them are proved with `intuition`. Remaining 2 lemmas can be proved with the sequence of `intros`, `rewrite _` and `omega`.

3.2. Detecting truck movement

Direction of movement of a truck, to the left or to the right, is detected by two photo sensors `S1`, `S2` and signaled by the lamps `L1`, `L2` (Fig. 6). If the truck moves to the left, `L1` is on. Distance between the sensors is smaller than length of the truck. One assumes that the truck can stop only outside the sensors.

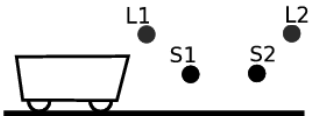

Fig. 6. Detecting direction of truck movement
Rys. 6. Wykrywanie kierunku ruchu wózka

Specification can be represented by Moore automaton shown in Fig. 7, and coded accordingly as in listing 7. As before, to simplify proof obligations, it is assumed that all variables are declared globally as `char` type. Listing 8 involves implementation of the control program.

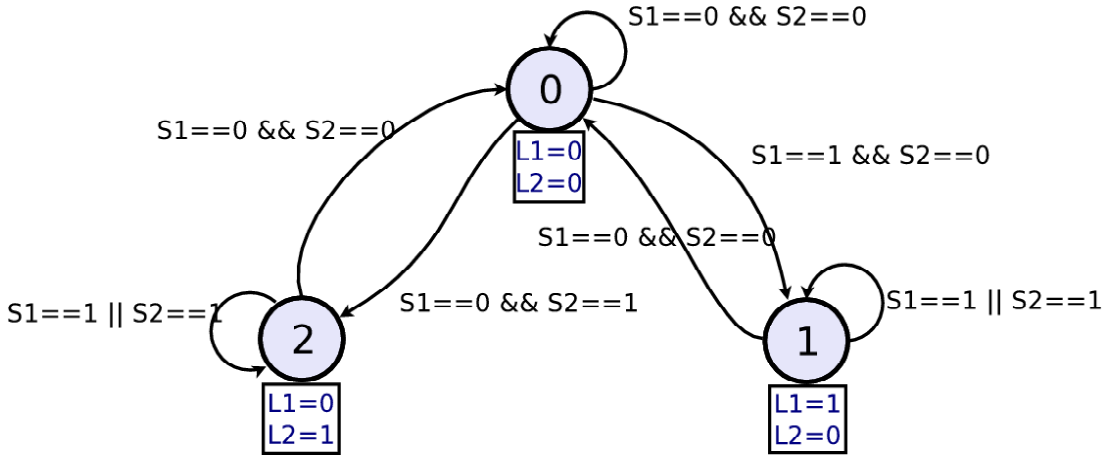

Fig. 7. Automaton detecting direction of truck movement
Rys. 7. Automat wykrywający kierunek ruchu wózka

Jessie analysis produces 97 proof correctness obligation lemmas, and 2 safety proof obligations. Due to the use of global declaration, 89 lemmas are proved immediately by `intuition` tactic. Additional `rewrite _` and `assumption` tactics prove remaining 8 lemmas. Two safety lemmas are proved by `intuition`.

```
// ----- Listing  7 -----
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));

// ----- Listing  8 -----
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;
    }
}
```

Verification of sequential logic with Frama C and Jessie does not differ much from combinatorial one. The only difference is the use of pointers in function declarations. It leads to somewhat complicated verification lemmas, which require knowledge on proving Jessie pointer arithmetics with Coq.

4. Sequential logic with time constraints

Software implementations of sequential logic with time constraints involve time span called *cycle time*. Measured time is a multiplication of the cycle time. Specifications expressed by automata do not differ much from previous examples, except that formulae describing states and transitions are extended by time.

4.1. Flashing LED

Led `L` flashes when key `K` is pressed (Fig. 8). `Th`, `Tl` denote time periods when the LED is on or off, respectively. LED is turned off immediately when the key is released. The automaton in Fig. 9 is derived directly from the time plot. Specification based on it is shown in listing 9. Expressions describing states and transitions involve condition written above the separating line and time operation (if needed) below the line. As seen from Fig. 9, the condition `K==0` describes each of the two incoming edges into `0` state. This allows to write only one part `ensures` clause for (`K==0`). Other parts of the clause involve time operations.

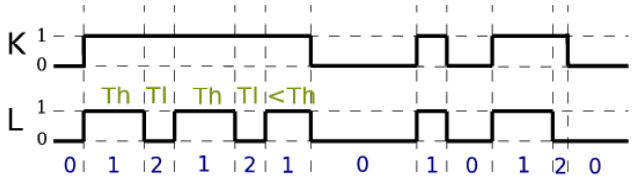

Fig. 8. Time-signal plots of flashing LED
Rys. 8. Przebiegi czasowe sygnałów dla błyskającego LEDa

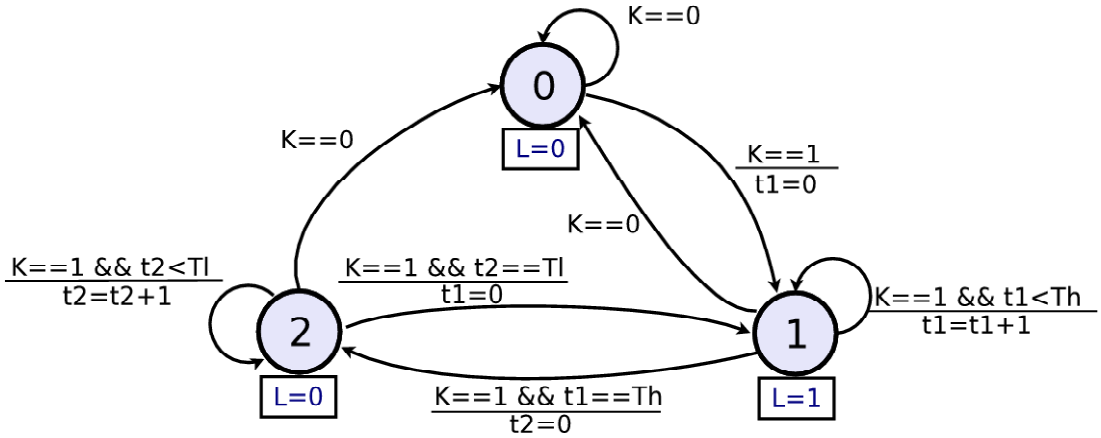

Fig. 9. State automaton for flashing LED
Rys. 9. Automat stanów dla błyskającego LEDa

```
// ----- Listing  9 -----
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));
```

Software implementation is presented on listing 10 (variables are `char` globals). `T1` and `T2` are assumed to have `const` modifier initialized with some values (e.g. `6` and `3`, not shown in the listing). Those values represent numbers of program execution cycles for elapsing of required times.

```
// ----- Listing 10 -----
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;
  }
}
```

Jessie analysis produces 209 lemmas to prove program correctness, and 28 lemmas for variable overflow safety. Most of the correctness lemmas are verified by `intuition` tactic.

Remaining ones can be proved similarly as in the multiplexer and truck movement examples. Safety lemmas are however not so trivial. Four of them cannot be proved because of weakness of `requires` clause in the current form. Main problem is that `Th` and `Tl` are treated by Jessie not as constants but as variables. Besides, there are no assumptions that `t1 <= Th` and `t2 <= Tl` nor `t1 <= 126` and `t2 <= 126`, which guaranties overflow safety on `char` type for that program. Strengthening `requires` clause with these assumptions allows to prove that overflow does not occur.

4.2. Cargo lift

The lift can move up or down as shown in Fig. 10. If it is turned back while moving, it stops first for a moment, e.g. 2 seconds, and then returns. Signals are as follows:

Inputs: `pup`, `pdn` – pushbuttons (up or down)
`lup`, `ldn` – limit switches
Outputs: `mup`, `mdn` – motor
Times: `rup`, `rdn` – reverse stop

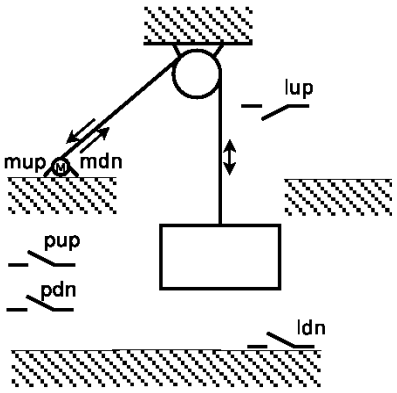

Fig. 10. Cargo lift
Rys. 10. Winda towarowa

Automaton representing the algorithm is shown in Fig. 11. Listing 11 involves corresponding specification in code annotation form (variables are `char` globals). Variables `rdn` and `rup` are declared with `const` modifier and initialized with values `10`. Listing 12 presents the lift program.

```
// ----- Listing 11 -----
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));
```

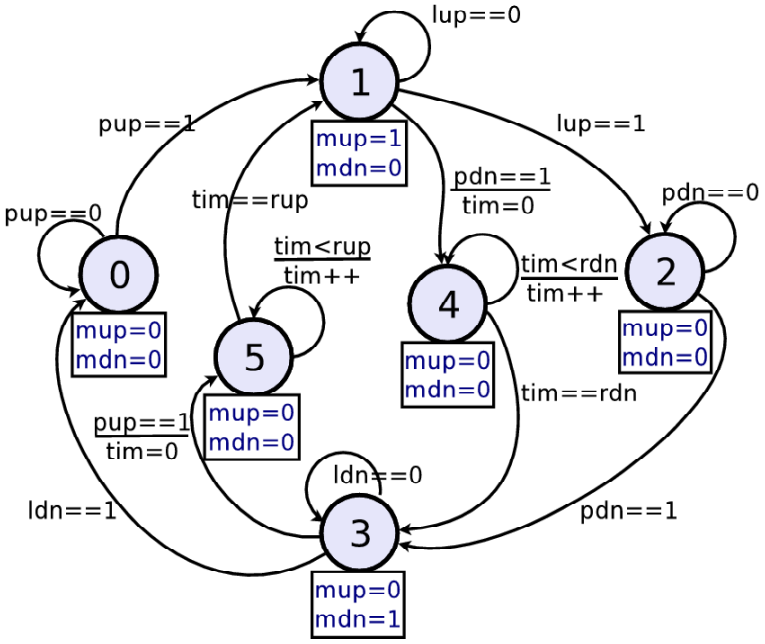

Fig. 11. Automaton of lift control with time stop
Rys. 11. Automat windy towarowej z postojem

```
// ----- Listing 12 -----
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;
  }
}
```

Jessie analysis generates 196 proof correctness obligation lemmas and 26 safety proof obligations. As before most of correctness lemmas can be proved by `intuition`. The remaining lemmas require sequence of `intros`, `rewrite _`, `assumption` or `intros`, `rewrite _`, `rewrite _ in _`, `assumption`. Only two safety lemmas cannot be proved because of `requires` clause weakness, because the variable `tim` can exceed its maximum value. By strengthening `requires` with `tim>=0 && tim <= 126` expression, those lemmas can be proved with `omega` tactic.

Verification of sequential logic with time constraints does not differ from earlier cases. Declaration of global variable simplifies proofs of most lemmas to `intuition` tactic and frees them from pointer arithmetic. However, if no constraints are given for time variables, lack of overflow cannot be proved.

5. Summary

Verification approach for simple control programs written in ANSI C has been presented. The programs involve combinatorial, sequential and sequential-plus-time constraints examples. Specification can be given in the form of formula, verbally, as graph automaton or time plot. Implementation is checked with respect to verification using Frama C, Jessie and Coq open-source tools. Besides correctness, variable overflow safety lemmas are also evaluated.

The verification provides formal justification of program correctness but is really quite cumbersome. Therefore to make it more practical, future work will focus on so-called dynamic verification, similar somewhat to debugging, where intermediate results are checked symbolically “on-line”.

BIBLIOGRAPHY

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 2007.
7. Kerbœuf M., Novak D., Talpin J. P.: Specification and Verification of a Steam-Boiler with Signal-Coq, University of Oxford, 2000.
8. Świder Z. (red.): Sterowniki mikroprocesorowe. Oficyna Wydawnicza Politechniki Rzeszowskiej, 2002.
9. Moy Y., Marché C.: Jessie Plugin Tutorial. [online] http://why.lri.fr.
10. Sadolewski J.: Introduction to verification simple programs in ST language with Coq, Why and Caduceus. Metody Informatyki Stosowanej. No. 2 (19) 2009 (in Polish).

Recenzent: Prof. dr hab. inż. Bolesław Pochopień

Wpłynęło do Redakcji 7 lutego 2011 r.

Omówienie

Praca przedstawia proces weryfikacji prostych programów sterowania zapisanych w języku ANSI C z wykorzystaniem ogólnodostępnych narzędzi Frama C z modułem Jessie oraz Coq. Weryfikacja polega na wykazaniu zgodności specyfikacji z implementacją, a także na wykazaniu braku sytuacji, w których nastąpiłoby przepełnienie wartości zmiennych. Specyfikacja może być podana wzorami, np. (1) i (2), słownie, w postaci automatu (rys. 4, 6, 7) lub jako wykres czasowy (rys. 5). Każda z tych form jest zapisywana w postaci adnotacji do kodu, a następnie poddawana analizie programem Frama C z modułem Jessie, które badają zgodność specyfikacji z utworzonym kodem. W wyniku analizy otrzymuje się lematy dowodzące poprawności (*ensures proof obligations*) oraz lematy bezpieczeństwa zakresów zmiennych (*safety proof obligations*). Dowody tych lematów przeprowadza się półautomatycznie programem Coq.

Proces tworzenia dowodów pokazano na przykładach układów kombinacyjnych, sekwencyjnych oraz sekwencyjnych z uzależnieniami czasowymi. Zwrócono uwagę na przypadki, gdy kod ma umożliwiać deklarację kolejnych instancji. Przykłady niewymagające wieloinstancyjności przedstawiono z wykorzystaniem zmiennych globalnych, które eliminując arytmetykę wskaźników upraszczają dowody. Niemożliwość udowodnienia jednego z lematów blokuje wykazanie poprawności lub bezpieczeństwa weryfikowanej jednostki. Potencjalnymi przyczynami mogą być błąd w implementacji, błąd w specyfikacji lub zbyt słabe warunki wstępne. W dwóch przypadkach pokazano, jak należy je wzmocnić.

Addresses

Jan SADOLEWSKI: Rzeszow University of Technology, Department of Computer and Control Engineering, ul. W. Pola 2, 35-959 Rzeszów, Poland, js@prz-rzeszow.pl.

Zbigniew ŚWIDER: Rzeszow University of Technology, Department of Computer and Control Engineering, ul. W. Pola 2, 35-959 Rzeszów, Poland, swiderzb@prz-rzeszow.pl.