Proving Reflex Program Verification Conditions in Coq Proof Assistant

Ivan Chernenko
Institute of Automation and Electrometry, SB RAS
Novosibirsk, Russia

Igor Anureev
Institute of Automation and Electrometry, SB RAS
Novosibirsk, Russia

Natalia Garanina
Institute of Automation and Electrometry, SB RAS
Novosibirsk, Russia

***Abstract*— The process-oriented paradigm is a promising approach to the development of control software based on the natural concept of the process. Many safety-critical systems uses control software. This is a reason for formal verification such systems. Deductive verification is the formal methods of proving the program correctness (the satisfiability program requirements). Requirements are formalized as annotations added to programs. The resulting annotated programs are reduced to verification conditions – formulas in some logical language. The original program is considered to be correct if all the verification conditions are true. This paper presents the results of experiments on proving verification conditions in Coq proof assistant within the framework of the two-step method of deductive verification of process-oriented programs in Reflex language.**

Keywords*— *process-oriented programming, Reflex language, deductive verification, requirements, annotations, verification conditions, temporal properties, control software

I. Introduction

The formal basis of process-oriented programs is the hyperprocess model [1] – a special type of finite state automata, in which states are distributed into classes called processes. Process states are defined as sequences of actions, including actions to change the states of other processes and actions with timeouts. Thus, process-oriented programs are defined as sets of communicating processes.

The process-oriented program language Reflex [2] is a domain-specific language that describes a control system as a set of interacting processes at the top level, while maintaining the familiar C syntax at the bottom level. This made it possible to successfully apply it in a number of industrial applications, e. g. control software for a silicon single crystal growth furnace, a big solar vacuum telescope and precision angle measuring machine which are examples of safety-critical systems [3, 4]. Such systems require applying formal verification methods, in particular, deductive verification [5] as rather powerful approach.

We developed the two-step method of deductive program verification for Reflex programs [6]. In the first step, the annotated Reflex program is translated into a limited subset of annotated C programs, and in the second step, verification conditions are generated for programs from this subset.

The main difficulty of deductive verification for program correctness is the proof of generated verification conditions. In [6], the two-step method was applied to a hand dryer control program and two requirements for this program were verified using the Z3 SMT solver [7]. The experiments showed that this solver could not handle requirements which verification conditions contain quantifiers and require inductive proof. In this paper, we used the same example to show that the Coq proof assistant [8] successfully verified 5 requirements for this hand dryer control program due to ability to prove higher-order logic formulas and use inductive proof schemes.

In the rest of the paper, we provide the code of the Reflex hand dryer control program (Section II), the requirements for this program (Section III), annotations, that are a formal description of these requirements (Section IV), the result of the transformation of the hand dryer control program into the C program (Section V), the generated verification conditions (Section VI) and describe the proving verification conditions in the Coq proof assistant with the standard Coq library (Section VII).

II. Reflex Program

We consider the program *HandDryerController*:

```
PROGR HandDryerController {
 tact 100;
 const ON true;
 const OFF false;
 proc Ctrl {
    bool hands;
    bool dryer;
    state waiting {
      if ( hands == ON ) {
            dryer = ON;
            set next;
         }
      else
            dryer = OFF;
    }
    state drying {
    if ( hands == ON ) reset timeout;
    timeout 10 set state waiting;
  }
 }
}
```

Single process *Ctrl* is defined in this program. This process has two states *waiting* and *drying*. Two logical variables are declared: *hands* and *dryer*. The variable *hands* shows whether hands are detected under the fan heater. Its value is changed by the environment. The *dryer* variable describes whether the fan heater is turned on. In the *waiting* state, the process checks if there are hands. If hands are detected, the fan heater turns on and the process goes to the *drying* state; if there are no hands, the fan heater turns off. In the *drying* state, the process also checks if there are hands. If hands are detected, the timeout is reset. The program *tact* is

978-0-7381-4389-7/21/$31.00 ©2021 IEEE

the time of the execution one iteration of the control loop. In this program, it takes 100 milliseconds. Then a timeout is 10 tacts, regardless of whether there were hands or not.

III. Requirements for the Program

We verify the following 5 requirements for the hand dryer control program *HandDryerController*:

1. The fan heater should turn on in a reasonable time (e.g. 0.2 seconds) after detecting hands.
2. The fan heater never turns on spontaneously (If there are no hands and the fan heater is not turned on, it will not turn on until the hands appear).
3. If the hands are removed, the fan heater will turn off after no more than 1 second, if the hands do not reappear during this time.
4. If there are hands and the fan heater is turned on, it will not turn off.
5. The time of uninterrupted functioning of the fan heater is not more than an hour.

Note that the first four requirements are satisfied by the program, and the fifth is not. Non-satisfiability of the fifth requirements makes us to try the Coq on a negative example.

IV. Formal Annotations

In the two-step method [6], the requirements are formalized as three annotations, which are formulas of the many-sorted predicate logic. The first annotation gives restrictions on the initial values of program variables at the beginning of the first loop of the control system. The second annotation formulates restrictions on changing the input variables of the program by the environment (in particular, by the controlled object). The third annotation defines a condition that is true at each entering the control loop, before the input variables are changed by the environment (in particular, the control object). It is called the *invariant* of the control loop. In our case, only the loop invariant is used, because there are no restrictions on the single input variable *hands* (hands can appear at any time), and the initial values are explicitly assigned during the program initialization. For each requirement of number *i*, the control loop invariant *inv* is the conjunction *inv_i & & extraInv*. The first conjunct is a formal description of the corresponding requirement, and the second describes invariant, the same for all requirements. For example, for the first requirement, the conjunct *inv_1* is: *forall i (0<i && i<=timer-1 && hands[i-1] = OFF && hands[i] = ON => exists j (i <= j && j <= i+1 && (forall k (i <= k && k < j => dryer[k] = OFF && hands[k+1] = ON)) && dryer[j] = ON)).*

V. Result of the Transformation of the Reflex Program into the C Program

The annotated *HandDryerController* program after transformation to the C program looks as follows:

```
# define tact 100
# define ON true
# define OFF false
# define stopState 0
# define errorState 1
# define ctrlWaiting 2
# define ctrlDrying 3
int timer;
int ctrlState[];
int ctrlTimer;
bool hands[];
bool dryer[];
inline void init () {
 cycleNumber = 0;
 timer = 0;
 ctrlTimer = 0;
 ctrlState[0] = ctrlWaiting;
 hands[0] = OFF;
 dryer[0] = OFF;
}
inline void ctrl_exec () {
 switch(ctrlState[timer]) {
  case ctrlWaiting:
   if (hands[timer] == ON) {
    dryer[timer] = ON;
    ctrlTimer = 0;
    ctrlState[timer] = ctrlDrying;
   }
   else dryer[timer] = OFF;
   break;
  case ctrlDrying:
   if (hands[timer] == ON) {
    ctrlTimer = 0;
    ctrlState[timer] = ctrlDrying;
   }
   If (ctrlTimer >= 10) {
    ctrlTimer = 0;
    ctrlState[timer] = ctrlWaiting;
   }
   break;
  default: unreachable;
 }
}
void main () {
 init();
 for (;;) {
  invariant inv(hands, dryer,
         ctrlState, ctrlTimer, timer);
  timer = timer + 1;
  ctrlTimer = ctrlTimer + 1;
  ctrlState[timer]=ctrlState[timer-1];
  dryer[timer] = dryer[timer - 1];
  havoc hands[timer];
  ctrl_exec();
 }
}
```

The result has the syntax and semantics of a C program, except for two things. First, arrays are considered infinite and dynamic, i.e. extensible when assignment is performed. Second, the new `havoc a[i]` instruction is used. It means assigning an arbitrary value to an array `a` at index `i`. This instruction simulates how the environment updates the variables' values of the Reflex program.

In the C program, the macro definition *tact* corresponds to the *tact* construction in the Reflex program. The Reflex program constants are replaced with the corresponding macro definitions. Other macro definitions encode process states. The macro definitions *stopState* and *errorState* encode the *stop* and *error* states, *ctrlWaiting* and *ctrlDrying* encode the *waiting* and *drying* states, respectively.

The variable *timer* corresponds to the global timer. The *ctrlState* array specifies the states of the process *Ctrl*, and the variable *ctrlTimer* specifies the local timer. Next, the arrays *hands* and *dryer* are declared, corresponding to the variables in the Reflex program.

The function *init* initializes the process. It sets the value 0 for the global timer and the local timer of process *Ctrl*, sets the process *Ctrl* to the initial state *waiting*, and assigns the initial values OFF to the Reflex variables *hands* and *dryer*.

The function *ctrl_exec* defines the actions of the process *Ctrl* in the control loop. It is the switch statement containing two labels *ctrlWaiting* and *ctrlDrying* macro constants corresponding to all the states (*waiting* and *drying*) defined in the Reflex program for the process *Ctrl*. Instructions mapped to labels correspond to the Reflex program statements in the states that are encoded by these labels. The *ctrlWaiting* label is matched with the operator if (*hands[timer]==ON*) (the current value of the variable hands is checked). If *hands* is set to ON, the current value of the variable *dryer* is set to ON, the local timer of the process *Ctrl* is reset, and the process enters the *drying* state. If the current value of *hands* is not ON, the current value of *dryer* is set to OFF. The *ctrlDrying* label is matched with the sequential execution of two conditional statements. First, the current value of *hands* is checked, and if it is ON, the local timer is reset and the process is set to the *drying* state, which corresponds to the reset of the timeout. Then it checks the value of the local timer, and if *ctrlTimer>=10*, the local timer is reset and the process enters the *waiting* state, which corresponds to the timeout being triggered.

In the function *main* corresponding to the program execution, initialization is performed first, and then an infinite loop is executed. The loop body begins with the *invariant annotator*, which specifies that the loop invariant should be true at this point in the program. Next, the values of the global timer and the local timer of the process *Ctrl* are increased. Then the current state and the value of the variable *dryer* are set to the previous ones (the states of the processes and the values of the variables do not change after the global timer is incremented), the input variable *hands* is assigned an arbitrary value and the process *Ctrl* is executed.

In the next section, we describe the verification conditions generated for this C program.

VI. Verification Conditions

The algorithm for generating verification conditions in the two-step method is based on the computation of the strongest postcondition [9]. It generates the representation of the symbolic execution of the program along all paths. There are 8 paths in the *HandDryerController* program. Thus, 8 verification conditions are generated for each requirement. The total number of verification conditions for all 5 requirements is 40. For example, the verification condition for the path from the loop invariant to the loop invariant (one iteration of the control loop) in the case when the process *Ctrl* of program *HandDryerController* at time *timer* is in the *waiting* state (*ctrlState[timer] = ctrlWaiting*) and hands appear (*hands[timer] == ON*), has the form:

inv(hands0, dryer0, ctrlState0, ctrlTimer0, timer0) &&
ctrlState0[timer0] = ctrlWaiting &&
hands0[timer0] == ON &&
dryer1 = upd(dryer0, timer0, ON) && ctrlTimer1 = 0 &&
ctrlState1 = upd(ctrlState0, timer0, ctrlDrying) &&
timer1 = timer0 + 1 && ctrlTimer2 = ctrlTimer1 + 1 &&
ctrlState2=upd(ctrlState1, timer1, ctrlState1[timer1-1]) &&
dryer2 = upd(dryer1, timer1, dryer0[timer1-1]) &&
hands1 = upd(hands0, timer1, logvar2) =>
inv(hands1, dryer2, ctrlState2, ctrlTimer2, timer1).

VII. Proving Verification Conditions in Coq Proof Assistant

For proving the verification conditions in Coq, we define the following theory describing their content. The basic theory contains the definitions of the constants *ON*, *OFF*, *stopState*, *errorState*, *ctrlWaiting* and *ctrlDrying*, the declarations for the variables *hands0*, *hands1*, *dryer0*, *dryer1*, *ctrlState0*, *ctrlState1*, *ctrlTimer0*, *ctrlTimer1*, *timer0*, *timer1*, *dryer2*, *ctrlState2*, *ctrlTimer2* used in the verification conditions, and axioms postulating that the arrays *hands0*, *dryer0*, *ctrlState0* are infinite. In [10], we give these formal annotations and verification conditions in the language of the Coq proof assistant. The verification conditions are formulated as theorems. Let us give theorem corresponding to the verification condition presented in section VI:

Theorem *proof1_2:*
(startnewloop hands0 hands1 dryer0 dryer1 ctrlState0
ctrlState1 ctrlTimer0 ctrlTimer1 timer0 timer1) /\ cond2 ->
(inv hands1 dryer2 ctrlState2 ctrlTimer2 timer1).,
where *cond2* is defined as follows:

Definition *cond2 :=*
ctrlState1.[of_Z timer1] = ctrlWaiting /
hands1.[of_Z timer1] = ON /
dryer2 = dryer1.[of_Z timer1 <- ON] /\ ctrlTimer2 = 0 /
ctrlState2 = ctrlState1.[of_Z timer1 <- ctrlDrying].

The conjuncts *propInv* in the loop invariant is:

forall *i, (0 < i /\ i <= t ->*
(P hands dryer ctrlState ctrlTimer timer i)).

To prove the requirement, we analyze the cases ($i < t$) and ($i = t$). To do this, we use the tactic *elim* with the lemma *Zle_lt_or_eq* from the standard library *ZArith*. The proofs for the case ($i < t$) follows the invariant for the previous iteration of the loop. They can be done in a similar way for all the requirements and verification conditions. We prove the theorem *startnewloop_to_propInv* for each requirement. This theorem is used for partially automatization of proving the conjunct *propInv* for the case ($i < t$).

Requirements 1, 3, and 5 for the program *HandDryerController* define its temporal properties. Hence, proving by induction is necessary for several verification conditions of these requirements. In particular, induction is used to prove the verification conditions 2, 3, 5, and 7 for the third requirement. We use the following approach to prove by induction. We define the predicate which is proved by induction on the last argument. For the third requirement, it has the form:

Definition *ind_prove_pred (dryer : array bool) (x y l : Z) : Prop :=*
*(***forall** *k, (x <= k /\ k < l -> dryer.[of_Z k] = ON /*
hands1.[of_Z k] = OFF)) ->
exists *j (l<= j /\ j <= y /*
*(**forall** k, (x<= k /\ k < j -> dryer.[of_Z k] = ON /*
hands1.[of_Z k]=OFF)) /

(dryer.[of_Z j] = OFF \/ hands1.[of_Z j] = ON)).,
where *x = i, y = i + (11 - 1).*

It is an implication. For *l = 0*, the right part of this implication matches the goal, and the left part is *true*. The value *l = y* is used as the base of the induction. Step of induction: if the predicate is *true* for *l=y0*, then it will be *true* for *l = (Z. pred y0)* (*Z. pred y0* — the previous number for *y0*). To do this, we proved our inductive scheme based on the *natlike_ind* inductive scheme from the *ZArith* library.

To prove that the fifth requirement is not satisfied, we use the following approach. The values of variables for which the negation of some verification condition is true were determined. We prove that this requirement is not satisfied for paths in the program corresponding to verification conditions 2, 5, 6 and 7.

For arrays in Coq, we use the standard library *Coq.Array.pArray*. It contains functions *make*, *get* and *set* a value by index and computing the length of the array and axioms for these functions. We use the following functions and notations to prove the verification conditions:

- *t.[i]* — the *i*-th element of the array *t*
- *t.[i <- a]* - the array obtained from *t* by replacing the value of the *i*-th element with *a*
- *length t* — the length of the array *t*

When falsifying the fifth requirement, we use the *make* function to construct counterexamples of arrays. The *make* function allows defining arrays with elements equal to the same value. But we also need arrays with element values depend on the indexes. To define such arrays, we create a function that requires a function *f* of *type Z->bool*, an integer *n* of type *Z* and a proof that *0<=n<=max_length* and we define a bool array, such that its length is *n* and the value of the element at index *i* is equal to (*f i*) for all *i*, such that *0<=i<=n*.

For array indexes, Coq uses the type *int*. But formulas contain comparisons and arithmetic addition and subtraction. The standard Coq library defines lemmas for these operations on types, such as *nat* (natural numbers) and *Z* (integers). Since the Coq standard library has the function *of_Z* for translating *Z* to *int*, the *timer* and *ctrlTimer* variables have the type *Z*.

The *not_true_is_false* lemma is also used to prove that the value is *OFF* if it is not *ON*, and the *negb_true_iff*, *negb_false_iff*, *andb_true_iff*, and *andb_false_iff* lemmas are used to prove the properties of the *hands0* array, for which the fifth requirement is not satisfied. These lemmas are defined in the library *Coq.Bool.Bool*.

All 32 verification conditions for the first four requirements have been proven. Hence, we consider the hand dryer program to be correct with respect to these requirements. We also falsify the fifth requirement by proving the negations of its verification conditions. The Coq code of all proofs is given in [10].

VIII. CONCLUSION

In this paper, we present the experiments on proving the verification conditions generated within the framework of the two-step method of deductive verification of process-oriented programs in Coq proof assistant. The experiments have shown that the Coq proof assistant is more powerful for proving the verification conditions of process-oriented programs compared to the previously used SMT solver Z3.

The formulated requirements for the hand dryer control program and the generated verification conditions are typical for a wide class of process-oriented control programs. Hence, we consider the combination of the two-step method of deductive verification for process-oriented programs with the Coq proof assistant can be successfully used in the practical formal verification of such control programs.

In this example, the verification conditions were built manually, and the simplification of this process was achieved only by introducing notation for subformulas. To verify more complex Reflex programs, it is planned to develop a tool generating verification conditions in the Coq format for annotated Reflex programs.

REFERENCES

[1] Zyubin, V.E. “Hyper-automaton: A Model of Control Algorithms,” in Proceedings of the IEEE Intern. Siberian Conf. on Control and Communications (SIBCON-2007), 2007, pp. 51-57.

[2] Liakh T.V., Rozov A.S., and Zyubin V.E., “Reflex Language: a Practical Notation for Cyber-Physical Systems,” J. System Informatics, 12(4), pp. 85-104, 2018.

[3] Pedersen Notander J., Höst M., Runeson P., “Challenges in Flexible Safety-Critical Software Development – An Industrial Qualitative Survey,” Lecture Notes in Computer Science, 7983, pp. 283-297, 2013.

[4] Leveson N., Engineering a safer world: systems thinking applied to safety, MIT Press, Cambridge, 2011.

[5] Hähnle, R. and Huisman, M. “Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools,” Lecture Notes in Computer Science, 10000, pp. 345-373, 2019.

[6] Anureev I. Garanina N.O., Liakh T.V., Rozov A.S., Zyubin V.E., and Gorlatch S.P., “Two-Step Deductive Verification of Control Software Using Reflex,” [J. Programming and Computer Software, 46(4), pp. 261-272, 2020].

[7] Z3 API in Python, https://ericpony.github.io/z3py-tutorial/guide-examples.htm. Last accessed 26 Dec 2020.

[8] The Coq Proof Assistant, https://coq.inria.fr/. Last accessed 26 Dec 2020.

[9] Dijkstra, E.W. and Schölten, C.S. Predicate Calculus and Program Semantics, Texts and Monographs in Computer Science. Springer, New York, 1990.

[10] Proving verification conditions, https://github.com/ivchernenko/HandDryerController-proofs. Last accessed 20 Feb 2021.