ISBN 978-83-60810-22-4

Automated Conversion of ST Control Programs to Why for Verification Purposes

Jan Sadolewski
Rzeszów University of Technology
ul. W. Pola 2, 35-959 Rzeszów, Poland
Email: js@prz-rzeszow.pl

***Abstract*—The paper presents a prototype tool ST2Why, which converts a Behavioral Interface Specification Language for ST language from IEC 61131-3 standard to Why code. The specification annotations are stored as special comments, which are close to implementation and readable by the programmer. Further transformation with Why tool into verification lemmas, confirms compliance between specification and implementation. Proving lemmas is performed in Coq, but other provers can be used as well.**

I. Introduction

IN SOME cases control programs should be formally proved before deployment. Large control systems are usually programmed in IEC 61131-3 standard languages, such like graphical: LD (*Ladder Diagram*), FBD (*Function Block Diagram*), SFC (*Sequential Function Chart*), and textual: ST (*Structured Text*), IL (*Instruction list*). ST language is the most flexible, similar to Pascal, and it is often used by experienced programmers. It allows to declare function, function blocks and programs, called POU (*Program Organization Units*), which are components of the control application.

Contemporary program developing often uses *design by contract* method [8] and Behavioral Interface Specification Languages. JML (*Java Modelling Language*) [6] is a comprehensive example for such languages which uses the method. Such approach can be found in similar tools like Caduceus [5] and Frama C [1] for ANSI C language and in Krakatoa [7] which is also for Java.

The paper presents a proposition of Behavioral Interface Specification Language based on JML for ST language, and shows formal verification of compliance between specification and implementation. It employs multi-target open-source software Why [4] for generating Dijkstra Weakest Preconditions [3], and open-source Coq [2] as backed prover. The work presents an improved ST code verification proposed in [13], [14], which omits translation to ANSI C code and involving of Caduceus. Direct conversion from ST language to Why uses preliminary version of ST2Why, which supports functions, function blocks and programs declarations, but limits ST code to a subset composing of assignments, if statements, while loops and other function block calls.

Verification presented in paper [17] uses embedding of ST constructs in HOL (*Higher Order Logic*) terms. Function block is visible as functional program written in HOL terms, where time is treated like additional input variable (parameter), which stays constant in each round. It main weakness is keeping requirements in LTL (*Linear Temporal Logic*), so the verification is not simple to use by engineers. Developing user friendly language for storing specification annotations conformed with known standards (like JML) can improve applying of formal methods by the programmers.

The paper is organised as follows. Current state of verification of ST programs, and the proposition of improved version are presented in Sec. II. Next section briefly presents useful constructions of JML language adapted to ST and useful in control programs. Section IV describes direct conversion of ST code with specification annotations to Why. Code conversion is performed in three aspects: translating interface POUs into Why language functions, translating POU code into equivalent form, and translating annotations into Why form. Section V presents verification process by example of D flip-flop. The verification is processed half-automatically with prover standard tactics. Lemmas proofs are presented as tactic trees, which describe the proving method.

II. Verification Concept

Freely available software such as Why and Coq allow to be used as programs provers. These tools can prove compliance between specification and implementation and help localising mistakes and side effects of developed programs. Specifications of such programs are stored in annotations located in Why code. For ST language the specification can be saved in special comments (see sec. III), which are invisible for other ST compilers.

One of current verification method is the conversion of

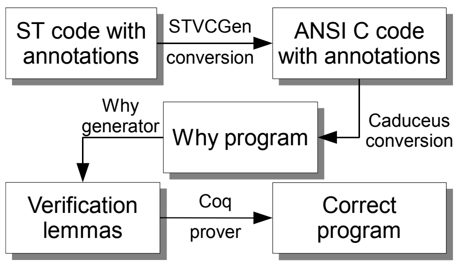


Fig. 1. Current verification of ST language control programs

The research has been supported by MNiSzW under the grant N N516 415638 in years 2010–2011

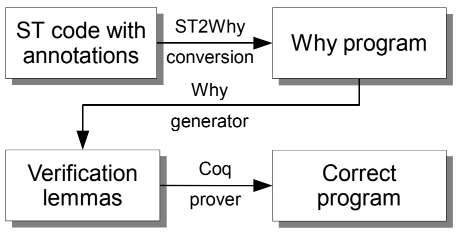


Fig. 2. Improved verification of control programs

annotated ST code to ANSI C and further conversion by Caduceus program into Why language (Fig. 1). In next step Why generator produces verification lemmas in Coq format. Lemmas can be proved half automatically with tactics. If all lemmas are proved then correctness of the code is confirmed.

The verification method uses intermediate form in ANSI C code, which is suitable for small systems. The main weakness of the approach is that one of the most popular ST types – `BOOL` has not corresponding equivalent, and it must be replaced with larger type like `char`. Operations on `char` type are treated by Caduceus like operation on numerical values. It limits access to well known laws on Boolean values like de Morgan Laws, and double negation law. The only one method to prove such numerical lemmas in Coq is using Presburger Arithmetic, but as referenced in [16] the method is slow and may explore many redundant cases. To simpler prove the verification lemma, specification clauses required special processing by the designer, for example clause `inp1=TRUE`, need to be stored as `inp1<>FALSE`. Introducing direct ST translation to Why code (Fig. 2) with ST2Why tool, such disadvantages can be avoided.

The Why language has build-in three types: `bool`, `real`, and `int` which are sufficient for typical programming. Additional user types can be declared, if necessary, with basic arithmetic operations on them. Why contains a collection of libraries which are contributed into provers at install time. They are used as support of verification lemmas and, in some standard cases, can made proving simpler and faster.

III. Behavioral Interface Specification Language for ST

The main purpose for introducing the BISL language is to define behaviour of parts of developed code. Software developing with design by contract use such languages, which can be seen in Eiffel [9], Why, and JML code. The first two languages use build-in constructions for storing specification clauses, the last one uses special kind of comments beginning with '`@`' character. Such method is also used for storing specification in ST language.

Specification clauses are stored as assertions. An assertion is a part of code composed of conditional Boolean expression, which evaluated at time and order of its execution must be satisfied. In design by contract two assertions are commonly used: `requires` to denote preconditions, and `ensures` for postconditions. These assertions must be kept near developed

TABLE I
Adaptation of JML in ST language

Clause type	Standard JML	ST adaptation	Scope
Assertions	`assert`	`assert`	instruction
	`ensures`	`ensures:`	local
	`requires`	`requires:`	local
Localise modifiers	`\at`	`\at` or `at`	instruction
	`\old`	`\old`	instruction
Quantifiers	`\exists`	`\exists`	mixed
	`\forall`	`\forall`	mixed
Invariant	`invariant`	`invariant:`	instruction
Declarations	`label`	`label:`	instruction
	`logic`	`logic:`	global
	`ghost`	`ghost:`	local
	`predicate`	`predicate:`	global
	`axiom`	`axiom:`	global
Function return value	`\result`	`\result` or *function_name*	local
Operations	`set`	`set:`	instruction
	`assigns`	`assigns:`	local
W-F iteration	`variant`	`variant:`	instruction

code, as special comments like mentioned above. They express conditions, which must be satisfied when given subroutine is called and guaranteed at its termination.

Function blocks and programs from IEC standard are similar to lightweight Java objects, so using JML as a base of BISL for ST seems motivated. It is natural that only some subset of JML standard can be applied in control programs, so other features of that will not be described. The adaptation of JML for ST language, called *assertional extension*, is presented at Tab. I and grouped according to clause types. Each clause has its own affection scope. Scope *instruction* means that the clauses can be placed when instructions (or sometimes expressions) are expected. Scope *local* means clauses defined for whole POU, and *global* for whole project (configuration in IEC standard). *Mixed* denotes clause whose use depends on context.

The adaptation of JML for ST has been described in more details in [13]. Verification clauses, with different scope than global, are located inside corresponding unit. For example annotation clause of function block is written after identifier with the name of the block. The clause must contain at least `ensures` section, but often involves `requires` and `assigns` – especially when annotated POU is a program which modifies global variables. The `\result` or function name can be used in `ensures` section to access function return value. Modifier `\old` represents variable value at beginning of execution, and `\at` at specified location in the code which can be declared with `label` clause.

Additional functions not appearing in the code can be obtained with `logic` clause, similar local variables for specification can be defined by `ghost` clause and operated by `set` clause. The `predicate` declares additional logic function which returns Boolean value. The `axiom` generates new axiom which can be used by the prover. Quantifiers appear in declarations of loop invariants, declared with `invariant` clause. To examine if loops are well founded `variant` clause is employed.

IV. Conversion ST to Why

As indicated in Sec. II conversion POUs from ST language into Why form is needed to use open source tools for program verification. The ST2Why tool provides the conversion, which is based on ST compiler included in CPDev package [12]. The parser is built according to top-down scheme with syntax-directed translation. It recognises of ST code shape and produces correspond Why code. In addition to ST code translating, the parser also collects annotations and emits them into valid positions of Why language.

Code translation is performed in three aspects:

- converting POU interfaces,
- converting body code,
- converting specification.

The first aspect concentrates on POUs translating into Why language functions. If POU to be converted is a function, then translation seems obviously, except return value, which must be declared locally to conform to the code. Situation is more complicated with function block or program. Function block is translated into Why function in the following way:

- block inputs are converted into function parameters,
- block outputs become function parameters, but declared as reference,
- local variables are also declared as reference parameters.

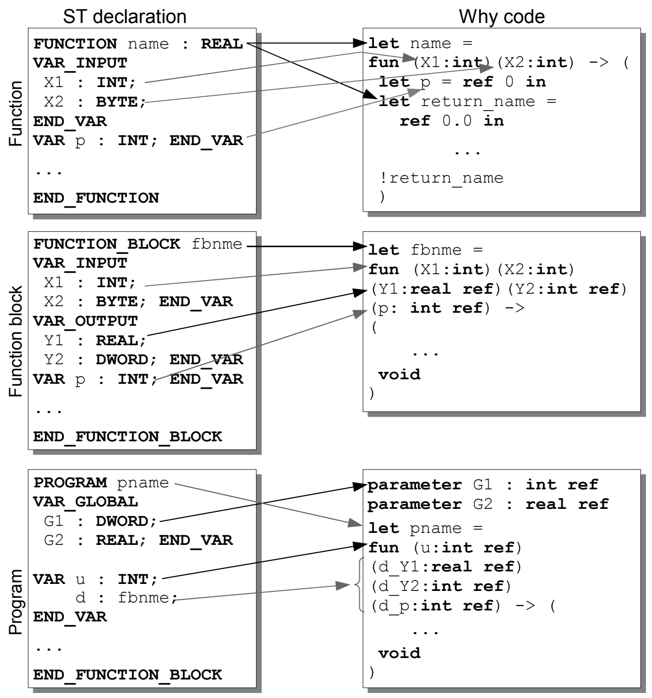
An ST program is translated into Why function as follows:

- global variables are converted into global reference parameters,
- local variables become function parameters, but declared as reference,
- local function block instances are ignored, but their reference parameters are also declared like local variables.

Such interfaces conversion is illustrated in Fig. 3. The IEC 61131-3 defines 20 standard data types, but currently the conversion process is limited for the basic ones `BOOL`, `INT`, `REAL` and `TIME`. Boolean type is translated into `bool` type in Why and floating point type `REAL` is translated into single `real` type. Due to early developing stage of ST2Why converter, all remaining fixed point types are translated into single `int` type in Why.

The second aspect is to convert instruction code into valid Why form. Most of Why common arithmetic operations are available as functions with type name and operation name. As seen at Fig. 4a the sum calculation of two integer values involves `int_add` function. Boolean expressions (Fig. 4b) can be converted without complications, like the `if` statement from Fig. 4c. More effort is necessary with `while` loop conversion. The Why language is functional, so loops can work only on pointers, which are forbidden in ST language. It requires redeclaration as pointers of those variables, which are used in condition expression and are assigned in loop body. Currently none of the remaining loops in ST language (`FOR`, `REPEAT`, etc.) are supported.

Calls of function blocks require more effort, because values of local and output variables from previous execution cycle must be preserved. As shown before in Fig. 3, the program



```
FUNCTION name : REAL
VAR_INPUT
 X1 : INT;
 X2 : BYTE;
END_VAR
VAR p : INT; END_VAR
...
END_FUNCTION
```

```
let name =
fun (X1:int)(X2:int) -> (
 let p = ref 0 in
 let return_name =
   ref 0.0 in
     ...
 !return_name
)
```

```
FUNCTION_BLOCK fbnme
VAR_INPUT
 X1 : INT;
 X2 : BYTE; END_VAR
VAR_OUTPUT
 Y1 : REAL;
 Y2 : DWORD; END_VAR
VAR p : INT; END_VAR
...
END_FUNCTION_BLOCK
```

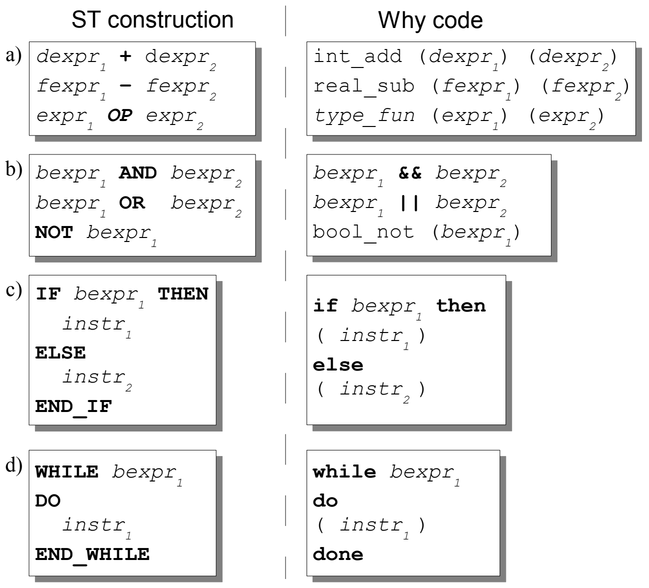
```
let fbnme =
fun (X1:int)(X2:int)
(Y1:real ref)(Y2:int ref)
(p: int ref) ->
(
    ...
 void
)
```

```
PROGRAM pname
VAR_GLOBAL
 G1 : DWORD;
 G2 : REAL; END_VAR

VAR u : INT;
    d : fbnme;
END_VAR
...
END_FUNCTION_BLOCK
```

```
parameter G1 : int ref
parameter G2 : real ref
let pname =
fun (u:int ref)
(d_Y1:real ref)
(d_Y2:int ref)
(d_p:int ref) -> (
    ...
 void
)
```

Fig. 3. Interface conversion of POU from ST to Why language



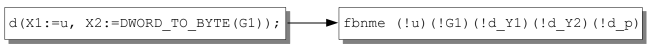
	ST construction	Why code
a)	*dexpr*$_1$ + *dexpr*$_2$ *fexpr*$_1$ - *fexpr*$_2$ *expr*$_1$ ***OP*** *expr*$_2$	int_add (*dexpr*$_1$) (*dexpr*$_2$) real_sub (*fexpr*$_1$) (*fexpr*$_2$) *type_fun* (*expr*$_1$) (*expr*$_2$)
b)	*bexpr*$_1$ **AND** *bexpr*$_2$ *bexpr*$_1$ **OR** *bexpr*$_2$ **NOT** *bexpr*$_1$	*bexpr*$_1$ **&&** *bexpr*$_2$ *bexpr*$_1$ **||** *bexpr*$_2$ bool_not (*bexpr*$_1$)

c)	**IF** *bexpr*$_1$ **THEN**
	instr$_1$
	ELSE
	instr$_2$
	END_IF
		if *bexpr*$_1$ **then**
		(*instr*$_1$)
		else
		(*instr*$_2$)

d)	**WHILE** *bexpr*$_1$
	DO
	instr$_1$
	END_WHILE
		while *bexpr*$_1$
		do
		(*instr*$_1$)
		done

Fig. 4. ST source code to Why conversion

`pname` uses a hypothetical function block `fbname` with the instance called `d`, so additional inputs (beginning with `d_`) have been also declared. Call of the instance `d` in ST code and the translation to Why is presented in Fig. 5. The single variable `d` does not exist here, but it is replaced by corresponding arguments of the converted program. Such approach is necessary to combine complex data type, like function block interface, from elementary data types.

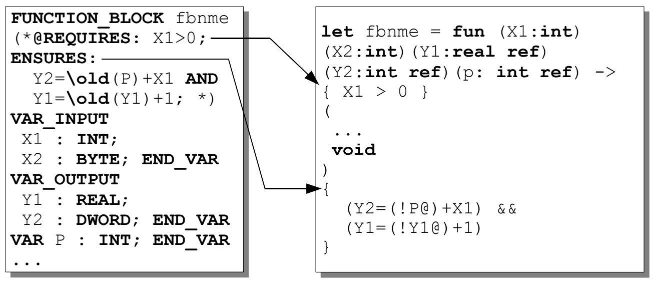


```
d(X1:=u, X2:=DWORD_TO_BYTE(G1));
```

```
fbnme (!u)(!G1)(!d_Y1)(!d_Y2)(!d_p)
```

Fig. 5. Function block call conversion

The third aspect of conversion it to change annotations describing a POU in ST language into equivalent form in Why with necessary modifications. Conversion of annotations affect shape and position of its source. The `REQUIRES` clause is enclosed in { } brackets, with removed introducing word and following colon, and moved to position after the arrow (`->`) sign. Finally, separating semicolon is removed (Fig. 6). The `ENSURES` clause is moved after function implementation, and similar shape modification are performed.



```
FUNCTION_BLOCK fbnme
(*@REQUIRES: X1>0;
ENSURES:
  Y2=\old(P)+X1 AND
  Y1=\old(Y1)+1; *)
VAR_INPUT
 X1 : INT;
 X2 : BYTE; END_VAR
VAR_OUTPUT
 Y1 : REAL;
 Y2 : DWORD; END_VAR
VAR P : INT; END_VAR
...
```

```
let fbnme = fun (X1:int)
(X2:int)(Y1:real ref)
(Y2:int ref)(p: int ref) ->
{ X1 > 0 }
(
 ...
 void
)
{
  (Y2=(!P@)+X1) &&
  (Y1=(!Y1@)+1)
}
```

Fig. 6. Converting ST assertional extension

The composition of those aspects produce coherent Why code, which can be handled by Why tool to produce verification lemmas.

V. Verification Example

The verification example will be presented on developing of D flip-flop. It is a elementary block in control applications, which preserves state of one electric wire. Its symbol and time plot are given at Fig. 7a and Fig. 7b.

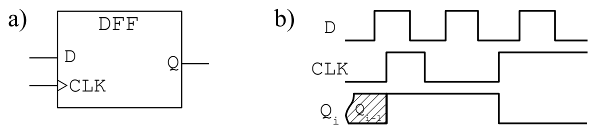


Fig. 7. D flip-flop; a) symbol, b) time plot

The *design by contract* process begins from describing function block requirements by the designer. As it can be seen at Fig. 7b, the output signal `Q` is only changing when raising edge on `CLK` input is detected, otherwise output signal remains unchanged. Detecting raising edge requires additional variable, here called `PCKL`, which holds value `CLK` from previous program cycle. It leads to following ST code interface:

```
FUNCTION_BLOCK DFF
(*@ENSURES:
((CLK=FALSE) ==> (Q=\old(Q))) AND
((\old(PCLK)=FALSE AND CLK=TRUE)
   ==> (Q=D)) AND
((\old(PCLK)=TRUE AND CLK=TRUE)
   ==> (Q=\old(Q))); *)
VAR_INPUT D : BOOL; CLK : BOOL; END_VAR
VAR_OUTPUT Q : BOOL; END_VAR
VAR PCLK : BOOL; END_VAR

END_FUNCTION_BLOCK
```

Analysing `CLK` and `PCLK` input states, one can notice that only `CLK` equal to false determines unchanging the `Q`. It produces following part of ensures expression `CLK=FALSE ==> Q=\old(Q)`. Second part `\old(PCLK)=FALSE AND CLK=TRUE ==> Q=D` can be taken from specification in textual form. The third part is taken from remaining input states which have not been described, so it is `\old(PCLK)=TRUE AND CLK=TRUE ==> Q=\old(Q)`. Because all of the inputs are fully qualified in the specification, and work of the D flip-flop is not restricted, then `REQUIRES` clause remains empty.

In the second stage of the design by contract the developer produces an implementation from the time plot according to given interface and specification:

```
IF (NOT PCLK) AND CLK THEN Q := D; END_IF;
PCLK := CLK;
```

and transforms them with rules mentioned in Sec. IV, or automatically with ST2Why tool into following format:

```
let dff = fun (D:bool)(CLK:bool)
  (Q:bool ref)(PCLK:bool ref) -> {}
((if ((not !PCLK) && (CLK))
  then (Q := D) else void);
 PCLK := CLK)
{ (CLK=false -> Q=Q@) and
  (PCLK@=false and CLK=true -> Q=D) and
  (PCLK@=true and CLK=true -> Q=Q@)
}
```

From that form after Why usage two verification lemmas are obtained:

```
Lemma dff_po_1:forall(D CLK PCLK Q:bool),
forall (HW_1: PCLK=false /\ CLK=true),
forall (Q0: bool), forall (HW_2: Q0=D),
forall (PCLK0: bool),
forall (HW_3: PCLK0 = CLK),
(((CLK=false -> Q0=Q)) /\ ((PCLK=
  false /\  CLK=true -> Q0=D)) /\
((PCLK=true /\ CLK=true -> Q0=Q))).

Lemma dff_po_2:forall(D CLK PCLK Q: bool),
forall (HW_4: PCLK=true \/ PCLK=false
   /\ CLK=false), forall (PCLK0: bool),
forall (HW_5: PCLK0=CLK),
(((CLK=false -> Q=Q)) /\ ((PCLK=false
   /\ CLK=true -> Q=D)) /\
((PCLK=true /\ CLK=true -> Q=Q))).
```

The proofs of the lemmas can be performed in Coq via a set of tactics. The tactic is a prover command which:

- transforms lemma into hypotheses and goals, or
- splits goal into subgoals, or
- indicates agreement between hypothesis and goal, or
- contradicts two hypotheses.

Verification begins from the whole lemma which is used as goal in empty context. As the first lemma `dff_po_1` is taken for proving. Tactic `intros` introduces new hypotheses from

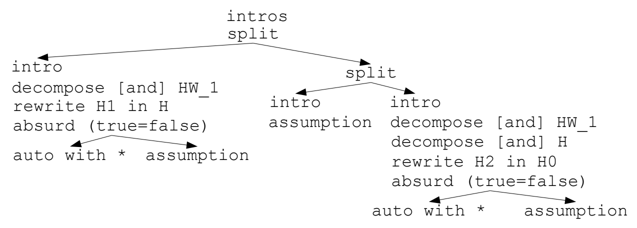


Fig. 8. Coq proof of Lemma 1

goal to context (Fig. 8). The goal will change into three conjunctions, so `split` tactic is necessary to divide goal into two subgoals. The first subgoal contains implication, so `intro` tactic is used to introduce `H0` hypothesis. Further it could be observed that part of hypothesis `HW_1` (`CLK=true`) is in contradiction with `H` (`CLK=false`). It leads to the following method: changing the one of the occurrences of variable with opposite value and proving as hypotheses contradiction. To extract a part of the hypothesis with conjunction, `decompose` tactic is used. It produces additional hypotheses in context with separated parts. Value from `H1` hypothesis is applied into `H` with tactic `rewrite H1 in H`. The hypothesis `H` become contradiction (`true=false`), so tactic `absurd (true=false)` is applied. Tactic `absurd` produces also two subgoals, fist with negated contradiction, and second with the contradiction itself, so `auto with *` proves the first one (`true<>false`), which is handled automatically by internal libraries, and `assumption` proves the second one, due to existing such hypothesis in context.

After that prover returns into subgoal which was left after the first `split` command. Because it is also conjunction, so another `split` is necessary. Current first subgoal can be proved with `intro` and `assumption` which matches goal with `HW_2` hypothesis. In the last one goal after `intro` another contradiction in hypotheses can be found. Variable `PCLK` from part of hypothesis `H` cannot be equal to `true` and also equal to `false` in part of `HW_1` hypothesis. It leads to mentioned verification method with tactic `absurd`. Detailed proof of lemma `dff_po_` is presented as tree in Fig. 8. To obtain a list of proving commands for Coq simple *in-order* tree walk (begin from root node, then its left child tree, and next right child tree) should be performed.

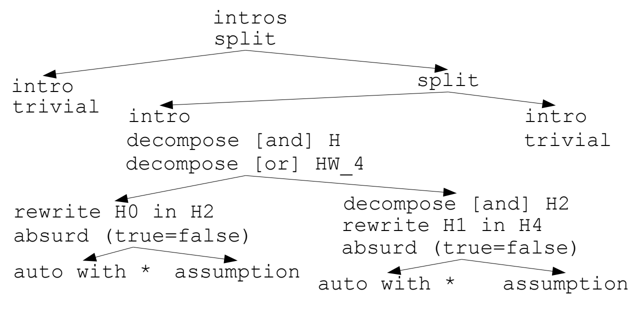


Fig. 9. Coq proof of Lemma 2

The second lemma (`dff_po_2`) can be proved in similar way. The proof begins from `intros` and `split` tactics (Fig. 9). The first subgoal is a implication, so `intro` tactic is applied. After that goal reduces to `Q=Q` form, which is very simple and can be proved with `trivial` tactic. The second subgoal is still conjunction so another `split` is required. After introducing new hypothesis `H` with `intro` tactic, current first subgoal have contradicted hypotheses. First decomposition of first hypothesis is performed with tactic `decompose [and] H`, and due to disjunction in `HW_4` the tactic `decompose [or] HW_4` is used. The last one tactic splits goal for two subgoals, each one with separated part of disjunction as hypothesis. In current first subgoal assignment with `rewrite` tactic is needed and `absurd` command can be applied. In the second subgoal additional `decompose` is necessary before rewriting, and `absurd` applying. The remaining subgoal can be proved exactly like the first one with `intro` and `trivial` tactics.

Proving all verification lemmas confirms compliance between specification and implementation. Developed code formally satisfies designer guidelines, and the contract has been fulfilled.

Presented verification tactics are not comprehensive for all programs, especially when program contains integer or floating point variables. For that lemmas more complex tactics such like `omega`, `ring` and `Fourier` are needed. Complete reference for all build-in tactics in Coq can be found in [16]. It may be helpful for an inexperienced users, but in some cases `intuition` tactic may prove the goal automatically or to present reason for which current goal cannot be proved.

VI. Summary

The method of proving programs written in ST language with BISL extension has been presented. The language extension is stored as special comment inside the function, function block or program being verified. It accords to JML language which is commonly used in design by contract developing approach. Conversion with ST2Why and Why tools produce verification lemmas. Verification of compliance between specification and implementation can be performed in a few provers, but here only simple build-in Coq tactics have been used. Other provers like PVS [11] or Mizar [10] can be also used, it requires only one additional parameter in Why call, which will change the output shape of lemmas.

Future work will concentrate on transforming remaining clauses of ST language into Why code (like `REPEAT` and `FOR` loops, `CASE` statements), and on introducing remaining data types into Why, conformed with ST language types. It may require to develop additional Why libraries, where their logical definitions will be stored.

References

[1] P. Baudin, P. Cuoq, J. Ch. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, “ACSL: ANSI/ISO C Specification Language”, http://frama-c.cea.fr, 2011.

[2] Y. Bertot, P. Castéran, *Interactive Theorem Proving and Program Development*, Springer-Verlag, Berlin Heidelberg, 2004.

[3] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall Inc., 1976.
[4] J. Ch. Filliâtre, "The Why verification tool. Tutorial and reference manual", http://www.lri.fr, 2011.
[5] J. Ch. Filliâtre, T. Hubert, C. Marché, "The Caduceus verification tool for C programs", http://caduceus.lri.fr, 2008.
[6] G. T. Leavens, A. L. Baker and C. Ruby, "JML: a Notation for Detailed Design", Behavioral Specifications of Businesses and Systems. 1999.
[7] C. Marché. "The Krakatoa verification tool for Java programs. Tutorial and reference manual", http://proval.lri.fr.
[8] B. Meyer, "Applying design by contract", *Computer*, vol. 25, no. 10, pp. 40-51, 1992.
[9] B. Meyer, *Eiffel: the language*. Object-Oriented Series, Prentice Hall New York, 1992.
[10] M. Muzalewski, *An outline of PC Mizar*, Foundation Philippe le Hodey, Brussels, 1993.
[11] S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert, "PVS system guide", SRI International, 2001.
[12] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, L., "A Control Program Developer". XXI MicroCAD International Scientific Conference, Miskolc, March 2009, pp. 49-54.
[13] J. Sadolewski, Assertional extension in ST language of IEC 61131-3 standard for control systems dynamic verification, *Pomiary Automatyka Robotyka*, no 2, pp. 305-314, 2011 (in Polish).
[14] J. Sadolewski, An introduction to verification of simple programs in ST language with Coq, Why and Caduceus tools, *Metody Informatyki Stosowanej*, vol. 19, no 2, pp. 121-138, 2009 (in Polish).
[15] J. Sadolewski, Conversion of ST Control Programs to ANSI C for Verification Purposes, *e-Informatica Software Engineering Journal*, (in review).
[16] The Coq Development Team, *The Coq Proof Assistant Reference Manual*, Ecole Polytechnique, INRIA, Universit de Paris-Sud, http://coq.inria.fr, 2010.
[17] N. Völker, B. J. Krämer, "Modular Verification of Function Block Based Industrial Control Systems", *in Proceedings of Joint 24th IFAC/IFIP Workshop on Real-Time Programming and the 3rd International Workshop on Active and Real-Time Database Systems*, Schloß Dagstuhl, Germany, May 30th – June 2nd, 1999.