e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 65–76, DOI 10.2478/v10233-011-0031-3

VERSITA

Conversion of ST Control Programs to ANSI C for Verification Purposes

Jan Sadolewski*

*Department of Computer and Control Engineering, Rzeszow University of Technology

js@prz-rzeszow.pl

Abstract

The paper presents a Behavioral Interface Specification Language for control programs written in ST language of IEC 61131-3 standard. The specification annotations are stored as special comments in ST code. The code and comments are then converted into ANSI C form for further transformation with Caduceus and Why tools. Verification of compliance between specification and code is performed in Coq.

1. Introduction

In some safety oriented applications control programs should be formally proved before deployment in the controllers. Control systems are usually programmed in languages of IEC 61131-3 standard, however ANSI C is typically used for prototype systems. The IEC standard defines five programming languages, i.e. LD, IL, FBD, ST and SFC, allowing the user to choose the one suitable for particular application. Instruction list (IL) and Structured Text (ST) are text languages, whereas Ladder Diagram (LD), Function Block Diagram (FBD) and Sequential Function Chart (SFC) are graphical ones.

Recently developed compiler called MatPLC [1] converts the code from ST, IL, FBD and LD languages into ANSI C form. It seems that the main purpose of MatPLC developers was to provide equivalent ANSI C code for small hardware platforms and prototypes, where IEC languages are not available.

This paper presents somewhat different approach to code conversion, focusing instead on extension of ST language towards formal verification of compliance between specification and implementation. The conversion can also be used for *design by contract* method [2] in which clauses describe specification. The approach employs open source software Caduceus [3], Why [4] and Coq [5], whose connection can be used for formal verification of ANSI C programs. The specification is based on adaptation of JML language [6] for ST. Special annotations stored as comments express Dijkstra Weakest Preconditions [7] for programs, functions and function blocks (Program Organization Units in ST). The method presented here starts from ST source code with annotations and uses automated tools to obtain lemmas whereas approach described in chapter [8] starts from function blocks models written in Why language by hand. The annotation extending ST language was proposed in [9] and currently developed features are presented here.

The paper is organised as follows. Current state of verification of C programs, and corresponding concept of verification of ST programs are presented in Section 2. Next section briefly describes assertions and useful constructs of JML language adapted to ST. Section 4 describes translation of ST code with specification annotations to ANSI C with corresponding annotations. The translation is made automatically by program STVCGen developed for the purpose of

[0] The research has been supported by MNiSzW under the grant N N516 415638 (2010–2011).

this paper. Code translation takes into account three aspects: (1) translation of POU interfaces into C language functions, (2) conversion of POU ST code into equivalent C form, (3) translation of specification annotations into C form for Caduceus. Example of conversion of TON standard function block (timer), supplemented with specification annotations is presented in Section 5. Section 6 describes verification process of C code of the TON block (it becomes a function). The verification is processed half-automatically with standard tactics from Coq prover. For one of the lemmas the whole proof tree is presented, which can be of some help for similar examples.

2. Verification Concept

Freely available software such as Caduceus, Why and Coq can be used to verify correctness of programs written in ANSI C language. These tools may prove compliance between specification and implementation, or help to find mistakes and side effects. Specifications of programs are stored in annotations placed in special comments as BISL code (Behavioral Interface Specification Language). The Caduceus program converts the annotated C code to Why language (Fig. 1, second and third blocks). In the following step, Why generator produces verification lemmas based on Dijkstra Weakest Preconditions. Such lemmas are stored in Coq format, for further proving with tactics. If all the lemmas are proved, then correctness of the code is confirmed.

Control programs are typically written in ST language, so as to use such approach it is necessary to convert ST to C at the beginning, as shown in Fig. 1. A prototype tool called STVC-Gen described in Section 4 converts ST language code supplemented with specification annotations into C code with corresponding annotations. This is further converted by Caduceus into Why program. After applying Why generator we obtain a collection of lemmas to be proved by Coq.

3. Behavioral Interface Specification Language for ST

The main purpose for introducing the BISL languages was to define behaviour of components of developed code. Such languages are used in *design by contract* programming methods. Generally speaking the BISL languages are based on assertions, examined at run-time. Some languages like Eiffel and Why use build-in clauses for storing such assertions, but popular languages like Java and C use special kind of comments beginning with '`@`' character.

An assertion is a part of code composed of conditional Boolean expression, which should be satisfied when evaluated at specific place of the executed program (i.e. it returns true). Typical assertions from popular languages are shown in Tab. 1. They are used solely for testing purposes, and their code is not compiled into the final distribution. Assertion failure may be represented by message box, with exception or interruption of program execution. The message may involve current call stack, place in source code, etc.

Design by contract uses two special assertions, i.e. `requires` to denote preconditions, and `ensures` for postconditions. They must be kept near developed code, in the form of special comments beginning with '`@`' mentioned above. The assertions express conditions, which must be satisfied when given subroutine is called, and conditions guaranteed at its termination.

Java Modelling Language (JML) is an example of a BISL language, which uses comments to store annotations. This feature allows for code migration between compilers of different providers, which do not support annotations. Program Organization Units (POUs) from IEC standard are similar to lightweight Java objects, so JML can be adapted as a base of BISL language for ST. Naturally, only a subset of JML will suffice for verification problem considered here.

Adaptation of JML for ST language is presented at Tab. 2. The clauses are grouped according to their types. Each clause has its own range. Range *instruction* means that corresponding clause can be placed where instruction or expression is expected. Ranges *local* and *global* re-

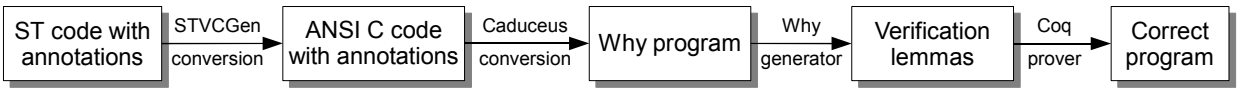

Figure 1. Method for verification of ST programs

Table 1. Assertions in popular languages

ANSI C, POSIX	Delphi	C#

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

Table 2. Adaptation of JML in ST language

Type	Standard JML	ST adaptation	Range
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

fer to POU or whole project, respectively. Clause whose use depends on the context has the range *mixed*.

Verification clauses are located inside corresponding program 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 it often involves `requires` and `assigns`, especially when annotated POU is a program which modifies global variables. There are two ways to access return value of function, i.e. `\result` or function name, which is specific for ST language. Verification is based on memory states [10], which contain variable values at specified moment of execution. Modifier `\old` represents variable value at the beginning of execution, obtained in previous cycle. Similarly, modifier `\at` denotes variable value at specified location in the code, declared with `label`.

Sometimes additional function that does not appear in the original code may help in construction of specification. The function can be reached by global `logic` clause. Additional local variables can be used to express the specification. Such variables are defined by `ghost` clause and operated on 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. They may also examine if the loops are well founded.

More details on adaptation of JML for ST are given in [9].

4. Conversion of ST to ANSI C

As indicated in Section 2, conversion of POUs from ST language into ANSI C code is needed to use open source tools for program verification. The STVCGen tool based on ST compiler from CPDev package [11] executes the conversion. Components of the compiler are classes in C# language, so they can be reused with typical mechanisms like inheritance and overriding. Main goal while developing the STVCGen has been to get a compiler quickly from existing code of CPDev. The parser is built according to top-down scheme with syntax-directed translation [12]. It recognises meaning of ST code and produces corresponding ANSI C code. In addition to translating ST, the parser collects annotations and generates code for Caduceus or Frama C tools[1].

Code translation is performed in three aspects, i.e. concerning POUs, instructions, and annotations, respectively. The first one is to translate POUs into C language functions. If POU is a function, then translation proceeds directly. Return value must be declared only to conform with the code. Translation of function block or program is more complicated. Function block is translated into C function in the following way:

- block inputs are converted into function parameters,
- block outputs become function parameters, however declared as pointers,
- local variables are also declared as pointer parameters,
- all pointer parameters produce extra `requires` expression with different base addresses.

An ST program is translated into C as follows:

- global variables remain global in C,
- local variables become function parameters, declared as pointers,
- local function block instances are ignored, but their pointer parameters are also declared as additional pointers.

The conversion cases are illustrated in Fig. 2. ST variable types are converted into corresponding C types, with equivalents presented in Table 3.

The second aspect is to convert instruction code into valid C form. Generally speaking, code shape in both languages is similar, so examples presented at Fig. 3a where *OP* is arithmetic or logic operator are natural. Most of ST operators have equivalents in C, so C code construction involves operator replacements and parentheses in case of different priorities. The problem arises when converted variable after conversion is declared as a pointer. In such case each instance must appear in C code with a star and parentheses. If an ST operator does not have C equivalent (like power `**`), STVCGen replaces it with function provided by header file bundled with the tool, as in Fig. 3a (macros). Some ST operators have more equivalents in C code. For example `AND` operator may be logical operator between Boolean expressions $\texttt{bexpr}_1$ and $\texttt{bexpr}_2$ (Fig. 3b), and can be also used for bitwise calculations in digital expression involving $\texttt{dexpr}_1$ and $\texttt{dexpr}_2$. When such operator (`AND`, `OR`, `NOT`) appears in source code, the compiler checks if the expression evaluates to Boolean. If yes, the logical operator is used, otherwise bitwise one. Conversion of `NOT` operator may lead to one of two macros. When the operand is Boolean then the `NOT` operator is converted to '`!`' in C hidden under `_BOOL__NOT__` macro. If the operand is bitwise, `NOT` is converted to '`~`' in `_BIT__NOT__`.

Some ST and C constructs are very similar, as `IF` statement in Fig. 3c. Conditional Boolean expression remains valid after conversion into C. This does not happen however, in case of `FOR` loop whose conversion depends on values in source code. If the constant $\texttt{im}_3$ in Fig. 3d is greater than zero, then the equivalent C statement uses less or equal comparison and increment operator. If the constant is lower than zero, then the C statement uses greater or equal comparison and decrement operator.

[1] Due to different annotations, Caduceus or Frama-C are chosen by compiler settings.

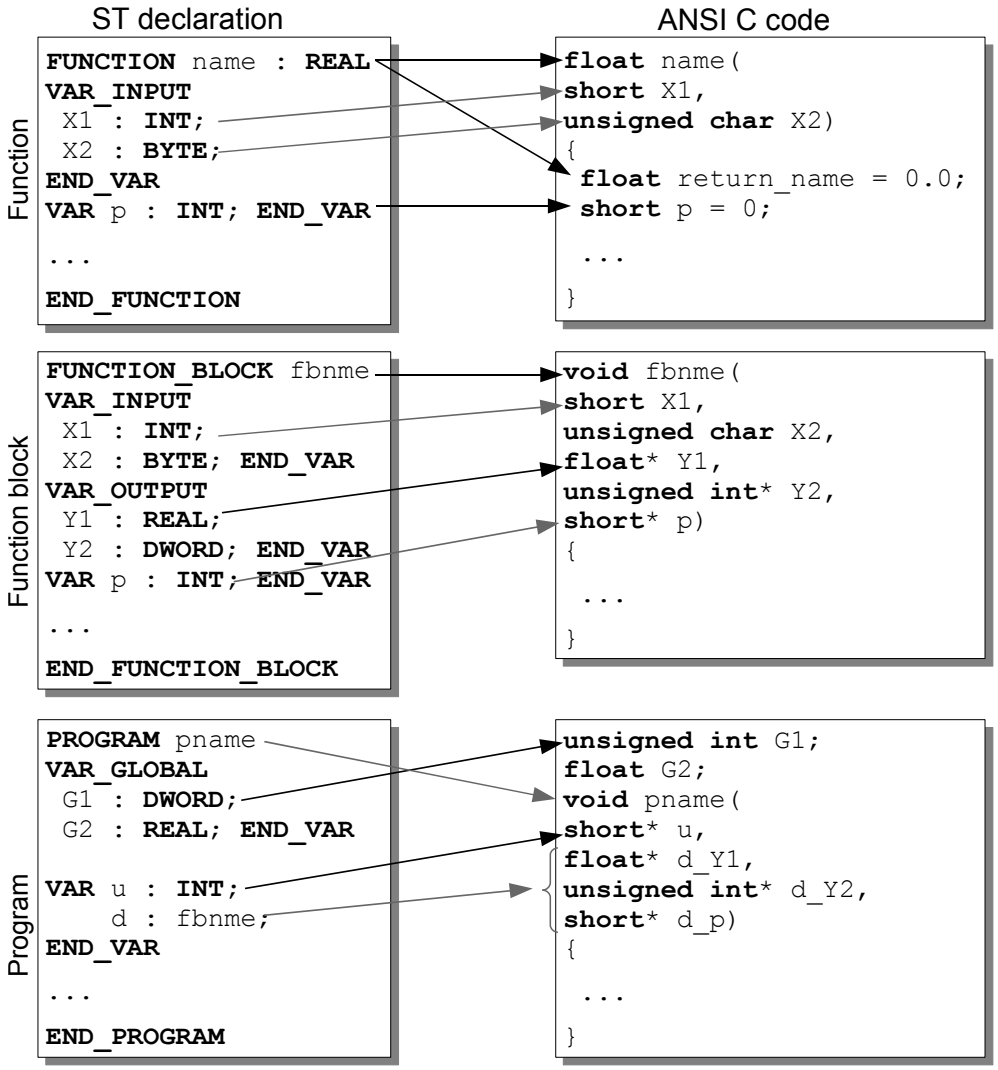

Figure 2. Conversion of three types of POUs into C

Calls of instances of function blocks require more effort, because values of local and output variables from previous execution must be preserved. As shown before in Fig. 2, the program `pname` uses a hypothetical function block `fbname` with the instance called `d`, so additional function inputs (beginning with `d_`) have also been declared. Call of the instance `d` in ST and the translation to ANSI C are presented in Fig. 4. The single variable `d` does not exist here, but is replaced by corresponding arguments of the converted program. Such approach produces less complicated verification lemmas, which can be proved half automatically.

The third aspect of conversion is to change annotations describing a POU in ST language into equivalent form in C with necessary modifications and supplements.

Converted annotations do not differ much from original ones, except operator syntax and removal of some characters not needed by Caduceus (ST assertional extension involves characters that specify range and objective of some clauses). However, the conversion generates additional components in specification, mostly describing pointer properties and arithmetic, including different base addresses for pointer variables and their non-`NULL` values. Since pointers do not exist in ST, therefore each variable, so also a pointer, is allocated at different address. Values different than `NULL` are preserved by task allocator, which can execute programs only with

Table 3. Type conversion

ST type	C type	ST type	C type
BOOL	char	BYTE	unsigned char
SINT	char	INT	short
WORD	unsigned short	DINT	int
DWORD	unsigned int	LINT	long long
REAL	float	LREAL	double
LWORD	unsigned long long	TIME	int
DATE_AND_TIME	unsigned long long	TIME_OF_DAY	unsigned int

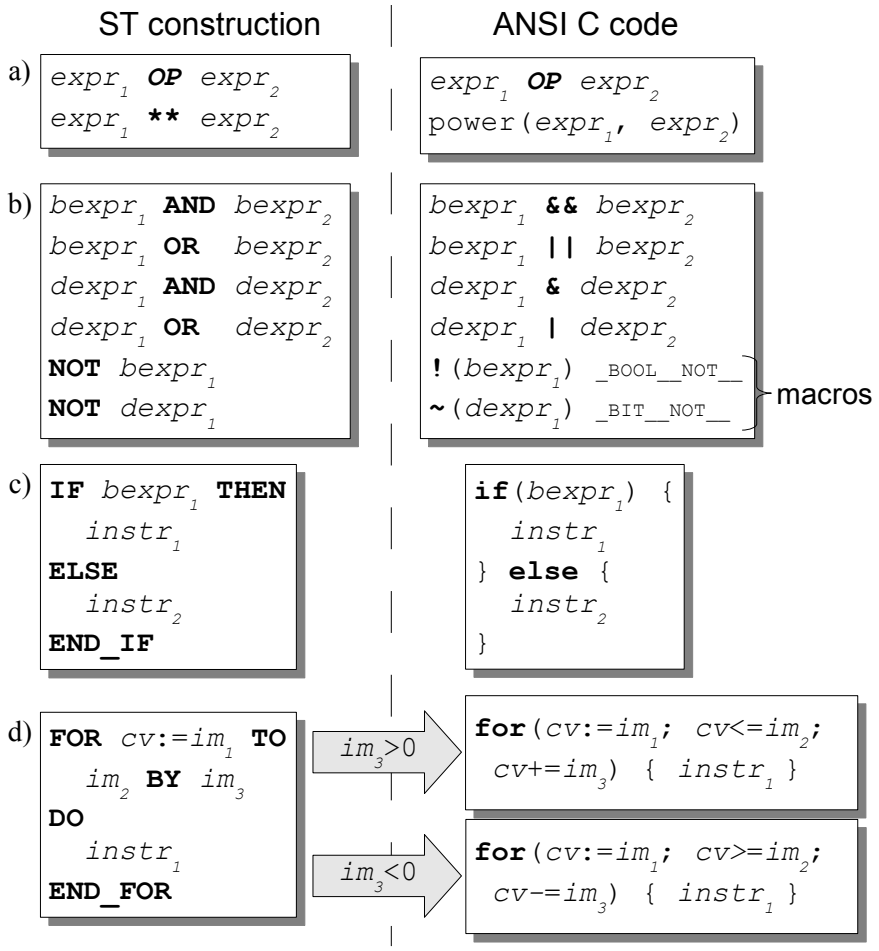

Figure 3. ST code conversion to C form

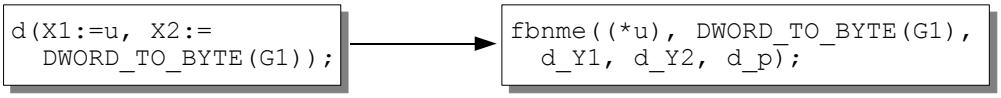

Figure 4. Conversion of function block call

complete set of parameters. The example in Fig. 5 presents an instance of assertional ST extension and converted form in ANSI C for Caduceus. The clause `requires` is directly converted into destination form and supplemented with expression (denoted by circled 1) which by the clause `\valid` indicates non-`NULL` values of pointer variables, and by the clause `\base_addr` assures different addresses pointed by the pointers. The use of pointer variables at C side also requires `assigns` clause (circled 2), which defines variables changed by the function.

Application of the three translation aspects in STVCGen produces coherent ANSI C code, which can be handled by verification tools like Caduceus, Why and Coq.

5. Example of TON function block

As stated in Sec. 2, the sequential verification process consists of source code transformations from ST language with annotations through ANSI C and Why into verification lemmas (Fig. 1). The example considered now involves function block TON (on-delay timer) of Fig. 6a, whose input-output time plots are shown in Fig. 6b. The plots can be split into three parts (states) denoted by the circled digits. The ST source code with specification annotations at the beginning is presented in Fig. 7. Each part of the plot is associated with a single line in `ENSURES` specification clause. In design by contract approach the clause expression and block interface (inputs and outputs declaration) are written by designer. Construct *var*`<>FALSE` implies that Boolean variable *var* equals `TRUE`. It is necessary, because strict Boolean type does not exist in C language. Here it is simulated by integer value zero (`FALSE`) and non-zero (`TRUE`).

The implementation code beginning from `IF` defines instructions to be performed. The `REQUIRES` clause defines constraints. If they are not satisfied, execution of the block may return invalid results. The constraints are also used in verification. The ST code from Fig. 7 is translated by STVCGen to ANSI C form presented in Fig. 8 (in printable version[2]).

According to Sec. 4, function block TON becomes function in C, and `requires` clause is strengthened with `\valid` and `\base_addr` constructs. Block outputs and local variables become pointers in C, so additional clause `assigns` is necessary to deal with pointer arithmetic while proving. Transformation of function block body applies statement conversion and pointer substitution of some variables.

6. ANSI C Verification

The ANSI C code of Fig. 8 is further converted with Caduceus which produces equivalent program in Why code (Fig. 2). In the next step the Why tool generates verification lemmas, which must be proved to confirm program correctness. Details of Caduceus and Why conversion are skipped due to limited space. Here we focus on the lemmas produced by Why generator. In case of TON function (Fig. 8), Why produces 12 lemmas which must be proved with Coq Proof Assistant. First four lemmas refer to correct allocation of variables declared as pointers. One of them is presented in Fig. 9, remaining lemmas have different variable in the goal part (last not indented line). They are easily proved with default tactic `intuition`. Fifth lemma listed in Fig. 10 deals with first possible execution of the program and is more complex. Using `intuition` tactic to prove it leads to undetermined value. This means that `intuition` must be replaced by elementary tactics.

At first `intros` tactic is applied, which introduces local hypothesis into the context. The following `repeat split` splits the goal into five subgoals (denoted as circled numbers in Fig. 11). The first subgoal can be proved by sequential reduction of subsequent memory states (`subst intM_global` with appropriate number), and `caduceus` tactic, when reduction reaches the initial state. The second subgoal invloves contradiction in hypotheses, so one of the opposite hypotheses is passed as argument to the `absurd`

[2] Actually STVCGen produces output with a lot of brackets, so it is not easily readable.

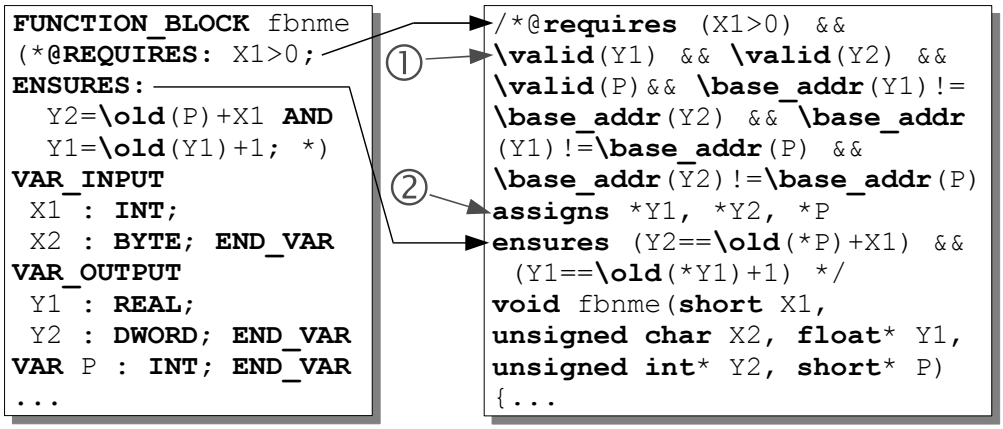

Figure 5. Converting assertional extension with supplements

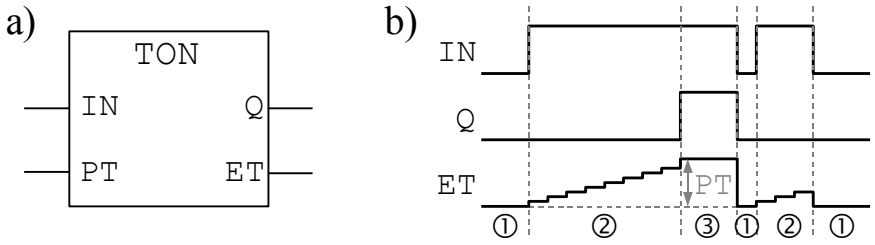

Figure 6. TON function block: a) symbol, b) time plots

```
FUNCTION_BLOCK TON
(*@REQUIRES: (PT > TIME#0ms) AND (ET >= TIME#0ms) AND (ET <= PT);
ENSURES: ((IN = FALSE) ==> ((ET=TIME#0ms) AND (Q=FALSE))) AND
(((ET < PT) AND (\old(Q)=FALSE) AND (IN<>FALSE)) ==> (Q=FALSE)) AND
(((ET = PT) AND (\old(Q)=FALSE) AND (IN<>FALSE)) ==> (Q<>FALSE));  *)
VAR_INPUT  IN : BOOL;  PT : TIME;  END_VAR
VAR_OUTPUT  Q : BOOL;  ET : TIME;  END_VAR
VAR   L_STIME : TIME;  LC : TIME;  END_VAR

IF IN THEN
  IF NOT Q THEN
      LC := CUR_TIME() - L_STIME;
       IF LC >= PT THEN  Q := TRUE;  ET := PT;
       ELSE   ET := LC;
       END_IF
  END_IF
ELSE
     Q := FALSE;  ET := TIME#0ms;  L_STIME := CUR_TIME();
END_IF
END_FUNCTION_BLOCK
```

Figure 7. Source code of TON function block

```
/*@requires (((PT>0x00000000) && (*ET>=0x00000000)) && (*ET<=PT)) &&
\valid(Q) && \valid(ET) && \valid(L_STIME) && \valid(LC) &&
\base_addr(Q)!=\base_addr(ET) && \base_addr(Q)!=\base_addr(L_STIME) &&
\base_addr(Q)!=\base_addr(LC) && \base_addr(ET)!=\base_addr(L_STIME) &&
\base_addr(ET)!=\base_addr(LC) && \base_addr(L_STIME)!=\base_addr(LC)
 assigns *Q, *ET, *L_STIME, *LC
 ensures ((IN==0) => ((*ET==0x00000000) && (*Q==0))) &&
  (((((*ET<PT) && (\old(*Q)==0)) && (IN!=0)) => (*Q==0)) &&
  ((((*ET==PT) && (\old(*Q)==0)) && (IN!=0)) => (*Q!=0))) */
void TON (char IN, int PT, char* Q, int* ET, int* L_STIME, int* LC)
{
 if(IN) {
    if(_BOOL__NOT__(*Q)) {
      *LC = CUR_TIME() - *L_STIME;
      if(*LC >= PT) { *Q = 1; *ET = PT; }
      else { *ET = *LC; }
    }
 } else {
 *Q = 0; *ET = 0x00000000;  *L_STIME = CUR_TIME();
 }
}
```

Figure 8. Result of ST conversion to ANSI C

```
Lemma TON_impl_po_1 :
 forall (IN: Z), forall (PT: Z), forall (Q: (pointer global)),
 forall (ET: (pointer global)), forall (L_STIME: (pointer global)),
 forall (LC: (pointer global)), forall (alloc: alloc_table),
 forall (intM_global: (memory Z global)),
 forall (HW_1: ((((((((((((PT > 0 /\ (acc intM_global ET) >= 0) /\
 (acc intM_global ET) <= PT) /\ (valid alloc Q)) /\ (valid alloc ET)) /\
 (valid alloc L_STIME)) /\ (valid alloc LC)) /\
 ~((base_addr Q) = (base_addr ET))) /\
 ~((base_addr Q) = (base_addr L_STIME))) /\
 ~((base_addr Q) = (base_addr LC))) /\
 ~((base_addr ET) = (base_addr L_STIME))) /\
 ~((base_addr ET) = (base_addr LC))) /\
 ~((base_addr L_STIME) = (base_addr LC)))), forall (HW_2: IN <> 0),
(valid alloc Q).
```

Figure 9. First lemma generated by the Why

tactic. Two generated subgoals are proved with `assumption`. The third subgoal is proved in the same way as the first one. The fourth subgoal, after introduction of additional hypothesis and decomposition of the conjunction in hypothesis `HW_1`, can also be proved like the first subgoal.

The fifth subgoal requires more effort, because it begins with `not_assigns` clause for pointer arithmetic. However, standard scheme described in [3] can be applied to prove corresponding lemmas, extended here to handle four pointer variables (instead of one). At first the `intros A B C`[3] tactic is applied. Then we must duplicate the last lemma `C` so many times, as to match the number of variables declared as pointers, except the first one (so three here). It is done with `generalize C` and `intro D`, or `E`, or `F` tactics (see Fig. 11). Next the `apply`

[3] A sequence of Latin letters is used for brevity. If one of them is already used in the lemma, it can be replaced with another unique name.

```
Lemma TON_impl_po_5 :
 forall (IN: Z), forall (PT: Z), forall (Q: (pointer global)),
 forall (ET: (pointer global)), forall (L_STIME: (pointer global)),
 forall (LC: (pointer global)), forall (alloc: alloc_table),
 forall (intM_global: (memory Z global)),
 forall (HW_1: ((((((((((PT > 0 /\ (acc intM_global ET) >= 0) /\
  (acc intM_global ET) <= PT) /\ (valid alloc Q)) /\ (valid alloc ET)) /\
  (valid alloc L_STIME)) /\ (valid alloc LC)) /\
  ~((base_addr Q) = (base_addr ET))) /\
  ~((base_addr Q) = (base_addr L_STIME))) /\
  ~((base_addr Q) = (base_addr LC))) /\
  ~((base_addr ET) = (base_addr L_STIME))) /\
  ~((base_addr ET) = (base_addr LC))) /\
  ~((base_addr L_STIME) = (base_addr LC)))),
 forall (HW_2: IN <> 0),
 forall (HW_3: (valid alloc Q)), forall (result: Z),
 forall (HW_4: result = (acc intM_global Q)),
 forall (HW_5: result = 0),
 forall (result0: Z), forall (HW_7: (valid alloc L_STIME)),
 forall (result1: Z),
 forall (HW_8: result1 = (acc intM_global L_STIME)),
 forall (HW_9: (valid alloc LC)),
 forall (intM_global0: (memory Z global)),
 forall (HW_10: intM_global0 = (upd intM_global LC (result0 - result1))),
 forall (HW_11: (valid alloc LC)),
 forall (result2: Z),
 forall (HW_12: result2 = (acc intM_global0 LC)),
 forall (HW_13: result2 >= PT),
 forall (HW_14: (valid alloc Q)), forall (intM_global1: (memory Z global)),
 forall (HW_15: intM_global1 = (upd intM_global0 Q 1)),
 forall (HW_16: (valid alloc ET)),
 forall (intM_global2: (memory Z global)),
 forall (HW_17: intM_global2 = (upd intM_global1 ET PT)),
((((IN = 0 -> (acc intM_global2 ET) = 0 /\ (acc intM_global2 Q) = 0)) /\
 ((((acc intM_global2 ET) < PT /\ (acc intM_global Q) = 0) /\ IN <> 0 ->
 (acc intM_global2 Q) = 0))) /\ ((((acc intM_global2 ET) = PT /\
 (acc intM_global Q) = 0) /\ IN <> 0 -> (acc intM_global2 Q) <> 0))) /\
 (not_assigns alloc intM_global intM_global2 (pset_union (pset_singleton
 LC) (pset_union (pset_singleton L_STIME) (pset_union (pset_singleton ET)
 (pset_singleton Q))))))).
```

Figure 10. Fifth lemma generated by Why

`pset_union_elim1 in C` is applied for hypothesis `C` to eliminate first set in the union of sets with pointer variables (`pset`). For the second hypothesis `D` the tactic `elim2` is applied first, followed by `elim1` as before. The remaining `E` and `F` hypotheses require increase of `elim2` applications by one, followed by single `elim1` in `D`, and not in the last `F`. After these steps the tactic `apply pset_singleton_elim in _` applied for all four hypotheses provides pure distinction between addresses of pointer variables. Last steps involve the approach used already to prove the first and third subgoal, i.e. `subst intM_global` tactic to reduce memory states, terminated by final `caduceus` tactic.

The remaining lemmas 6, 8, 9, 10 and 11 are proved automatically with `intuition` tactic. Lemmas 7 and 12 can be proved similarly as lemma 5.

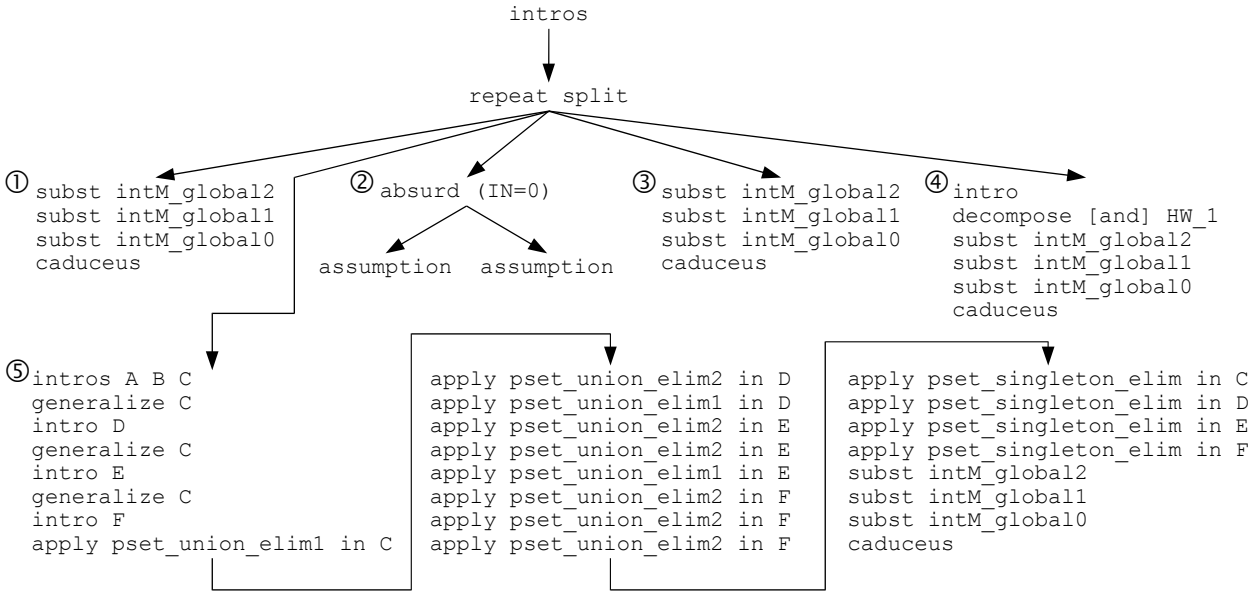

Figure 11. Tactics of proof tree for TON function block lemma

The method presented here can be used for verification of simple functions, function blocks and programs which do not call other function blocks. The limitation is caused by annotation injection arising when a subroutine is called, so some kind of decomposition must be used to deal with it. More information on decomposition can be found in [8].

7. Summary

The application of Behavioral Interface Specification Language for ST language of IEC 61131-3 standard concerning control programs has been presented. The annotations to ST code express specification of function, function block or program, which after conversion to C can be used for formal verification of compliance between specification and implementation. Such approach is typical for design by contract method applied while developing advanced applications. Stepwise conversion by STVCGen, Caduceus and Why tools produce verification lemmas which can be proved by Coq with a set of appropriate tactics. Till now several function blocks and programs have been verified. The examples involve combinatorial logic (binary multiplexer, two-bit sum, heater control), sequential logic (flip-flops, water level control, wood sorting machine), and sequential logic with time constraints (cargo lift).

Specification in the form of annotations is transparent for compilers which do not support such assertions. Therefore for practical reasons, one general purpose IEC 61131-3 compiler may be used for verification, and another one, dedicated to particular hardware, applied for implementation. The presented compiler may be also extended to perform dynamic run-time verification, as provided by JML with some supporting tools.

Future work will concentrate on direct conversion from ST language into Why code, without limitation of available types. The types constrained by Caduceus conversion will be transformed to suit types provided by Coq or by external libraries. Naturally, direct conversion will require some additional algorithms to construct proofs of the lemmas.

References

[1] E. Tisserant, L. Bessard, and M. de Sousa, "An open source IEC 61131-3 integrated development environment," in *5th Int. Conf. Industrial Informatics*. Piscataway, NJ, USA, 2007.

[2] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, p. 40–51, 1992.

[3] J.-C. Filliâtre, T. Hubert, and C. Marché, "The

Caduceus verification tool for C programs," [online] http://caduceus.lri.fr, 2008.

[4] J.-C. Filliâtre, "The Why verification tool. tutorial and reference manual," [online] http://www.lri.fr, 2010.

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

[6] G. T. Leavens, A. L. Baker, and C. Ruby, *JML: a Notation for Detailed Design*, ser. Behavioral Specifications of Businesses and Systems, 1999.

[7] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Inc., 1976.

[8] J. Sadolewski, *Verification of complex programs for control systems*, ser. Methods of producing and applying real time systems. Wydawnictwa Komunikacji i Łączności, 2010, (in Polish).

[9] ——, "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).

[10] R. Bornat, "Proving pointer programs in Hoare logic," in *Mathematics of Program Construction*. Springer-Verlag. London, 2000, pp. 102–126.

[11] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Open environment for programming small controllers according to IEC 61131-3 standard," *Scalable Computing Practice and Experience*, vol. 10, no. 3, pp. 325–336, 2009.

[12] K. D. Cooper and L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, San Francisco, 2003.