UDK 004.4'42

Model Checking Programs in Process-Oriented IEC 61131-3 Structured Text

Garanina N.O. (Institute of Automation and Electrometry SB RAS)

Staroletov S.M. (Institute of Automation and Electrometry SB RAS)

Zyubin V.E. (Institute of Automation and Electrometry SB RAS)

Anureev I.S. (Institute of Automation and Electrometry SB RAS)

The process-oriented programming is a paradigm based on the process concept where each process is a concurrent finite state machine inside. The paradigm is intended for PLC (programmable logic controllers) developers to write Industry 4.0-enabled software. The poST language is a promising process-oriented extension of the IEC 61131-3 Structured Text (ST) language designed to provide a conceptual consistency of the PLC source code with technological description of the process under control. This language combines the advantages of FSM-based programming with the standard syntax of the ST language. We propose transformational semantics of poST providing rules for translation of poST language statements to Promela – the input language of the SPIN model checker. Following these semantic rules, our Xtext-based translator outputs a Promela model for the poST program. Our contribution is a method for automatic generation of the Promela code from poST control programs. The resulting Promela program is ready to be verified with SPIN model checker against linear temporal logic requirements to the source poST program.

Keywords: control software, model checking, process-oriented programming, LTL, SPIN, Structured Text

1. Introduction

With the significant progress in formal methods for proving program models, the question of their applicability to real software remains. It can be stated that the application of such methods to general programs does not seem appropriate due to the laboriousness of translating language constructs with ambiguous semantics, a large state space, and the uncertainty or inexpressibility of requirements.

In the field of control software, the situation is twofold: on the one hand, the cost of an error is high, and control programs need formalization and proving correctness with respect to formalized requirements, on the other hand, programs for PLC (Programmable Logic Controllers) designed to implement industrial process control algorithms are concise, the languages themselves contain a small number of constructions, the requirements are expressible as there are few key control variables. These features of PLC languages makes applicability of formal methods more promising.

Currently, the languages of the IEC 61131-3 standard [11] are used for programmable industrial controllers. They differ in program representation: in text, ladder diagrams, function block diagrams and instruction lists. In this work, we focus on a textual representation in Structured Text (ST) language. This language is Pascal-like and has a simple syntax with some features (e.g. timers, intervals) specific to PLC programs [10, 11]. However, most control programs are cyclical and state dependent, resulting in a large number of switch statements.

Exploiting this feature and presenting developers with convenient means of operating with states and transitions, we introduce the concept of process as the main logical entity of the program. We call programming according to this methodology process-oriented, and the language that adds such syntactic sugar is called process-oriented programming language. Note, that it is reasonable to extend standardized ST language, since the development, simulation and firmware creation environments for real controllers are already adapted for this language. The poST (process-oriented Structured Text) language is one of such extensions, which is compiled into the ST language. The source-to-source compiling is also called transpiling, and nowadays it is a convenient means to switch between various languages [8].

The use of formal verification methods for PLC programs is justified primarily by the fact that such programs can work with expensive equipment of some plant and the incorrect behavior of the program can lead to both financial losses and serious consequences for the environment and plant personnel. Therefore, it is necessary to provide as many means as possible for static checks of such programs in the early stages of their production.

In this paper, we present an automatic translation of poST programs into programs in the input language of a program verifier which use the model checking method. A review of the practices of applying the model checking for such systems is presented in the paper [9].

In our series of previous works, we used the SPIN verification tool [7] coming with the Promela input language (Protocol Meta-Language), which corresponds to the CSP approach [6], that is, it has the ability to describe systems as interacting processes. This is close to process-oriented, but not quite the same: the language does not work with process states by language means, although states, processes and switching between them can be represented in the form of conditional structures, additional variables, and passing the progress through channels.

In this paper, we describe, the code transformation from poST to Promela as well as the issues of formalizing and implementing such transpiling. The implementation is a solution based on the Eclipse Xtext parser tool in Java and Xtend.

The rest of the paper has the following structure. In Section 2, we consider features of the poST and Promela languages. Section 3 defines rules for the transformational semantics for the poST language and the conclusion is given in Section 4.

This work is supported by the Russian Ministry of Education and Science, project no. 122031600173-8.

2. The poST and Promela languages

The poST language is a novel process-oriented extension of the IEC 61131-3 Structured Text (ST) language which provides conceptual consistency of the PLC source code with technological description of the process under control. The language combines the advantages of FSM based programming with the conventional syntax of the ST language which would facilitate its adoption.

Inspired by a general PLC scan cycle, a poST program includes processes whose activity is orchestrated into a cycle in order of their appearance in the program code. This scheme expects PLC models that abstract from a scan cycle time [5]. Each process is specified by an ordered set of process states. To describe process states, we use standard ST constructs (variable declarations, control-flow statements, etc.) and specific process-oriented features: process states statements (`START/STOP PROCESS`, `SET NEXT` and other) and timeout statements. The semantics of the poST language assumes an automatic implementation of low-level constructions for mapping I/O signals to program variables, process states, timeout statements, and cyclic time-triggered control. The grammar of the poST language in the Xtext format is available in the repository [15].

Promela language [13] is used to describe parallel communicating processes based on the CSP formalism [6]. Promela program consists of parallel processes communicating through channels or shared variables. The execution of a set of Promela parallel processes exploits the interleaving semantics. Interleaving can be bounded by `atomic` and `d_step` statements, which permit interruption of specified sequence of process actions. The Promela language includes blocking control-flow statements `if` and `do`, unlike standard non-blocking poST control-flow statements. Promela model can be verified by model checker SPIN [7] against LTL requirements, hence it assumes only finite types for model variables. This causes the main difficulty of translation poST programs to Promela models because poST types for real numbers cannot be directly translated to Promela types.

The poST language specifies control algorithms which interact with some environment. Hence, we suppose that the source poST code may contain the control program and the controlled object program. Translating this code, we construct the Promela model that corresponds to the order of processes activation, the structure of process states, timeout management, and variable types up to abstracting from real types.

3. poST Language Transformation Rules

In this section, we describe representative translations from poST language to Promela language. We start from common constructs as types, variables, standard operations and control-flow statements (Sections A and B). Further, we present special process-oriented features, namely, a scan cycle, process states, timeouts, and special LTL forms for requirements. The complete formal transformational semantics for poST is located in the repository [14].

3.1. Naming, Types, Declarations, and Operations

poST has global, program, and process namespaces, while Promela only has global and process namespaces. To avoid name collisions, we form the full entity names in the Promela model of a poST program taking care of (1) the entity name; (2) the entity kind (process, variable, etc.); and (3) the name of the entity owner. To improve program readability, we use the default naming mode which takes into account just entity kinds enriched with counters if there are several equal names in the global Promela namespace.

Variable types are translated trivially in most cases. We consider poST programs without real variables, since Promela does not have real types, and data type abstraction is beyond the scope of this paper. Translation of the `TIME` type is discussed below when describing the translation of the `TIMEOUT` expression.

Following this naming and type policy, all renamed poST variables are declared in Promela model as global variables, and poST constants are trivially translated using Promela directive `#define`.

Promela includes the same operations as poST except exponentiation which can be modeled bit-wise shift Promela operations.

3.2. Control-Flow Statements

In Table 1 we give translation two kinds of poST control-flow statements to Promela code. Let *code′* be the Promela image of the poST *code* made by our translation algorithm. We use the Promela `else` branch and the `skip` statement in the translation of the poST `IF` statement because the Promela `if` statement is blocking and the process cannot proceed further if condition *cond* is false. Following the Promela semantics, `else` branch is chosen when no other condition in `if` statement is satisfiable. The `skip` statement in this branch just does nothing. The poST `CASE` statement is translated in the similar way. The Promela `do` statement is also blocking, hence in modeling poST `DO` statement we need the `else` branch with the `break` statement to model the termination of a loop in Promela.

Table 1

Control-Flow statements

poST	**Promela**	**poST**	**Promela**
`IF` *cond* `THEN` *body* `END_IF`	`if ::` *cond′* `-> {` *body′* `}` `:: else -> skip;` `fi;`	`WHILE` *cond* `DO` *body* `END_WHILE`	`do ::` *cond′* `-> {` *body′* `}` `:: else -> break;` `od;`

3.3. Process-oriented features

A control system is specified in poST by a set of poST programs. Hence, we should translate several poST programs into a single Promela model. A particular poST program consists of processes which activated cyclically one after another. Several poST programs run in a joint scan cycle in the order of their appearance in the source code. In Promela, we model this cyclic activation sequentially passing *a move message* from a process to a process through a Promela channel in the order in which poST processes appear in the source programs. Promela channels support blocking reads and writes. We use the channel `turn` of capacity 1 to pass nicknames of processes in move messages. Every process is initially blocked until reading its nickname from this channel. After execution of its body, a process pushes the nickname for the next process into the channel to pass the move. The Promela process for the last poST process turns the move to the service process `Gremlin` which represents the non-deterministic environment generating input values. This process is the first process of the resulting Promela model. Due to sequential activation semantics of poST programs, we use Promela `atomic` statement for the body of every translated poST process. This statement permits interleaving execution of other processes and significantly simplifies verification. The left block of Table 2 gives translation of the upper process structure of poST programs to Promela.

Table 2

Process-oriented features

poST	**Promela**	**poST**	**Promela**
PROGRAM $prog_1$	mtype : P =	PROCESS id	mtype:S_id' =
PROCESS id_{11}	{ p_id'_{11}, ... }	STATE s_1	{s_s'_1,...,s_$Error'$}
$body_{11}$	chan turn=[1]of{mtype:P}	$body_1$	mtype:S_id'
END_PROCESS		END_STATE	c_id' = s_$Stop'$;
...	active proctype id'_{11}(){	...	active proctype id'(){
END_PROGRAM	do :: turn ? p_id'_{11} ->	STATE s_n	do :: turn ? p_id'->
...	atomic {	$body_n$	atomic {
PROGRAM $prog_m$	$body'_{11}$;	END_STATE	if
PROCESS id_{1m}	turn ! p_id'_{21};}	END_PROCESS	:: c_id'==s_s'_1->
$body_{1m}$	od;		{ $body'_1$ }
END_PROCESS	}		...
...	...		:: else ->skip;
END_PROGRAM	active proctype id'_{nm}(){		fi;
	...		turn ! nx_pr;}
	}		od;
			}

A body of a poST process consists of *states*, including special states of inactivity `STOP` and `ERROR`. At every iteration of the scan cycle, the poST process executes the code corresponding to some of its states except states `STOP` and `ERROR` when it does nothing. In Promela, we use a special state counter for every translated process to keep the name of the current state. At the start of a poST program, its first declared process is in its first state and all other processes are in state STOP. The right block of Table 2 gives translation to Promela for a particular poST process.

poST processes can check an activity status of other processes with statement `ACTIVE` and `INACTIVE`. Also each process can force itself or anther process to change its state with statements `RESTART`, `STOP`, `START PROCESS`, `STOP PROCESS`, and others. These poST statements are translated trivially to Promela if the goal state do not include `TIMEOUT` statement. Please, see examples in the right block of Table 3.

poST process states can have a `TIMEOUT` block as the last state block. Instructions of this block is executed after the time specified in the timeout has elapsed since the process entered this state. To model this behaviour in Promela, we introduce a counting time variable — one per each process that contains states with `TIMEOUT`.

To reduce the size of the Promela model, we provide the following optimisation for model time counters. First, we use the value of poST scan cycle (`INTERVAL`) to reduce all timeout values to the nearest multiple of this interval. Second, we divide all timeout values by their greatest common divisor. In addition, we choose the minimum sufficient size *nb* of the unsigned type for the time counters. For example, we add one time counter with a size of 4 bits if a resulting Promela process has two states with timeouts that count 5 (101b) and 9 (1001b) units of time.

At every scan cycle, if a process in a state with a timeout, its time counter is incremented. A process counter sets to zero when (1) the process resets the timeout; (2) the process moves to other state; and (3) timeout happens. Following poST semantics, we use the `>` sign in the Promela timeout `if` statement because the execution of the timeout block begins at the next cycle after the timeout time has passed. We give the representative model constructs for timeouts in the left block of Table 3.

Table 3

State and Timeout Statements

poST	Promela	poST	Promela
IF (PROCESS id	if	PROCESS id_1	unsigned t_id'_1 : nb
INACTIVE)	:: c_id' == s_$Stop'$ ||	STATE s_1	active proctype id'_1()
THEN $body$	c_id' == s_Err' ->	$body_1$	{ ...
END_IF	{ $body'$ }	TIMEOUT T#tt	:: c_id'_1 == s_s'_1 ->{
PROCESS id	:: else -> skip; fi;	THEN $body_t$	$body'_1$
STATE s_1	active proctype id'(){	END_TIMEOUT	if
$body_1$	do :: turn ? p_id'->	END_STATE	:: t_id'_1 > tt' ->
SET NEXT	atomic {	...	$body'_t$
END_STATE	if	END_PROCESS	:: else ->t_id'_1++;
STATE s_2	:: c_id'==s_s'_1 ->		fi;}
$body_2$	{ $body'_1$		... }
END_STATE	c_id' = s_s'_2; }	PROCESS id_2	active proctype id'_2()
...	:: c_id'==s_s'_2 ->	STATE s_2	{ ...
END_PROCESS	{ $body'_2$ }	$body_2$	:: c_id'_2 == s_s'_2 ->{
	...	START	$body'_2$
	:: else -> skip;	PROCESS id_1	c_id'_1 = s'_1;
	fi;	END_STATE	t_id'_1 = 0; }
	turn ! $next$;}	...	...
	od; }	END_PROCESS	}

Table 4

The Promela model for poST programs

poST	**Promela**
`PROGRAM` $prog_1$	$Var\_Declaration'_1$
$Var\_Declaration_1$	`...`
`PROCESS` $name_{11}$	$Var\_Declaration'_m$
`...`	$Service\_Declarations$
`PROCESS` $name_{n1}$	`init{ turn !  p_Gremlin; }`
`END_PROGRAM`	`active proctype Gremlin(){...}`
`...`	`active proctype OutInput(){...}`
`PROGRAM` $prog_m$	`active proctype BOC(){...}`
$Var\_Declaration_m$	`active proctype` $name'_{11}$`(){...}`
`PROCESS` $name_{1m}$	`...`
`...`	`active proctype` $name'_{nm}$`(){`
`PROCESS` $name_{nm}$	`do ::  turn ?  p_`$name'_{nm}$ `->`
`END_PROGRAM`	`atomic { ...`
	`turn !  p_Gremlin; }`
	`od;`
	`}`

3.4. Overall Promela model for poST programs

In general, our translation algorithm takes as input several poST program that describe a control system in one file. This control system may include the control algorithm and its environment: controlled and uncontrolled objects. In Table 4, we give the resulting Promela model, which includes three service processes and processes corresponding the source poST processes. Non-deterministic service process `Gremling` models uncontrolled object. Service process `OutInput` cares about proper corresponding of inputs and outputs of programs composing the source program. Service process `BOC` captures the beginning of the scan cycle for checking requirements. Activity of these processes forms a scan cycle by passing move messages between them starting from the Gremling process modeling initial inputs from uncontrolled object and ending with the last source poST process which pass move message to Gremlin again. Inside the cycle, the processes activity is ordered as described in Table 2.

4. Discussion and Conclusion

In this paper, we have considered approaches to formalization and implementation of the source-to-source compiling (transpiling) process. For PLC programming languages with a small number of statements, such a process is justified, since all constructs of the input language can be converted into language constructs for which methods of formal program verification have already been well developed (in this case, we use the model checking method and SPIN verification system). This allows us to make examples of PLC programs intended for verification and not to write repetitive code constructs in Promela related to the semantics of process switching, transitioning through states, exchanging variable values and generating an impulse for checking program properties w.r.t. scan cycles. The project is publicly available in our repository [12]. Among the shortcomings of the current approach, we note the incomplete support for data types, in particular, real variables are not supported. This can be eliminated by implementing libraries of both fixed-point and floating-point arithmetic, however, this will entail a huge number of states and the verification of resulting programs cannot be done without manual abstraction methods. Also, library functions are not implemented.

As a result, with the development of the considered transpiler, we are approaching the development of a toolchain for writing verifiable programs in the process-oriented style.

References

1. Zyubin V. E., Rozov A. S., Anureev I. S., Garanina N. O. and Vyatkin V. poST: A Process-Oriented Extension of the IEC 61131-3 Structured Text Language // IEEE Access 2022. Vol. 10, P. 35238–35250.
2. Ponomarenko A. A., Garanina N. O., Staroletov S. M., and Zyubin V. E. Towards the Translation of Reflex Programs to Promela: Model Checking Wheelchair Lift Software // Proc. of IEEE 22nd Intern. Conf. of Young Professionals in Electron Devices and Materials (EDM). 2021. P. 493–498.
3. Anureev I. S., Garanina N. O., Liakh T. and Rozov A. S., and Schulte H. and ZyubinV. E. Towards safe cyber-physical systems: the Reflex language and its transformational semantics // Proc. of 2019 Intern. Siberian Conf. on Control and Communications (SIBCON). 2019. P. 1–6.
4. Clarke E.M., Henzinger T. A., Veith H., and Bloem R. Handbook of model checking. Springer, 2018. 1210 p.
5. Mader A. A Classification of PLC Models and Applications // Discrete Event Systems 2000. Vol. 596. P. 239–246.
6. Hoare C. A. R. Communicating sequential processes. Prentice-Hall: 1985.
7. Holzmann G. J. The Spin Model Checker, Primer and Reference Manual. Addison-Wesley: 2003.
8. Schneider L. and Schultes D. Evaluating Swift-to-Kotlin and Kotlin-to-Swift transpilers //

Proc. of the 9th IEEE/ACM Int. Conf. on Mobile Software Engineering and Systems. 2022. P. 102–106.

9. Ovatman T. An overview of model checking practices on verification of PLC software // Software & Systems Modeling. 2016. Vol 4, No 15. P. 937–960.
10. Antonsen T. M. PLC Controls with Structured Text (ST), V3: IEC 61131-3 and best practice ST programming. BoD–Books on Demand, 2020.
11. IEC 61131-3:2013. Programmable controllers - Part 3: Programming languages. 2013. URL: https://webstore.iec.ch/publication/4552
12. Translator-poST-Promela. 2023.
URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev
13. Promela grammar. URL: http://spinroot.com/spin/Man/grammar.html
14. URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/semantics/transformation.pdf, 2023.
15. URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/poST_grammar.xtext, 2022.
16. URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/samples/, 2023.
17. Xtext is a framework for development of programming languages and domain-specific languages. 2022. URL: http://eclipse.org/Xtext
18. Xtend is a flexible and expressive dialect of Java. 2021. URL: https://www.eclipse.org/xtend/