G4LTL-ST: Automatic Generation of PLC Programs

Chih-Hong Cheng[1], Chung-Hao Huang[2], Harald Ruess[3], and Stefan Stattelmann[1]

[1] ABB Corporate Research, Ladenburg, Germany
[2] Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
[3] fortiss - An-Institut Technische Universität München, München, Germany

Abstract. G4LTL-ST automatically synthesizes control code for industrial Programmable Logic Controls (PLC) from timed behavioral specifications of input-output signals. These specifications are expressed in a linear temporal logic (LTL) extended with non-linear arithmetic constraints and timing constraints on signals. G4LTL-ST generates code in IEC 61131-3-compatible *Structured Text*, which is compiled into executable code for a large number of industrial field-level devices. The synthesis algorithm of G4LTL-ST implements pseudo-Boolean abstraction of data constraints and the compilation of timing constraints into LTL, together with a counterstrategy-guided abstraction-refinement synthesis loop. Since temporal logic specifications are notoriously difficult to use in practice, G4LTL-ST supports engineers in specifying realizable control problems by suggesting suitable restrictions on the behavior of the control environment from failed synthesis attempts.

Keywords: industrial automation, synthesis, theory combination, assumption generation.

1 Overview

Programmable Logic Controllers (PLC) are ubiquitous in the manufacturing and processing industries for realizing real-time controls with stringent dependability and safety requirements. A PLC is designed to read digital and analog inputs from various sensors and other PLCs, execute a user-defined program, and write the resulting digital and analog output values to various output elements including hydraulic and pneumatic actuators or indication lamps. The time it takes to complete such a scan cycle typically ranges in the milliseconds.

The languages defined in the IEC 61131-3 norm are the industry standard for programming PLCs [1]. Programming in these rather low-level languages can be very inefficient, and yields inflexible controls which are difficult to maintain and arduous to port. Moreover, industry is increasingly moving towards more flexible and modular production systems, where the control software is required to adapt to frequent specification changes [2].

With this motivation in mind, we developed the synthesis engine G4LTL-ST for generating IEC 61131-3-compatible Structured Text programs from behavioral specifications. Specifications of industrial control problems are expressed in a suitable extension of linear temporal logic (LTL) [14]. The well-known LTL operators **G**, **F**, **U**, and **X** denote "always", "eventually", "(strong) until", and "next"s relations over linear execution traces. In addition to vanilla LTL, specifications in G4LTL-ST may also include

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 541–549, 2014.

1 Input: $\mathsf{x}, \mathsf{y} \in [0,4] \cap \mathbb{R}$, $\mathsf{err} \in \mathbb{B}$, Output: $\mathsf{grant1}, \mathsf{grant2}, \mathsf{light} \in \mathbb{B}$, Period: 50ms
2
3 $\mathbf{G}\,(\mathsf{x} + \mathsf{y} > 3 \rightarrow \mathbf{X}\,\mathsf{grant1})$
4 $\mathbf{G}\,(\mathsf{x}^2 + \mathsf{y}^2 < \frac{7}{2} \rightarrow \mathbf{X}\,\mathsf{grant2})$
5 $\mathbf{G}\,(\neg(\mathsf{grant1} \wedge \mathsf{grant2}))$
6 $\mathbf{G}\,(\mathsf{err} \rightarrow \mathsf{10sec(light)})$
7 $\mathbf{G}\,((\mathbf{G}\neg\mathsf{err}) \rightarrow (\mathbf{FG}\neg\mathsf{light}))$

Fig. 1. Linear temporal logic specification with arithmetic constraints and a timer

- non-linear arithmetic constraints for specifying non-linear constraints on real-valued inputs;
- timing constraints based on timer constructs specified in IEC 61131-3.

A timing constraint of the form 10sec(light), for example, specifies that the light signal is on for 10 seconds. Moreover, the semantics of temporal specifications in G4LTL-ST is slightly different from the standard semantics as used in model checking, since the execution model of PLCs is based on the concept of *Mealy machines*. Initial values for output signals are therefore undefined, and the synthesis engine of G4LTL-ST assumes that the environment of the controller makes the first move by setting the inputs.

Consider, for example, the PLC specification in Figure 1 with a specified scan cycle time of 50ms (line 1). The input variables $\mathsf{x}, \mathsf{y}, \mathsf{err}$ store bounded input and sensor values, and output values are available at the end of each scan cycle at grant1, grant2, and light (line 1). According to the specification in line 6, the output light must be on for at least 10 seconds whenever an error occurs, that is, input signal err is raised. Line 7 requires that if err no longer appears, then eventually the light signal is always off. The transition-style LTL specifications 3 and 4 in Figure 1 require setting grant1 (resp. grant2) to true in the next cycle whenever the condition $\mathsf{x} + \mathsf{y} > 3$ (resp. $\mathsf{x}^2 + \mathsf{y}^2 < \frac{7}{2}$) holds. Finally, grant1 and grant2 are supposed to be mutually exclusive (line 5).

The synthesis engine of G4LTL-ST builds on top of traditional LTL synthesis techniques [13,9,15,4] which view the synthesis problem as a game between the (sensor) environment and the controller. The moves of the environment in these games are determined by setting the input variables, and the controller reacts by setting output variables accordingly. The controller wins if the resulting input-output traces satisfy the given specification. Notably, arithmetic constraints and timers are viewed as theories and thus abstracted into a pseudo-Boolean LTL formula. This enables G4LTL-ST to utilize CEGAR-like [8,12,10] techniques for successively constraining the capabilities of the control environment.

Since specifications in linear temporal logic are often notoriously difficult to use in practice, G4LTL-ST diagnoses unrealizable specifications and suggests additional *assumptions* for making the controller synthesis problem realizable. The key hypothesis underlying this approach is that this kind of feedback is more useful for the engineer compared to, say, counter strategies. The assumption generation of G4LTL-ST uses built-in templates and heuristics for estimating the importance and for ordering the generated assumptions accordingly.

Synthesis of control software, in particular, has been recognized as a key *Industrie 4.0* technology for realizing flexible and modular controls (see, for example, [3],

Table 1. Real-time specification patterns and their encodings

Real-time specification pattern	Encoding in LTL
Whenever a, then b for t seconds	$\mathbf{G}\,(\mathsf{a} \rightarrow (\mathsf{t1.start} \wedge \mathsf{b} \wedge \mathbf{X}(\mathsf{b}\,\mathbf{U}\,\mathsf{t1.expire})))$
Whenever a continues for more than t seconds, then b	$(\mathsf{a} \leftrightarrow \mathsf{t1.start}) \wedge \mathbf{G}(\neg(\mathsf{a} \wedge \mathbf{X}\,\mathsf{a}) \leftrightarrow \mathbf{X}\,\mathsf{t1.start})$ $\wedge\, \mathbf{G}(\mathsf{t1.expire} \rightarrow \mathsf{b})$
Whenever a, then b, until c for more than t seconds	$\mathbf{G}(\mathsf{a} \leftrightarrow \mathsf{t1.start}) \wedge \mathbf{G}(\neg(\mathsf{c} \wedge \mathbf{X}\,\mathsf{c}) \leftrightarrow \mathbf{X}\,\mathsf{t1.start})$ $\wedge\, \mathbf{G}\,(\mathsf{a} \rightarrow (\mathsf{b} \wedge \mathbf{X}((\mathsf{b}\,\mathbf{U}\,\mathsf{t1.expire})) \vee \mathbf{G}\neg\mathsf{t1.expire}))$

RE-2 on page 44). The synthesis engine G4LTL-ST is planned to be an integral part of a complete development tool chain towards meeting these challenges. G4LTL-ST is written in Java and is available (under the GPLv3 open source license) at

`http://www.sourceforge.net/projects/g4ltl/files/beta`

In the following we provide an overview of the main features of G4LTL-ST including Pseudo-Boolean abstractions of timing constraints, the abstraction-refinement synthesis loop underlying G4LTL-ST and its implementation, and, finally, the template-based generation for suggesting new constraints of the behavior of the environment for making the control synthesis problem realizable. These features of G4LTL-ST are usually only illustrated by means of examples, but the initiated reader should be able to fill in missing technical details.

2 Timing Abstractions

The timing constraint in Figure 1 with its 10 seconds time-out may be encoded in LTL by associating each discrete step with a 50ms time delay. Notice, however, that up to 200 consecutive **X** operators are needed for encoding this simple example.

Instead we propose a more efficient translation, based on standard IEC 61131-3 timing constructs, for realizing timing specifications. Consider, for example, the timed specification $\mathbf{G}\,(\mathsf{err} \rightarrow \mathsf{10sec}(\mathsf{light}))$. In a first step, fresh variables t1.start and t1.expire are introduced, where t1 is a *timer variable* of type TON in IEC 61131-3. The additional output variable t1.start starts the timer t1, and the additional input variable t1.expire receives a time-out signal from t1 ten seconds after this timer has been started. Now, the timing specification $\mathbf{G}\,(\mathsf{err} \rightarrow \mathsf{10sec}(\mathsf{light}))$ is rewritten as an LTL specification for a function block in the context of a timer.

$$\mathbf{G}\,(\mathsf{t1.start} \rightarrow \mathbf{X}\,\mathbf{F}\,\mathsf{t1.expire}) \rightarrow \mathbf{G}\,(\mathsf{err} \rightarrow (\mathsf{t1.start} \wedge \mathsf{light} \wedge \mathbf{X}(\mathsf{light}\,\mathbf{U}\,\mathsf{t1.expire}))$$

The antecedent formula ensures that the expire signal is eventually provided by the timing block of the environment. Since no provision is being made that there is a time-out exactly after 10 seconds, however, the precise expected behavior of the time-out environment is over-approximated.

It is straightforward to generate PLC code using timing function blocks from winning strategies of the controller (see below for the automatically generated code). Whenever t1.start is set to true the instruction t1(IN:=0, PT:=TIME#10s) is generated for starting the timer t1. Instructions that set t1.start to false is ignored based on the underlying

semantics of timers. Finally, time-out signals t1.expire are simply replaced with the variable t1.Q of the IEC 61131-3 timing construct.

```
FUNCTION_BLOCK FB_G4LTL
VAR_INPUT     error: BOOL;      END_VAR
VAR_OUTPUT    light: BOOL;      END_VAR
VAR           cstate : INT := 0;   t1: TON;    END_VAR
VAR CONST     T1_VALUE  : TIME := TIME#10s;     END_VAR

CASE cstate OF
   0: IF ((error = TRUE) AND  (TRUE)) THEN cstate := 12; light := TRUE; t1(IN:=0, PT:=T1_VALUE);
       ELSIF ((error = FALSE) AND  (TRUE)) THEN cstate := 6; light := FALSE;
       END_IF;
 43: IF ((error = TRUE) AND  (TRUE)) THEN cstate := 12; light := TRUE; t1(IN:=0, PT:=T1_VALUE);
       ELSIF ((error = FALSE) AND  (TRUE)) THEN cstate := 43; light := FALSE;
       END_IF;
   6: IF ((error = TRUE) AND  (TRUE)) THEN cstate := 12; light := TRUE; t1(IN:=0, PT:=T1_VALUE);
       ELSIF ((error = FALSE) AND  (TRUE)) THEN cstate := 6; light := FALSE;
       END_IF;
396: IF ((error = TRUE) AND  (TRUE)) THEN cstate := 12; light := TRUE; t1(IN:=0, PT:=T1_VALUE);
       ELSIF ((error = FALSE) AND  (t1.Q = FALSE)) THEN cstate := 396; light := TRUE;
       ELSIF ((error = FALSE) AND  (t1.Q = TRUE)) THEN cstate := 43; light := FALSE;
       END_IF;
 81: IF ((error = TRUE) AND  ( TRUE )) THEN cstate := 12; light := TRUE; t1(IN:=0, PT:=T1_VALUE);
       ELSIF ((error = FALSE) AND  (t1.Q = FALSE)) THEN cstate := 396; light := TRUE;
       ELSIF ((error = FALSE) AND  (t1.Q = TRUE)) THEN cstate := 43; light := FALSE;
       END_IF;
12: IF ((error = TRUE) AND  (TRUE)) THEN cstate := 12; light := TRUE; t1(IN:=0, PT:=T1_VALUE);
      ELSIF ((error = FALSE) AND  (t1.Q = FALSE)) THEN cstate := 81; light := TRUE;
      ELSIF ((error = FALSE) AND  (t1.Q = TRUE)) THEN cstate := 6; light := FALSE;
      END_IF;
END_CASE;
END_FUNCTION_BLOCK
```

In Table 1 we describe some frequently encountered specification patterns and their translations using IEC 61131-3-like timing constructs. Each of these patterns requires the introduction of a fresh timer variable t1 together with the assumption $\mathbf{G}\,(\mathsf{t1.start} \rightarrow \mathbf{X}\,\mathbf{F}\,\mathsf{t1.expire})$ on the environment providing time-outs. These specification patterns, however, are not part of the G4LTL-ST input language, since there is no special support in the synthesis engine for these language constructs, and G4LTL-ST is intended to be used in integrated development frameworks, which usually come with their own specification languages.

3 Abstraction-Refinement Synthesis Loop

The input to the synthesis engine of G4LTL-ST are LTL formulas with non-linear arithmetic constraints with bounded real (or rational) variables, and the workflow of this engine is depicted in Figure 2. Notice, however, that the abstraction-refinement loop in Figure 2 is more general in that it works for any decidable theory Th.

In a preliminary step Abstract simply replaces arithmetic constraints on the inputs with fresh Boolean input variables. The resulting specification therefore is (like the timer abstraction in Section 2) an over-approximation of the behavior of the environment. In our running example in Figure 1 (ignoring line 6, 7), Abstract creates two fresh Boolean variables, say req1 and req2, for the two input constraints $\mathsf{x}+\mathsf{y} > 3$ and $\mathsf{x}^2 + \mathsf{y}^2 < \frac{7}{2}$ to obtain the pseudo-Boolean specification

$$\mathbf{G}(\mathsf{req1} \rightarrow \mathbf{X}\,\mathsf{grant1}) \wedge \mathbf{G}(\mathsf{req2} \rightarrow \mathbf{X}\,\mathsf{grant2}) \wedge \mathbf{G}(\neg(\mathsf{grant1} \wedge \mathsf{grant2})) \quad (1)$$

Clearly, this pseudo-Boolean specification with input variables req1 and req2 over-approximates the behavior of the environment, since it does not account for inter-relationships of the arithmetic input constraints.

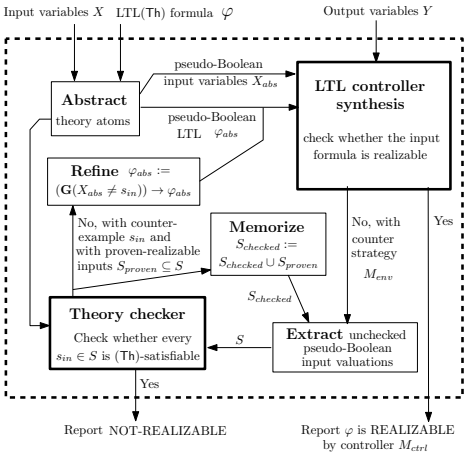

Fig. 2. Abstraction-refinement synthesis loop

In the next step, LTL controller synthesis checks whether or not the pseudo-Boolean LTL formula generated by Abstract is realizable. If the engine is able to realize a winning strategy for the control, say M_{ctrl}, then a controller is synthesized from this strategy. Otherwise, a candidate counter-strategy, say M_{env}, for defeating the controller's purpose is generated.

The pseudo-Boolean specification (1), for example, is unrealizable. A candidate counter-strategy for the environment is given by only using the input $(\mathsf{true}, \mathsf{true})$, since, in violation of the mutual exclusion condition (1), the controller is forced to subsequently set both grant1 and grant2 .

The Extract module extracts candidate counter-strategies with fewer pseudo-Boolean input valuations (via a greedy-based method) whose validity are not proven at the theory level. Consequently, the Extract module generates a candidate counter-strategy that only uses $(\mathsf{req1}, \mathsf{req2}) = (\mathsf{true}, \mathsf{true})$ and the input valuations $S = \{(\mathsf{true}, \mathsf{true})\}$ are passed to the Theory Checker.

A candidate counter-strategy is a genuine counter-strategy only if all pseudo-Boolean input patterns are satisfiable at the theory level; in these cases the environment wins and Theory Checker reports the un-realizability of the control problem. In our running example, however, the input $(\mathsf{true}, \mathsf{true})$ is not satisfiable at the theory level, since the conjunction of the input constraints $\mathsf{x} + \mathsf{y} > 3$ and $\mathsf{x}^2 + \mathsf{y}^2 < \frac{7}{2}$ is unsatisfiable for $\mathsf{x}, \mathsf{y} \in [0, 4]$. G4LTL-ST uses the JBernstein [5] verification engine for discharging quantifier-free verification conditions involving non-linear real arithmetic. In order to avoid repeated processing at the theory level, all satisfiable inputs are memorized.

Unsatisfiable input combinations s_{in} are excluded by Refine. In our running example, the formula $\mathbf{G}(\neg(\mathsf{req1} \wedge \mathsf{req2}))$ is added as a new assumption on the environment, since the input pair $(\mathsf{true}, \mathsf{true})$ has been shown to be unsatisfiable.

$$\mathbf{G}(\neg(\mathsf{req1} \wedge \mathsf{req2})) \rightarrow (1) \tag{2}$$

In this way, Refine successively refines the over-approximation of the behavior of the environment. Running the LTL synthesis engine on the refined specification 2 yields a controller: if one of req1 ($\mathsf{x} + \mathsf{y} > 3$) and req2 ($\mathsf{x}^2 + \mathsf{y}^2 < \frac{7}{2}$) holds, the controller may grant the corresponding client in the next round, since req1 and req2 do not hold simultaneously.

Refinement of Timer Environments. The refinement of over-approximations of environmental behavior also works for the abstracted timer environments. Recall from Section 2 that the initial abstraction is given by $\mathbf{G}\,(\mathsf{t1.start} \rightarrow \mathbf{X}\,\mathbf{F}\,\mathsf{t1.expire})$. Assuming, for example, that t1.expire appears two iterations after t1.start in a candidate counter-strategy, one might strengthen this initial assumption with $\mathbf{G}\,(\mathsf{t1.start} \rightarrow ((\mathbf{X}\neg\mathsf{t1.expire}) \wedge (\mathbf{XX}\neg\mathsf{t1.expire}) \wedge (\mathbf{XXX}\,\mathbf{F}\,\mathsf{t1.expire})))$.

Constraints over input and output variables. Even though the current implementation of G4LTL-ST is restricted to specifications with arithmetic constraints on inputs only, the abstraction-refinement synthesis loop in Figure 2 works more generally for arithmetic constraints over input and output variables. Consider, for example, the specification $\mathbf{G}(\mathsf{x} > \mathsf{y} \rightarrow \mathbf{X}(\mathsf{z} > \mathsf{x}))$ with input variables $\mathsf{x}, \mathsf{y} \in [1,2] \cap \mathbb{R}$ and output variable $\mathsf{z} \in [0,5] \cap \mathbb{R}$. Abstraction yields a pseudo-Boolean specification $\mathbf{G}(\mathsf{in} \rightarrow \mathbf{X}\mathsf{out})$ with in, out fresh input variables for the constraints $\mathsf{x} > \mathsf{y}$ and $\mathsf{z} > \mathsf{x}$, respectively. Now, pseudo-Boolean LTL synthesis generates a candidate winning strategy M_{ctrl} for the controller, which simply sets the output out to be always true. The candidate controller M_{ctrl} is realizable if every pseudo-Boolean output assignment of M_{ctrl} is indeed satisfiable on the theory level. This condition amounts to demonstrating validity of the quantified formula $(\forall \mathsf{x} \in [1,2] \cap \mathbb{R})\,(\exists \mathsf{z} \in [0,5] \cap \mathbb{R})\,\mathsf{z} > \mathsf{x}$. Using the witness, say, 3 for the existentially quantified output variable z, a winning strategy for the controller is to always set the output z to 3, and the control synthesis problem therefore is realizable.

Otherwise, the candidate controller strategy is not realizable at the theory level, and, for pseudo-Boolean outputs, refinement due to un-realizability of the control synthesis problem is achieved by adding new constraints as *guarantees* to the pseudo-Boolean specification. For example the constraint $\mathbf{G}(\neg(\mathsf{grant1} \wedge \mathsf{grant2}))$ is added to the pseudo-Boolean specification, if pseudo-Boolean outputs grant1 and grant2 are mutually exclusive at the theory level.

In this way, the abstraction-refinement synthesis loop in Figure 2 may handle arbitrary theory constraints on input and output variables as long as corresponding verification conditions in a first-order theory with one quantifier-alternation can be decided. The implementation of G4LTL-ST could easily be extended in this direction by using, for examples the verification procedure for the exists-forall fragment of non-linear arithmetic as described in [7]. So far we have not yet encountered the need for this extensions, since the PLC case studies currently available to us are restricted to Boolean outputs.

4 Assumption Generation

An unrealizable control synthesis problem can often be made realizable by restricting the capabilities of the input environment in a suitable way. In our case studies from the manufacturing domain, for example, suitable restrictions on the arrival rate of workpieces were often helpful. G4LTL-ST supports the generation of these assumptions from a set of given templates. For example, instantiations of the template $\mathbf{G}(?\mathsf{a} \rightarrow (\mathbf{X}(\neg ?\mathsf{a}\,\mathbf{U}\,?\mathsf{b})))$, where ?a and ?b are meta-variables for inputs, disallows successive arrivals of an input signal ?a. For a pre-specified set of templates, G4LTL-ST performs a heuristic match of the meta-variables with input variables by analyzing possible ways of the environment to defeat the control specification.

Table 2. Experimental result based on the predefined unroll depth (3) of G4LTL-ST. Execution time annotated with "(comp)" denotes that the value is reported by the compositional synthesis engine.

# Example (synthesis)	Timer(T)/ Data(d)	lines of spec	Synthesis Time	Lines of ST
Ex1	T, D	9	1.598s (comp)	110
Ex2	T	13	0.691s	148
Ex3	T	9	0.303s	80
Ex4	T	13	21s	1374
Ex5	T	11	0.678s (comp)	210
Ex6	-	7	0.446s	41
Ex7	D	8	17s	43
Ex8	T	8	0.397s (comp)	653
Ex9	abstract D,T	3 + model (< 200 loc)	1.55s	550
Ex10	abstract D,T	3 + model (< 200 loc)	3.344s	229
Ex11	abstract D,T	3 + model (< 200 loc)	0.075s	105

# Example (Assup. gen)	# Learned Assump.	Time of Learning
Ex1	1	0.127s
Ex2	1	0.452s
Ex3	1	3.486s
Ex4	4	22s (DFS)
Ex5	1	2.107s
Ex6	1	1.046s
Ex7	1	0.154
Ex8	1	2.877
Ex9	1	8.318

The underlying LTL synthesis engine performs bounded unroll [15] of the negated property to safety games. Therefore, whenever the controller can not win the safety game, there exists an environment strategy which can be expanded as a finite tree, whose leaves are matched with the risk states of the game. Then, the following three steps are performed successively:

- *Extract* a longest path from the source to the leaf. Intuitively, this path represents a scenario where the controller endeavors to resist losing the game (without intentionally losing the game). For example, assume for such a longest path, that the environment uses $(\mathsf{a})(\neg\mathsf{a})(\neg\mathsf{a})(\neg\mathsf{a})$ to win the safety game.
- *Generalize* the longest path. Select from the set of templates one candidate which can *fit* the path in terms of generalization. For example, the path above may be generalized as $\mathbf{FG}\neg\mathsf{a}$. For every such template, the current implementation of G4LTL-ST defines a unique generalization function.
- *Resynthesize* the controller based on the newly introduced template. For example, given ϕ as the original specification, the new specification will be $(\neg\mathbf{FG}\neg\mathsf{a}) \rightarrow \phi$, which is equivalent to $(\mathbf{GF}\mathsf{a}) \rightarrow \phi$. Therefore, the path is generalized as an assumption stating that a should appear infinitely often.

If this process fails to synthesize a controller, then new assumptions are added to further constrain the environment behavior. When the number of total assumptions reaches a pre-defined threshold but no controller is generated, the engine stops and reports its inability to decide the given controller synthesis problem.

5 Outlook

The synthesis engine of G4LTL-ST has been evaluated on a number of simple automation examples extracted both from public sources and from ABB internal projects[1]. This synthesized function block can readily be passed to industry-standard PLC development tools for connecting function blocks with concrete field device signals inside the main program to demonstrate desired behavior. The evaluation results in Table 2 demonstrate that, despite the underlying complexity of the LTL synthesis, G4LTL-ST can still

[1] Due to space limits, short descriptions of the case studies have been moved to the extended version [6].

provide a practical alternative to the prevailing low-level encodings of PLC programs, whose block size are (commonly) within 1000 LOC. This is due to the fact that many modules are decomposed to only process a small amount of I/Os. For small sized I/Os, the abstraction of timers and data in G4LTL-ST together with counter-strategy-based lazy refinement are particularly effective in fighting the state explosion problem, since unnecessary unrolling (for timing) and bit-blasting (for data) are avoided. Data analysis is also effective when no precise (or imprecise) environment model is provided, as is commonly the case in industrial automation scenarios.

Mechanisms such as assumption generation are essential for the wide-spread deployment of G4LTL-ST in industry, since they provide feedback to the designer in the language of the problem domain. Extensive field tests, however, are needed for calibrating assumption generation in practice. Moreover, a targeted front-end language for high-level temporal specification of typical control problems for (networks of) PLCs needs to be developed [11].

References

1. International Electrotechnical Commission IEC 61131-3 Ed. 3.0: Programmable Controllers – Part 3: Programming languages. International Electrotechnical Commission, Geneva, Switzerland (2013)
2. Recommendations for implementing the strategic initiative - INDUSTRIE 4.0. German National Academy of Science and Engineering (AcaTech) (April 2013)
3. Die Deutsche Normungs-Roadmap - INDUSTRIE 4.0. DKE German Commission for Electrical, Electronic & Information Technologies (December 2013)
4. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.-F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 652–657. Springer, Heidelberg (2012)
5. Cheng, C.-H., Ruess, H., Shankar, N.: JBernstein - a validity checker for generalized polynomial constraints. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 656–661. Springer, Heidelberg (2013)
6. Cheng, C.-H., Huang, C.-H., Ruess, H., Stattlemann, S.: G4LTL-ST: Automated Generation of PLC Programs (full version). arXiv:1405.2409 (2014)
7. Cheng, C.-H., Shankar, N., Ruess, H., Bensalem, S.: EFSMT: A Logical Framework for Cyber-Physical Systems. arXiv:1306.3456 (2013)
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
9. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE (2006)
10. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-Guided Control. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 886–902. Springer, Heidelberg (2003)
11. Ljungkrantz, O., Akesson, K., Fabian, M., Yuan, C.: Formal Specification and Verification of Industrial Control Logic Components. IEEE Tran. on Automation Science and Engineering 7(3), 538–548 (2010)

12. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 36–50. Springer, Heidelberg (2005)
13. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM (1989)
14. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
15. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)