Structural Synthesis for GXW Specifications

Chih-Hong Cheng(✉), Yassine Hamza, and Harald Ruess

fortiss - An-Institut Technische Universität München,
Guerickestr. 25, 80805 Munich, Germany
{cheng,ruess}@fortiss.org,
yassine.hamza@in.tum.de

Abstract. We define the GXW fragment of linear temporal logic (LTL) as the basis for synthesizing embedded control software for safety-critical applications. Since GXW includes the use of a *weak-until* operator we are able to specify a number of diverse programmable logic control (PLC) problems, which we have compiled from industrial training sets. For GXW controller specifications, we develop a novel approach for synthesizing a set of synchronously communicating actor-based controllers. This synthesis algorithm proceeds by means of recursing over the structure of GXW specifications, and generates a set of dedicated and synchronously communicating sub-controllers according to the formula structure. In a subsequent step, 2QBF constraint solving identifies and tries to resolve potential conflicts between individual GXW specifications. This structural approach to GXW synthesis supports traceability between requirements and the generated control code as mandated by certification regimes for safety-critical software. Our experimental results suggest that GXW synthesis scales well to industrial-sized control synthesis problems with 20 input and output ports and beyond.

1 Introduction

Embedded control software in the manufacturing and processing industries is usually developed using specialized programming languages such as ladder diagrams or other IEC 61131-3 defined languages. Programming in these rather low-level languages is not only error-prone but also time- and resource-intensive. Therefore we are addressing the problem of correct-by-construction and automated generation of embedded control software from high-level requirements, which are expressed in a suitable fragment of linear temporal logic.

Moreover, an explicit correspondence between the high-level requirements and the generated control code is essential, since embedded control software is usually an integral part of safety-critical systems such as supervisory control and data acquisition (SCADA) systems for controlling critical machinery or infrastructure. In particular current industrial standards for safety-related development such as IEC 61508, DO 178C for avionics, and ISO 26262 for automotive applications mandate traceability between the control code and it requirements. Controllers generated by state-of-the-art LTL synthesis algorithms and tools such as generalized reactivity(1) (GR(1)) [15,25] or bounded LTL synthesis [8,11,28],

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part I, LNCS 9779, pp. 95–117, 2016.
DOI: 10.1007/978-3-319-41528-4_6

however, usually do not explicitly support such traceability requirements. For example, the GR(1) synthesis tool Anzu generates circuit descriptions in Verilog from BDDs [15].

We are therefore proposing a novel approach for synthesizing structured control software. In essence, the control code is generated by means of structural recursion on the given LTL formulas. Therefore, the structure of the control code corresponds closely to the syntactic structure of the given requirements, and there is a direct correspondence between controller components and subformulas of the specification.

In a first step towards this goal, we identify a fragment of LTL for specifying the input-output behavior of typical embedded control components. Besides the specification of input assumptions, invariance conditions on outputs, and transition-like reactions of the form $\mathbf{G}(\mathsf{input} \rightarrow \mathbf{X}^i\mathsf{output})$, this fragment also contains specifications of reactions of the form $\mathbf{G}(\mathsf{input} \rightarrow \mathbf{X}^i(\mathsf{output}\,\mathbf{W}\,\mathsf{release}))$, where input is an LTL formula whose validity is determined by the next i input valuations. The latter reaction formula states that if there is a temporal input event satisfying the constraint input, then the output constraint should hold on output events until there is a release event (or output always holds). The operator $\mathbf{G}$ is the universal path quantifier, $\mathbf{X}^i$ abbreviates i consecutive next-steps, $\mathbf{W}$ denotes the *weak until* temporal operator, the constraint output contains no temporal operator, and the subformula release may contain certain numbers of consecutive next-steps but no other temporal operators. The resulting fragment of LTL is called GXW. So far we have successfully modelled more than 70 different embedded control scenarios in GXW. The main source for this set of benchmarking problems are publicly available collections of industrial training materials for PLCs (including CODESYS 3.0 and AC500) [2,16,24]. The proposed GXW fragment of LTL is also similar to established requirements templates for specifying embedded control software in the aerospace domain, such as EARS [23].

Previous work on LTL synthesis (e.g., [5,7,8,10,11,14,15,25,28,31]) usually generates gate-level descriptions for the synthesized control strategies. In contrast, we generate controller in an actor language with high-level behavioral constructs and synchronous dataflow communication between connected actors. This choice of generating *structured controllers* is motivated by current practice of programming controllers using, say, Matlab Simulink [4], continuous function charts (IEC 61131-3), and Ptolemy II [12], which also supports synchronous dataflow (SDF) models [19]. Notice that the usual notions of LTL synthesis also apply to synthesis for SDF , since the composition of actors in SDF may also be viewed as Mealy machines with synchronous cycles [30].

Synthesis of structured controllers from GXW specifications proceeds in two subsequent phases. In the first phase, the procedure recurses on the structure of the given GXW formulas for generating dedicated actors for monitoring inputs events, for generating corresponding control events, and for wiring these actors according to the structure of the given GXW formulas. In the second phase, appropriate values for unknown parameters are synthesized in order to realize the conjunction of all given GXW specifications. Here we use satisfiability checking for quantified Boolean formula (2QBF) for examining if there exists such

conflicts between multiple GXW specifications. More precisely, existential variables of generated 2QBF problems capture the remaining design freedom when an output variable is not constrained by any trigger of low-level events. We demonstrate that controller synthesis for the GXW fragment is in PSPACE as compared to the 2EXPTIME -completeness result of full-fledged LTL [27]. Under some further reasonable syntactic restrictions on the GXW fragment we show that synthesis is in coNP .

An implementation of our GXW structural synthesis algorithm and application to our benchmark studies demonstrates a substantial speed-up compared to existing LTL synthesis tools. Moreover, the structure of the generated control code in SDF follows the structure of the given GXW specifications, and is more compact and, arguably, also more readable and understandable than commonly used gate-level representations for synthesized control strategies.

The paper is structured as follows. We introduce in Sect. 2 some basic notation for LTL synthesis, a definition of the GXW fragment of LTL and SDF actor systems together with the problem of actor-based LTL synthesis under GXW fragment. Section 3 illustrates GXW and actor-based control for such specifications by means of an example. Section 4 includes the main technical contributions and describes algorithmic workflow for generating structured controllers from GXW, together with soundness and complexity results for GXW synthesis. A summary of our experimental results is provided in Sect. 5, and a comparison of GXW synthesis with closely related work on LTL synthesis is included in Sect. 6. The paper closes with concluding remarks in Sect. 7. Due to space limits, some details are moved to an extended report [1].

2 Problem Formulation

We present basic concepts and notations of LTL synthesis, and we define the GXW fragment of LTL together with the problem of synthesizing actor-based synchronous dataflow controllers for GXW.

2.1 LTL Synthesis

Given two disjoint sets of Boolean variables V_{in} and V_{out}, the *linear temporal logic* (LTL) formulae over $\mathbf{2}^{V_{in} \cup V_{out}}$ is the smallest set such that (1) $v \in \mathbf{2}^{V_{in} \cup V_{out}}$ is an LTL formula, (2) if ϕ_1, ϕ_2 are LTL-formulae, then so are $\neg\phi_1$, $\neg\phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\phi_1 \rightarrow \phi_2$, and (3) if ϕ_1, ϕ_2 are LTL-formulae, then so are $\mathbf{G}\phi_1$, $\mathbf{X}\phi_1$, $\phi_1 \mathbf{U} \phi_2$. Given an ω-word σ, define $\sigma(i)$ to be the i-th element in σ, and define σ^i to be the suffix ω-word of σ obtained by truncating $\sigma(0) \ldots \sigma(i-1)$. The satisfaction relation $\sigma \vDash \phi$ between an ω-word σ and an LTL formula ϕ is defined in the usual way. The *weak until* operator, denoted $\mathbf{W}$, is similar to the *until* operator but the stop condition is not required to occur; therefore $\phi_1 \mathbf{W} \phi_2$ is simply defined as $(\phi_1 \mathbf{U} \phi_2) \vee \mathbf{G}\phi_1$. Also, we use the abbreviation $\mathbf{X}^i \phi$ to abbreviate i consecutive $\mathbf{X}$ operators before ϕ.

Table 1. Patterns defined in GXW specifications

ID	Meaning	Pattern
P1	Initial-until	$\varrho_{out}\,\mathbf{W}\,\phi^i_{in}$
P2	Trigger-until	$\mathbf{G}(\phi^i_{in} \rightarrow \mathbf{X}^i(\varrho_{out}\,\mathbf{W}\,(\varphi^j_{in} \vee \rho^0_{out})))$
P3	If-then	$\mathbf{G}(\phi^i_{in} \rightarrow \mathbf{X}^i\varrho_{out})$
P4	Iff	$\mathbf{G}(\phi^i_{in} \leftrightarrow \mathbf{X}^i\varrho_{out})$
P5	Invariance	$\mathbf{G}(\phi^0_{out})$
P6	Assumption	$\mathbf{G}(\phi^0_{in})$

Table 2. Specification patterns and corresponding skeleton specification.

Pattern ID	High-level control specification
P1	output **W** input
P2	**G**(input → (output **W** release))
P3	**G**(input → output)

A deterministic *Mealy machine* is a finite automaton $\mathcal{C} = (Q, q_0, \mathbf{2}^{V_{in}}, \mathbf{2}^{V_{out}}, \delta)$, where Q is set of (Boolean) state variables (thus $\mathbf{2}^Q$ is the set of states), $q_0 \in \mathbf{2}^Q$ is the initial state, $\mathbf{2}^{V_{in}}$ and $\mathbf{2}^{V_{out}}$ are sets of all input and output assignments defined by two disjoint sets of variables V_{in} and V_{out}. $\delta = \mathbf{2}^Q \times \mathbf{2}^{V_{in}} \rightarrow \mathbf{2}^{V_{out}} \times \mathbf{2}^Q$ is the transition function that takes (1) a state $q \in \mathbf{2}^Q$ and (2) input assignment $v_{in} \in \mathbf{2}^{V_{in}}$, and returns (1) an output assignment $v_{out} \in \mathbf{2}^{V_{out}}$ and (2) the successor state $q' \in \mathbf{2}^Q$. Let δ_{out} and δ_s be the projection of δ which considers only output assignments and only successor states. Given a sequence $a_0 \ldots a_k$ where $\forall i = 0 \ldots k, a_i \in \mathbf{2}^{V_{in}}$, let $\delta^k_s(q_0, a_0 \ldots a_k)$ abbreviate the output state derived by executing $a_0 \ldots a_k$ as an input sequence on the Mealy machine.

Given a set of input and output Boolean variables V_{in} and V_{out}, together with an LTL formula ϕ on V_{in} and V_{out} the *LTL synthesis problem* asks the existence of a controller as a deterministic Mealy machine $\mathcal{C}_\phi$ such that, for every input sequence $a = a_0 a_1 \ldots$, where $a_i \in \mathbf{2}^{V_{in}}$: (1) given the prefix a_0 produce $b_0 = \delta_{out}(q_0, a_0)$, (2) given the prefix $a_0 a_1$ produce $b_1 = \delta_{out}(\delta_s(q_0, a_0), a_1)$, (3) given the prefix $a_0 \ldots a_k a_{k+1}$, produce $b_{k+1} = \delta_{out}(\delta^k_s(q_0, a_0 \ldots a_k), a_{k+1})$, and (4) the produced output sequence $b = b_0 b_1 \ldots$ ensures that the word $\sigma = \sigma_1 \sigma_2 \ldots$, where $\sigma_i = a_i b_i \in \mathbf{2}^{V_{in} \cup V_{out}}$, $\sigma \vDash \phi$.

2.2 GXW Synthesis

We formally define the GXW fragment of LTL . Let ϕ^i, φ^i, ψ^i be LTL formulae over input variables V_{in} and output variables V_{out}, where all formulas are (without loss of generality) assumed to be in disjunctive normal form (DNF), and each literal is of form $\mathbf{X}^j v$ or $\neg\mathbf{X}^j v$ with $0 \leq j \leq i$ and $v \in V_{in} \cup V_{out}$. Clauses in DNF are also called *clause formulae*. Moreover, a formula ϕ^i_{in} is restricted to contain only input variables in V_{in}, and similarly, ϕ^i_{out} contains only output variables in V_{out}. Finally, ϱ_{out} denotes either v_{out} or $\neg v_{out}$, where v_{out} is an output variable.

For given input variables V_{in} and output variables V_{out}, a GXW *formula* is an LTL formula of one of the forms (P1)–(P6) as specified in Table 1. For example,

GXW formulas of the form (P2) stop locking ϱ_{out} as soon as $(\varphi^j_{in} \vee \rho^0_{out})$ holds. *GXW specifications* are of the form

$$\varrho \rightarrow \bigwedge_{m=1\ldots k} \eta_m \ , \tag{1}$$

where ϱ matches the GXW pattern (P6), and η_m matches one of the patterns (P1) through (P5) in Table 1. Furthermore, the notation "." is used for projecting subformulas from η_m, when it satisfies a given type. For example, assuming that sub-specification η_m is of pattern P3, i.e., it matches $\mathbf{G}(\phi^i_{in} \rightarrow \mathbf{X}^i \varrho_{out})$, $\eta_m.\varrho_{out}$ specifies the matching subformula for ϱ_{out}. Notice also that GXW specifications, despite including the **W** operator, have the *finite model property*, since the smallest number of unrolling steps for disproving the existence of an implementation is linear with respect to the structure of the given formula (cmp. Sect. 4.4).

Instead of directly synthesizing a Mealy machine as in standard LTL synthesis, we are considering here the generation of *actor-based controllers* using the computational model of *synchronous dataflow* (SDF) without feedback loops. An *actor-based controller* is a tuple $\mathcal{S} = (\mathcal{V}_{in}, \mathcal{V}_{out}, Act, \tau)$, where $\mathcal{V}_{in}$ and $\mathcal{V}_{out}$ are disjoint sets of external input and output ports. Each port is a variable which may be assigned a Boolean value or undefined if no such value is available at the port. In addition, actors $\mathcal{A} \in Act$ may be associated with internal input ports U_{in} and output ports U_{out} (all named apart), which are also three-valued. The projection $\mathcal{A}.\mathsf{u}$ denotes the port u of $\mathcal{A}$. An actor $\mathcal{A} \in Act$ defines Mealy machine $\mathcal{C}$ whose input and output assignments are based on $\mathbf{2}^{U_{in}}$ and $\mathbf{2}^{U_{out}}$, i.e., the output update function of $\mathcal{C}$ sets each output port to true or false , when each input port has value in {true, false}. Lastly, $\mathcal{A}^{(i)}$ denotes a *copy* of $\mathcal{A}$ which is indexed by i.

Let $Act.U_{in}$ and $Act.U_{out}$ be the set of all internal input and output ports for Act. The wiring $\tau \subseteq (\mathcal{V}_{in} \cup Act.U_{out}) \times (\mathcal{V}_{out} \cup Act.U_{in})$ connects one (external, internal) input port to one or more (external, internal) output ports. For convenience, denote the wiring from port out of $\mathcal{A}_1$ to port in of $\mathcal{A}_2$ as $(\mathcal{A}_1.\mathsf{out} \dashrightarrow \mathcal{A}_2.\mathsf{in})$. All ports are supposed to be connected, and every internal input port and every external output port is only connected to one wire (thus a port does not receive data from two different sources). Also, we do not consider actor systems with feedback loops here (therefore no cycles such as the one in Fig. 1(c)), since systems without feedback loops can be statically scheduled [18].

Evaluation cycles are triggered externally under the semantics of synchronous dataflow. In each such cycle, the data received at the external input ports is processed and corresponding values are transferred to external output ports. Notice also that the composition of actors under SDF acts cycle-wise as a Mealy machine [30]. We illustrate the *operational semantics* of actor-based systems under SDF by means of the example in Fig. 1(a), with input ports in1, in2, output port out, and actors f_1, f_2, f_3, f_4 (see also Fig. 1(b))[1]. Now, assume that

[1] The formal operational semantics, as it is standardized notation from SDF , is relegated to [1].

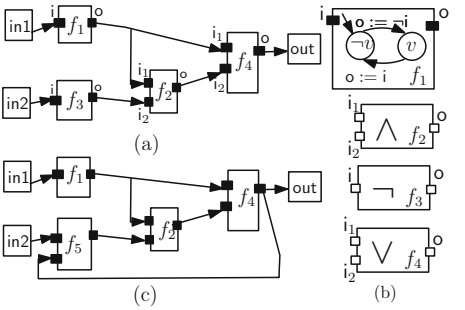

Fig. 1. An actor system allowing functional composition and corresponding actors f_1 (a)(b), feedback loops such as (c) are not considered here.

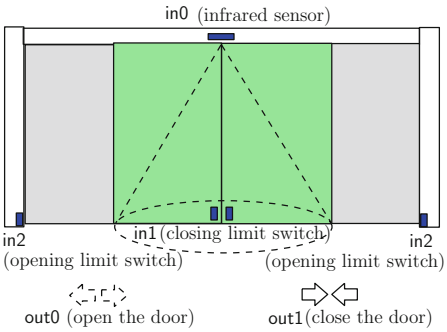

Fig. 2. Control of automatic door switch.

in the first cycle, the input ports in1 and in2 receive the value (false, true) and in the second cycle the value (false, true). The false value in in1 is copied to f_1.i. As f_1 is initially at state where v = false, it creates the output value true (places it to f_1.o) and changes its internal state to v = true. The value true from f_1.o is then transferred to f_4.i_1 and f_2.i_1. However, at this stage one cannot evaluate f_2 or f_4, as the i_2 port is not yet filled with a value. f_3 receives the value from in2 and produces f_3.o to false. Continuing this process, at the end of first cycle out is set to true, while in the second cycle, out is set to false.

As we do not consider feedback loops between actors in *Act*, from input read to output write, one can, using the enumeration method as exemplified above, create a static linear list Ξ of size $|Act| + |\tau|$, where each element $\xi_{ind} \in \Xi$ is either in *Act* or in τ, for specifying the linear order (from the partial order) how data is transferred between wires and actors. Such a total order Ξ is also called an *evaluation ordering* of the actor system $\mathcal{S}$.

One may wrap any Mealy machine $\mathcal{C}$ as an actor $\mathcal{A}(\mathcal{C})$ by simply creating corresponding ports in $\mathcal{A}(\mathcal{C})$ and by setting the underlying Mealy machine of $\mathcal{A}(\mathcal{C})$ to $\mathcal{C}$. Therefore, actor-based controllers may be synthesized for a given LTL specification ϕ by first synthesizing a Mealy machine $\mathcal{C}$ realizing ϕ, followed by the wrapping $\mathcal{C}$ as $\mathcal{A}(\mathcal{C})$, creating external I/O ports, and connecting external I/O ports with $\mathcal{A}(\mathcal{C})$.

Given a GXW specification ϕ over the input variables V_{in} and output variables V_{out}, the problem of GXW *synthesis* is to generate an actor-based SDF controller $\mathcal{S}$ realizing ϕ. As one can always synthesize a Mealy machine followed by wrapping it to an actor-based controller, GXW synthesis has the same complexity for Mealy machine and for actor-based controllers.

3 Example

We exemplify the use of GXW specifications and actor-based synthesis for these kinds of specification by means of an automatic sliding door[2], which is visualized in Fig. 2. Inputs and outputs are as follows: in0 is true when someone enters the sensing field; in1 denotes a closing limit switch - it is true when two doors touch each other; in2 denotes an opening limit switch - it is true when the door reaches the end; out0 denotes the opening motor - when it is set to true the motor rotates clockwise, thereby triggering the door opening action; and out1 denotes closing motor - when it is set to true the motor rotates counter-clockwise, thereby triggering the door closing action. Finally, the triggering of a timer t0 is modeled by means a (controllable) output variable t0start and the expiration of a timer is modeled using an (uncontrollable) input variable t0expire.

Before stating the formal GXW specification for the example we introduce some mnemonics.

- $\mathsf{entering}^1 := \neg\mathsf{in0} \wedge \mathbf{X}\,\mathsf{in0}$
- $\mathsf{expired}^1 := \neg\mathsf{t0expire} \wedge (\mathbf{X}\mathsf{t0expire})$
- $\mathsf{lim\_reached}^1 := \neg\mathsf{in2} \wedge \mathbf{X}\mathsf{in2}$
- $\mathsf{closing\_stopped} := \mathsf{in1} \vee \mathsf{in0} \vee \mathsf{out0}$

The superscripts denote the maximum number of consecutive next-steps. Now the automatic sliding door controller is formalized in GXW as follows.

S1: $\mathbf{G}(\mathsf{entering}^1 \rightarrow \mathbf{X}(\mathsf{out0}\,\mathbf{W}\,\mathsf{in2}))$
S2: $\mathbf{G}(\mathsf{expired}^1 \rightarrow \mathbf{X}(\mathsf{out1}\,\mathbf{W}\,\mathsf{closing\_stopped}))$
S3: $\neg\mathsf{out0}\,\mathbf{W}\,\mathsf{entering}^1$
S4: $\mathbf{G}(\mathsf{in2} \rightarrow \neg\mathsf{out0})$
S5: $\mathbf{G}(\mathsf{lim\_reached}^1 \leftrightarrow \mathbf{X}(\mathsf{t0start}))$
S6: $\mathbf{G}(\mathsf{in0} \rightarrow \neg\mathsf{out1})$
S7: $\mathbf{G}(\neg(\mathsf{out0} \wedge \mathsf{out1}))$

In particular, formula (S1) expresses the requirement that the opening of the door should continue (out0 = true) until the limit is reached (in2), and formulas (S3) and (S7) specify the expected initial behavior of the automatic sliding door. The GXW specifications for the sliding door example are classified as follows: formulas (S1), (S2) are of type (P2), (S3) is of type (P1), (S4), (S6) is of type (P3), (S5) is of type (P4), and (S7) of type (P5) according to Table 1.

Figure 3 visualizes an actor-based automatic sliding door controller which realizes the GXW specification (S1)-(S7). It is constructed from a small number of building blocks, which are also described in Fig. 3. Monitor actors, for example, are used for monitoring when the entering, expired, and lim_reached constraints are fulfilled, the OR actor is introduced because of the closing_stopped release condition in specification (S1), and the two copies of the *trigger-until* actors are introduced because of the (P2) shape of the specifications (S1) and (S2). The input and output ports of the *trigger-until* actor are in accordance with the namings for (P2) in Table 2. *Resolution actors* are used for resolving potential conflicts between individual GXW formulas in a specification. These actors are parameterized with respect to a Boolean A, which is the output of the resolution actor in case all inputs of this actor may be in $\{\mathsf{true}, \mathsf{false}\}$ (this set is denoted by the shorthand "–" in Fig. 3). The presented algorithm sets up a 2QBF problem for

[2] The automatic door example is adapted from http://plc-scada-dcs.blogspot.com/2014/08/basic-plc-ladder-programming-training_20.html.

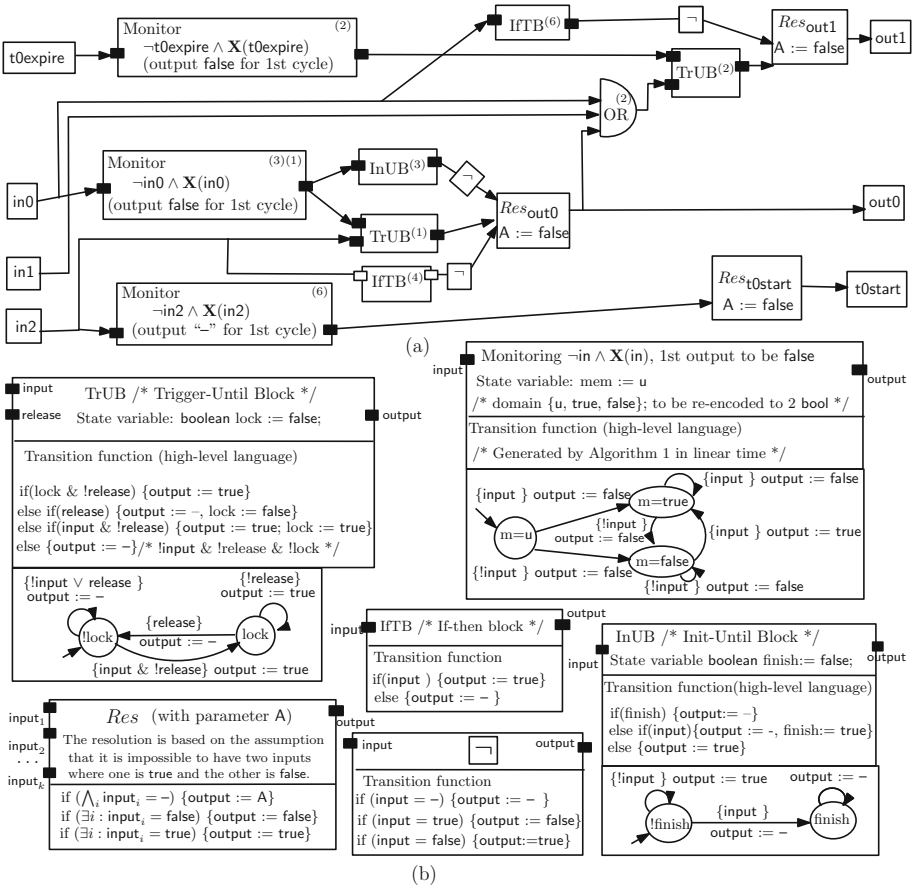

Fig. 3. Actor-based controller realizing automatic sliding door.

synthesizing possible values for these parameters. Because of the constraint (S7) on possible outputs out0 and out1, the parameter A for the resolution actor for output out0, for example, needs to be set to A:=false . Figure 3 also includes the operational behavior of selected actors in terms of high-level transitions and/or Mealy machines. The internal state and behavior for monitor actors, however, is synthesized, in linear time, from a given GXW constraint on inputs (see Sect. 4).

Finally, the structural correspondence of the actor-based controller in Fig. 3 with the given GXW specification of the sliding door example is being made explicit by superscripting actors with index (i) whenever the actor has been introduced due to the i-th specification.

4 Structural Synthesis

We now describe the algorithmic details for generating structured controllers from the GXW specifications of the form $\varrho \rightarrow \bigwedge_{m=1\ldots k} \eta_m$. The automated sliding door is used as running example for illustrating the result of each step.

First, our algorithm prepares I/O ports, iterates through every formula η_m for creating high-level controllers (Step 1) based on the appropriate GXW pattern. For specifications of types P1 to P3, Table 2 lists the corresponding LTL specification (as high-level control objective), where input and release are input Boolean variables, output is an output Boolean variable.

Then, for each GXW formula, the algorithm constructs actors and wirings for monitoring low-level events by mimicking the DNF formula structure (Steps 2 and 3). On the structural level of clause formulas in DNF, the algorithm constructs corresponding controllers in linear time (Algorithm 1). Finally, the algorithm applies 2QBF satisfiability checking (and synthesis of parameters for resolution actors) for guaranteeing nonexistence of potential conflicts between different formulas in the GXW specifications (Step 4).

4.1 High-Level Control Specifications and Resolution Actors

The initial structural recursion over GXW formulas is described in Step 1.

Step 1.1 - Controller for high-level control objectives. Line 1 associates the three high-level controller actors InUB, TrUB, IfTB with their corresponding pattern identifier. Implementations for the actors InUB, TrUB, IfTB are listed in Fig. 3(b). For example, the actor IfTB is used for realizing $\mathbf{G}(\mathsf{input} \rightarrow \mathsf{output})$ in Table 2. When input equals false, the output produced by this actor equals "–". This symbol is used as syntactic sugar for the set $\{\mathsf{true}, \mathsf{false}\}$. Therefore the output is unconstrained, that is, it is feasible for output to be either true or false. The value "–" is transferred in the dataflow, thereby allowing the delay of decisions when considering multiple specifications influencing the same output variable.

Step 1.2 - External I/O ports. Line 2 and 3 are producing external input and output port for each input variable $v_{in} \in V_{in}$ and output variable $v_{out} \in V_{out}$.

Step 1.3 - High-level control controller instantiation. Lines 4 and 5 iterate through each specification η_m to find the corresponding pattern (using DetectPattern). Based on the corresponding type, line 6 creates a high-level controller by copying the content stored in the map. If there exists a specification which does not match one of the patterns, immediately reject (line 7). Notice that pattern P4 is handled separately in Step 3. For the door example, the controller in Fig. 3(a) contains the two copies $TrUB^{(1)}$ and $TrUB^{(2)}$ of the trigger-until actor $TrUB$; the subscripts of these copies are tracing the indices of the originating formulas (S1) and (S2).

Step 1. Initiate external I/O ports, high-level controller and resolution controllers

Input : LTL specification $\phi = \varrho \rightarrow \bigwedge_{m=1\ldots k} \eta_m$, input and output variables V_{in}, V_{out}

Output: Actor-based (partial) ctrl implementation $\mathcal{S} = (\mathcal{V}_{in}, \mathcal{V}_{out}, Act, \tau)$, map_{out}

1 **let** $\mathsf{map}_{pattern}$:= {P1 $\mapsto$ InUB, P2 $\mapsto$ TrUB, P3 $\mapsto$ IfTB}
2 $\mathcal{V}_{in}$:= $\{ \boxed{v_{in}} \mid v_{in} \in V_{in}\}$
3 $\mathcal{V}_{out}$:= $\{ \boxed{v_{out}} \mid v_{out} \in V_{out}\}$
4 **foreach** η_m, $m = 1 \ldots k$ **do**
5 | **if** $(p := \mathsf{DetectPattern}(\eta_m)) \in \{P1, P2, P3\}$ **then**
6 | | Create actor $\mathcal{A}^{(m)}$ from $\mathcal{A} := \mathsf{map}_{pattern}.\mathrm{get}(p)$, and add to $\mathcal{S}$;
7 | **else if** $(p := \mathsf{DetectPattern}(\eta_m)) \notin \{P4, P5, P6\}$ **then return error** ;
8 **let** map_{out} := NewEmptyMap();
9 **foreach** $v_{out} \in V_{out}$ **do** map_{out}.put(v_{out}, NewEmptyList()); ;
10 **foreach** η_m, $m = 1 \ldots k$ **do**
11 | map_{out}.get(v_{out}).add(m), where v_{out} is the output variable used in $\tau_m.\varrho_{out}$;
12 **foreach** $v_{out} \in V_{out}$ **do** Add actor $Res_{v_{out}}$:= CreateResActor(map_{out}.get(v_{out}).size()) to Act ;
13 **foreach** η_m, $m = 1 \ldots k$ **do**
14 | **let** v_{out} be the variable used in $\tau_m.\varrho_{out}$, ind := map_{out}.get(v_{out}).indexOf(m);
15 | **if** $\neg v_{out}$ *equals* $\tau_m.\varrho_{out}$ **then** `// negation is used in literal`
16 | | Create a negation actor $\boxed{\neg}^{(m)}$ and add it to Act;
17 | | $\tau := \tau \cup \{(\mathcal{A}^{(m)}.\mathsf{output} \dashrightarrow \boxed{\neg}^{(m)}.\mathsf{input}), (\boxed{\neg}^{(m)}.\mathsf{output} \dashrightarrow Res_{v_{out}}.\mathsf{input}_{\mathsf{ind}})\}$;
18 | **else** $\tau := \tau \cup \{(\mathcal{A}^{(m)}.\mathsf{output} \dashrightarrow Res_{v_{out}}.\mathsf{input}_{\mathsf{ind}})\}$;
19 **foreach** $v_{out} \in V_{out}$ **do** $\tau := \tau \cup \{(Res_{v_{out}}.\mathsf{output} \dashrightarrow \boxed{v_{out}})\}$;

Step 1.4 - Resolution Actors. This step is to consider all sub-specifications that influence the same output variable v_{out}. Line 9 to 11 adds, for each specification η_m using v_{out}, its index m maintained by map_{out}.get(v_{out}). E.g., for the door example, specifications S1, S3 and S4 all output out0. Therefore after executing line 10 and 11, we have map_{out}.get(out0)={1, 3, 4}, meaning that for variable out0, the value is influenced by S1, S3 and S4.

For each output variable v_{out}, line 12 creates one *Resolution Actor* $Res_{v_{out}}$ which contains one parameter equaling the number of specifications using v_{out} in ϱ_{out}. Here we make it a simple memoryless controller as shown in Fig. 3(b) - $Res_{v_{out}}$ outputs true when one of its inputs is true, outputs false when one of its inputs is false, and outputs A (which is currently an unknown value to be synthesized later) when all inputs are "–". The number of input pins is decided by calling the map. E.g., for Res_{out0} in Fig. 3(a), three inputs are needed because map_{out}.get(out0).size()=3. The output of the high-level controller $A^{(m)}$ is connected to the input of $Res_{v_{out}}$. When negation is needed due to the negation symbol in ϱ_{out} (line 15), one introduces a negation actor $\boxed{\neg}$ which

Step 2. Synthesize monitoring controllers (for pattern P1, P2, P3)

Input : $\phi = \varrho \rightarrow \bigwedge_{m=1\ldots k} \eta_m$, V_{in}, V_{out}, $\mathcal{S} = (\mathcal{V}_{in}, \mathcal{V}_{out}, Act, \tau)$ from Step 1.
Output: (partial) controller $\mathcal{S} = (V_{in}, V_{out}, Act, \tau)$ by adding more elements

foreach η_m, $m = 1 \ldots k$ **do**
  $p :=$ DetectPattern(η_m);
  if $p \in \{P1, P2, P3\}$ **then** `// Work on subformula ‘‘input ’’ of` η_m`, defined by` ϕ^i_{in}
    Add an OR-gate actor $\mathsf{OR}_{\phi^i_{in}}$ with $\mathsf{size}(\eta_m.\phi^i_{in})$ inputs to Act;
    foreach *clause formula* χ^i_{in} *from DNF of* $\eta_m.\phi^i_{in}$ **do**
      Add $\mathcal{A}(\mathcal{C})$ to Act, where $\mathcal{C} := \mathsf{Syn}(\mathbf{G}(\chi^i_{in} \leftrightarrow \mathbf{X}^i \mathsf{out}) \wedge \bigwedge_{z=0}^{i-1} \mathbf{X}^z \neg \mathsf{out}, \mathsf{In}(\chi^i_{in}), \{\mathsf{out}\})$;
      foreach $v_{in} \in \mathit{In}(\chi^i_{in})$ **do** $\tau := \tau \cup \{(\boxed{v_{in}} \dashrightarrow \mathcal{A}(\mathcal{C}).v_{in})\}$;
      $\tau := \tau \cup \{(\mathcal{A}(\mathcal{C}).\mathsf{out} \dashrightarrow \mathsf{OR}_{\phi^i_{in}}.\mathsf{in}_{\mathsf{Index}(\chi^i_{in}, \phi^i_{in})})\}$;
    $\tau := \tau \cup \{(\mathsf{OR}_{\phi^i_{in}}.\mathsf{out} \dashrightarrow \mathcal{A}^{(m)}.\mathsf{input})\}$;
  if $p \in \{P2\}$ **then** `// Work on subformula ‘‘release ’’ of` η_m`, defined by` $\varphi^j_{in} \vee \rho^0_{out}$
    Add an OR-gate actor $\mathsf{OR}_{\varphi^j_{in} \vee \rho^0_{out}}$ with $\mathsf{size}(\eta_m.(\varphi^j_{in} \vee \rho^0_{out}))$ inputs to Act;
    foreach *clause formula* χ^h_{in} *from DNF* $\eta_m.\varphi^j_{in}$ **do**
      Add $\mathcal{A}(\mathcal{C})$ to Act, where $\mathcal{C} := \mathsf{Syn}(\mathbf{G}(\chi^h_{in} \leftrightarrow \mathbf{X}^h \mathsf{out}) \wedge \bigwedge_{z=0}^{h-1} \mathbf{X}^z \neg \mathsf{out}, \mathsf{In}(\chi^h_{in}), \{\mathsf{out}\})$;
      foreach $v_{in} \in \mathit{In}(\chi^i_{in})$ **do** $\tau := \tau \cup \{(\boxed{v_{in}} \dashrightarrow \mathcal{A}(\mathcal{C}).v_{in})\}$;
      if $h = 0$ **then** Add $(\mathcal{A}(\mathcal{C}).\mathsf{out} \dashrightarrow \mathsf{OR}_{\varphi^j_{in} \vee \rho^0_{out}}.\mathsf{in}_{\mathsf{Index}(\chi^z_{in}, \varphi^j_{in} \vee \rho^0_{out})})$ to τ ;
      else
        Add $\mathcal{A}(\mathcal{C}_{\Theta_h})$ to Act, where $\mathcal{C}_{\Theta_h} :=$ CreateThetaCtrl(h);
        $\tau := \tau \cup \{(\mathsf{OR}_{\phi^i_{in}}.\mathsf{out} \dashrightarrow \mathcal{A}(\mathcal{C}_{\Theta_h}).\mathsf{set}), (\mathcal{A}(\mathcal{C}).\mathsf{out} \dashrightarrow \mathcal{A}(\mathcal{C}_{\Theta_h}).\mathsf{in}), (\mathcal{A}(\mathcal{C}_{\Theta_h}).\mathsf{out} \dashrightarrow \mathsf{OR}_{\varphi^j_{in} \vee \rho^0_{out}}.\mathsf{in}_{\mathsf{Index}(\chi^z_{in}, \varphi^j_{in} \vee \rho^0_{out})})\}$;
    foreach *clause formula* χ^0_{out} *from DNF of* $\eta_m.\rho^0_{out}$ **do**
      Add an AND-gate actor $\mathsf{AND}_{\eta_m.\chi^0_{out}}$ with $\mathsf{size}(\chi^0_{out})$ inputs to Act;
      foreach *literal* ω_{out} *of* χ^0_{out} **do**
        let v_{out} be the variable used in ω_{out};
        if ω_{out} *equals* $\neg v_{out}$ **then** `// negation is used in literal`
          Create $\boxed{\neg}_{Res_{v_{out}}}$ and add to Act, if not exists;
          Add $(Res_{v_{out}}.\mathsf{output} \dashrightarrow \boxed{\neg}_{Res_{v_{out}}}.\mathsf{input})$ to τ, if not exists;
          Add $(\boxed{\neg}_{Res_{v_{out}}}.\mathsf{output} \dashrightarrow \mathsf{AND}_{\chi^0_{out}}.\mathsf{in}_{\mathsf{Index}(\omega_{out}, \chi^0_{out})})$ to τ;
        else $\tau := \tau \cup \{Res_{v_{out}}.\mathsf{output} \dashrightarrow \mathsf{AND}_{\chi^0_{out}}.\mathsf{in}_{\mathsf{Index}(\omega_{out}, \chi^0_{out})}\}$;
      $\tau := \tau \cup \{\mathsf{AND}_{\chi^0_{out}}.\mathsf{out} \dashrightarrow \mathsf{OR}_{\varphi^j_{in} \vee \rho^0_{out}}.\mathsf{in}_{\mathsf{Index}(\chi^0_{out}, \varphi^j_{in} \vee \rho^0_{out})}\}$
    $\tau := \tau \cup \{\mathsf{OR}_{\phi^j_{in} \vee \rho^0_{out}}.\mathsf{out} \dashrightarrow \mathcal{A}^{(m)}.\mathsf{release}\}$;

negates $A^{(m)}.\mathsf{output}$ when $A^{(m)}.\mathsf{input}$ is true or false (line 16, 17). To ensure that connections are wired appropriately, map_{out} is used such that the number "ind" records the precise input port of the Res_{out} (line 14). Consider again the door example. Due to the maintained list $\{1, 3, 4\}$, $TrUB^{(1)}.\mathsf{output}$ is connected to $Res_{v_{out}}.\mathsf{input}_1$, i.e., the first input pin of $Res_{v_{out}}$. Also, as $\neg\mathsf{out0}$ is used in S3 and S4, the wiring from $InUB^{(3)}$ and $IfTB^{(4)}$ to Res_{out0} in Fig. 3(a) has a negation actor in between.

Lastly, line 19 connects the output port of a resolution actor to the corresponding external output port. If $Res_{v_{out}}$ receives simultaneously true and false from two of its input ports, then $Res_{v_{out}}.\mathsf{output}$ needs to be simultaneously true and false. These kinds of situations are causing unrealizability of GXW specification, and Step 4 is used for detecting these kinds of inconsistencies.

4.2 Monitors and Phase Adjustment Actors

The second step of the algorithm synthesizes controllers for monitoring the appearance of an event matching the subformula, and connects these controllers to previously created actors for realizing high-level control objectives. For a formula ϕ in DNF form, let $\mathsf{size}(\phi)$ return the number of clauses in ϕ. For clause formula χ^i_{in} in ϕ, let $\mathsf{In}(\chi^i_{in})$ return the set of all input variables and $\alpha = \mathsf{Index}(\chi^i_{in}, \phi^i_{in})$ specify that χ^i_{in} is the α-th clause in ϕ^i_{in}.

Step 2.1 - Realizing "input" *part for pattern P1, P2, P3.* In Step 2, from line 3 to 9, the algorithm synthesizes controller realizing the portion input listed in Table 2, or equivalently, the ϕ^i_{in} part listed in Table 1. Line 4 first creates an OR gate, as the formula is represented in DNF. Then synthesize a controller for monitoring each clause formula (line 5, 6) using function Syn, with input variables defined in $\mathsf{In}(\chi^i_{in})$ and a newly introduced output variable $\{\mathsf{out}\}$[3]. The first attempt is to synthesize $\mathbf{G}(\chi^i_{in} \leftrightarrow \mathbf{X}^i\mathsf{out})$. By doing so, the value of χ^i_{in} is reflected in out. However, as the output of the synthesized controller is connected to the input of an OR-gate (line 8) and subsequently, passed through the port "input" of the high-level controller (line 9), one needs to also ensure that from time 0 to $i-1$, out remains false, such that the high-level controller $\mathcal{A}_m$ for specification η_m will not be "unintentionally" triggered and subsequently restrict the output. To this end, the specification to be synthesized is $\mathbf{G}(\chi^i_{in} \leftrightarrow \mathbf{X}^i\mathsf{out}) \wedge \bigwedge_{z=0\ldots i-1} \mathbf{X}^z\neg\mathsf{out}$, being stated in line 6.

For above mentioned property that needs to be synthesized in line 6, one does not need to use full LTL synthesis algorithms. Instead, we present a simpler algorithm (Algorithm 1) which creates a controller in time linear to the number of variables times the maximum number of $\mathbf{X}$ operators in the formula. Here again for simplicity, each state variable is three-valued (true, false, u); in implementation every 3-valued state variable is translated into 2 Boolean variables. In the

[3] For pattern type P2 or P3, one needs to have each clause formula of ϕ^i_{in} be of form χ^i_{in}, i.e., the highest number of consecutive $\mathbf{X}$ should equal i. The purpose is to align χ^i_{in} with the preceding $\mathbf{X}^i$ in $\mathbf{G}(\phi^i_{in} \rightarrow \mathbf{X}^i(\varrho_{out}\,\mathbf{W}\,(\varphi^j_{in} \vee \rho^0_{out})))$ or $\mathbf{G}(\phi^i_{in} \rightarrow \mathbf{X}^i\varrho_{out})$. If a clause formula in DNF contains no literal starting with $\mathbf{X}^i$, one can always pad a conjunction $\mathbf{X}^i\,\mathsf{true}$ to the clause formula. The padding is not needed for P1.

Algorithm 1. Realizing Syn without full LTL synthesis

Input : LTL specification $\mathbf{G}(\chi^i_{in} \leftrightarrow \mathbf{X}^i\mathsf{out}) \wedge \bigwedge_{z=0}^{i-1} \mathbf{X}^z\neg\mathsf{out})$, input variables $\mathsf{In}(\chi^i_{in})$, output variables {out }

Output: Mealy machine $\mathcal{C} = (Q, q_0, \mathbf{2}^{V_{in}}, \mathbf{2}^{V_{out}}, \Delta)$ for realizing the specification

1 $V_{out} := \{\mathsf{out}\}$, $V_{in} := \mathsf{In}(\chi^i_{in})$;

2 **foreach** *Variable* $v_{in} \in \mathsf{In}(\chi^i_{in})$ **do** `// Create all state variables in the Mealy machine`

3 | **for** $j = 1 \ldots i$ **do** $Q := Q \cup \{v_{in}[j]\}$, where $v_{in}[j]$ is three-valued (true , false , u) ;

4 $q_0 := \bigwedge_{v_{in} \in \mathsf{In}(\chi^i_{in}), j \in \{1,\ldots i\}} v_{in}[j] := \mathsf{u}$ /* Initial state */;

5 **let** Cd := true /* Cd for output condition */;

6 **foreach** *literal* $\boldsymbol{X}^k v_{in}$ *in* χ^i_{in} **do**

7 | **if** $k = i$ **then** Cd := Cd $\wedge (v_{in} = \mathsf{true})$ **else** Cd := Cd $\wedge (v_{in}[i-k] = \mathsf{true})$;

8 **foreach** *literal* $\boldsymbol{X}^k \neg v_{in}$ *in* χ^i_{in} **do**

9 | **if** $k = i$ **then** Cd := Cd $\wedge (v_{in} = \mathsf{false})$ **else** Cd := Cd $\wedge (v_{in}[i-k] = \mathsf{false})$;

10 δ_{out} := (out := Cd) /* Output assignment should follow the value of Cd */;

11 $\delta_s := (\bigwedge_{v_{in} \in \mathsf{In}(\chi^i_{in}), j=1\ldots i-1} v_{in}[j+1] := v_{in}[j]) \wedge (\bigwedge_{v_{in} \in \mathsf{In}(\chi^i_{in})} v_{in}[1] := v_{in})$;

algorithm, state variable $v_{in}[i]$ is used to store the i-step history of for v_{in}, and $v_{in}[i] = \mathsf{u}$ means that the history is not yet recorded. Therefore, for the initial state, all variables are set to u (line 4). The update of state variable $v_{in}[i+1]$ is based on the current state of $v_{in}[i]$, but for state variable $v_{in}[1]$, it is updated based on current input v_{in} (line 11). With state variable recording previously seen values, monitoring the event is possible, where the value of out is based on the condition stated from line 6 to 10.

Consider a controller realizing $\chi^i_{in} := \neg\mathsf{in1} \wedge \mathbf{X}\mathsf{in1} \wedge \mathbf{X}\mathsf{in2} \wedge \mathbf{XX}\neg\mathsf{in2}$, being executed under a run prefix (false, false)(true, true)(true, false). As shown in Fig. 4, the update of state variables is demonstrated by a left shift. The first and the second output are false. After receiving the third input, the controller is able to detect a rising edge of in1 (via in1[2]=false and in1[1]=true) is immediately followed by a falling edge of in2 (via in2[1]=true and in2 =false).

Step 2.2 - Realizing "release" *part for pattern P2.* Back to Step 2, the algorithm from line 10 to 29 synthesizes a controller realizing the portion release listed in Table 2, or equivalently, the $\varphi^j_{in} \vee \rho^0_{out}$ part listed in Table 1. The DNF structure is represented as an OR-actor (line 11), taking input from φ^j_{in} (line 12–18) and ρ^0_{out} (line 19–28).

For ρ^0_{out} (line 19–28), first create an AND-gate for each clause in DNF. Whenever output variable v_{out} is used, the wiring is established by a connection to the output port of $Res_{v_{out}}$ (line 27). Negation in the literal is done by adding a wire to connect $Res_{v_{out}}$ to a dedicated negation actor $\boxed{\neg}_{Res_{v_{out}}}$ to negate the output (line 23 to 26). Consider, for example, specification $S2$ of the automatic door running example, where the "release" part (in1 $\vee$ in0 $\vee$ out0) is a

disjunction of literals using output variable out0. As a consequence, one creates an AND-gate (line 20) which takes one input $Res_{\mathsf{out0}}.\mathsf{output}$ (line 27), and connects this AND-gate to the OR-gate (line 28). Figure 3(a) displays an optimized version of this construction, since the single-input AND-gate may be removed and $Res_{\mathsf{out0}}.\mathsf{output}$ is directly wired with the OR-gate.

For φ^j_{in} (line 12 to 23), similar to Step 2.1, one needs to synthesize a controller which tracks the appearance of χ^h_{in} (line 13). However, the start of tracking is triggered by ϕ^i_{in} (the input subformula). That is, whenever ϕ^i_{in} is true, start monitoring if φ^j_{in} has appeared true. This is problematic when χ^h_{in} contains $\mathbf{X}$ operators (i.e., $h > 0$). To realize this mechanism, at line 17, the function CreateThetaCtrl additionally initiates a controller which guarantees the following: Whenever input variable set turns true, the following h values of output variable out are set to false. After that, the value of output variable out is the same as the input variable in. This property can be formulated as Θ_h (to trigger consecutive h false value over out after seeing set = true) listed in Eq. 2, with implementation shown in Fig. 6. By observing the Mealy machine and the high-level transition function, one infers that the time for constructing such a controller in symbolic form is again linear in h.

$$\Theta_h := (\neg \mathsf{out}\, \mathbf{W}\, \mathsf{set}) \wedge \mathbf{G}(\mathsf{set} \rightarrow (\bigwedge_{z=0}^{h-1} \neg \mathbf{X}^z \mathsf{out} \wedge \mathbf{X}^h((\mathsf{in} \leftrightarrow \mathsf{out})\, \mathbf{W}\, \mathsf{set}))) \quad (2)$$

The overall construction in Step 2 is illustrated using the example in Fig. 5, which realizes the formula

$$\mathbf{G}((\neg \mathsf{in1} \wedge \mathbf{X}\mathsf{in1}) \rightarrow \mathbf{X}(\mathsf{out1}\, \mathbf{W}(\neg \mathsf{in2} \wedge \mathbf{X}\mathsf{in2}))) \quad (3)$$

with $V_{in} = \{\mathsf{in1}, \mathsf{in2}\}$ and $V_{out} = \{\mathsf{out1}\}$. This specification requires to set output out1 to true when a rising edge of in1 appears, and after that, out1 should remain true until detecting a raising edge of in2. Using the algorithm listed in Step 2, line 6 synthesizes the monitor for the input part (i.e., detecting rising edge of in1), line 13 synthesizes the monitor for the release part (i.e., detecting rising edge of in2), line 14 creates the wiring from input port to the monitor. As $h = 1$ (line 16), line 17 creates $\mathcal{A}(\mathcal{C}_{\Theta_1})$, and line 18 establishes the wiring to and from $\mathcal{A}(\mathcal{C}_{\Theta_1})$.

The reader may notice that it is incorrect to simply connect the monitor controller for $\neg\mathsf{in2} \wedge \mathbf{X}\mathsf{in2}$ directly to TrUB.release, as, when both $\neg\mathsf{in1} \wedge \mathbf{X}\mathsf{in1}$ and $\neg\mathsf{in2} \wedge \mathbf{X}\mathsf{in2}$ are true at the same time, TrUB.output is unconstrained. On the contrary, in Fig. 5, when $\neg\mathsf{in1} \wedge \mathbf{X}\mathsf{in1}$ is true and the value is passed through TrUB.input, $\mathcal{A}(\mathcal{C}_{\Theta_1})$ enforces to invalidate the incoming value of TrUB.release for 1 cycle by setting it to false.

Step 3 - Realizing "input" *for pattern P4.* For pattern P4, in contrast to pattern P1, P2, and P3, the synthesized monitoring element is directly connected to a Resolution Actor (see Fig. 3(a) for example). To maintain maximum freedom over output variable, one synthesizes the event monitor from the specification allowing the first consecutive i output to be "–". The monitor construction is analogous to Algorithm 1 and we refer readers to [1] for details.

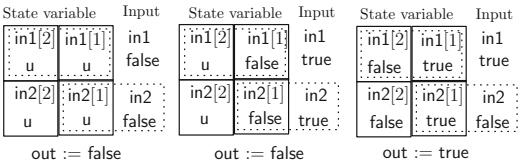

Fig. 4. Executing monitor with $\chi^i_{in} := \neg\mathsf{in1} \wedge \mathbf{X}\mathsf{in1} \wedge \mathbf{X}\mathsf{in2} \wedge \mathbf{XX}\neg\mathsf{in2}$, by taking first three inputs (false, false)(true, true)(true, false).

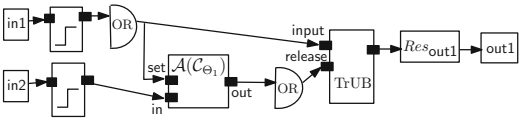

Fig. 5. Correct controller construction for specification satisfying pattern P2.

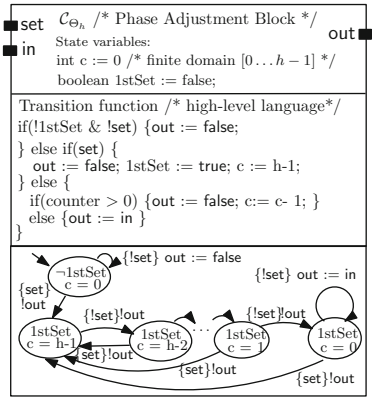

Fig. 6. Implementing Θ_h (state variables not mentioned in update remain the same value).

Optimizations. Runtimes for Steps 2 and 3 may be optimized by using simple pattern matching and hashing of previously synthesized controllers. We are listing three different opportunities for optimized *generation of monitors*. First, the controller in Fig. 3(a) for monitoring $\neg\mathsf{in0} \wedge \mathbf{X}\,\mathsf{in0}$ is connected to two high-level controllers. The second case can be observed in Fig. 5, where by rewriting in1 and in2 to in, the controller being synthesized is actually the same. Therefore, one can also record the pattern for individual monitor and perform synthesis once per pattern. A third opportunity for optimization occurs when Algorithm 1 takes $i = 0$ (i.e., no $\mathbf{X}$ operator is used). In this case there is no need to create a controller at all and one may proceed by directly building a combinatorial circuit, similar to the constructions of line 19 to 28 in Step 2. For example, for specification S2 of the automatic door, the release part is $\mathsf{in1} \vee \mathsf{in0} \vee \mathsf{out0}$; since no $\mathbf{X}$ operator occurs, a combinatorial circuit is created by wiring directly $\boxed{\mathsf{in1}}$ and $\boxed{\mathsf{in0}}$ to the OR-gate.

4.3 Parameter Synthesis for 2QBF Without Unroll

Previous steps construct actors as building blocks and wires the actors according to the structure of the given GXW specification from type P1 to P4. The resulting (partial) controller, however, does not yet realize this specification as it may still contain unknowns in the resolution actors. Further checks are necessary, and a controller is rejected if one of the following conditions holds.

(Condition 1) The wiring forms a directed loop in the constructed actor-based controller.

(Condition 2) It is possible for a resolution actor $Res_{v_{out}}$ to receive true and false simultaneously.
(Condition 3) Outputs violate invariance conditions of pattern P5.

Condition 1 is checked by means of a simple graph analysis: (1) let all ports be nodes and wirings be edges; (2) for each actor, create directed edges from each of its input ports to each of its output ports; (3) check if there exists a strongly connected component in the resulting graph using, for example, Tarjan's algorithm [29].

Conditions 2 and 3 are checked by means of creating corresponding 2QBF satisfiability problems. Recall that each resolution actor $Res_{v_{out}}$ is parameterized with respect to the output A when all incoming inputs for $Res_{v_{out}}$ are "–". The corresponding parameter assignment problem is encoded as a 2QBF[4] formula, where existential variables are the parameters to be synthesized, universal variables are input variables, and the quantifier-free body is a logical implication specifying that the encoding of the system guarantees condition 2 and 3.

Step 4 shows a simplified algorithm for generating 2QBF constraints which does not perform unrolling. Stated in line 15, the quantifier free formula is of form $\Upsilon_a \rightarrow \Upsilon_g$, where Υ_a are input assumptions and system dynamics, and Υ_g are properties to be guaranteed. First, unknown parameters are added to the set of existential variables $V_\exists$ (line 2). All other variables are universal variables. Then based on the evaluation ordering of $\mathcal{S}$, perform one of the following tasks:

- When an element ξ in the execution ordering Ξ is a wire (line 5), we add source and dest as universal variables (as $V_\forall$ is a set, repeated variables will be neglected), and establish the logical constraint (source $\leftrightarrow$ dest) (lines 6 to 8).
- When an element ξ in the execution ordering Ξ is an actor, we use function EncodeTransition to encode the transition (pre-post) relation as constraints (line 11), and add all state variables (for pre and post) in the actor (recall our definition of Mealy machine is based on state variables) to $V_\forall$ using function GetStateVariable (line 10).

Υ_a is initially set to ϱ (line 1) to reflect the allowed input patterns regulated by the specification (specification type P6). Line 12 creates the constraint stating that no two inputs of a resolution actor should create contradicting conditions. As the number of input ports for any resolution actor is finite, the existential quantifier is only an abbreviation which is actually rewritten to a quantifier-free formula describing relations between input ports of a resolution actor.

The encoding presented in Step 4 does not involve unroll (it encodes the transition relation, but not the initial condition). Therefore, by setting all variables to be universally quantified, one approximates the behavior of the system dynamics without considering the relation between two successor states. Therefore, using

[4] Quantified Boolean Formula with one top-level quantifier alternation.

Step 4. Parameter synthesis by generating 2QBF constraints

Input : LTL specification $\phi = \varrho \rightarrow \bigwedge_{m=1\ldots k} \eta_m$, input variables V_{in}, output variables V_{out}, partial implementation $\mathcal{S} = (V_{in}, V_{out}, Act, \tau)$ with unknown parameters

Output: Controller implementation $\mathcal{S}$ or "unknown"

1 **let** $\Upsilon_a := \varrho, \Upsilon_g :=$ true, $V_\exists, V_\forall :=$ NewEmptySet();
2 **foreach** $v_{out} \in V_{out}$ **do** $V_\exists := V_\exists \cup \{Res_{v_{out}}.\mathsf{A}\}$;
3 **let** Ξ be the evaluation ordering of $\mathcal{S}$;
4 **foreach** $\xi \in \Xi$ **do**
5 | **if** $\xi \in \tau$ **then** `// ξ is a wire; encode using biimplication`
6 | | Let ξ be (source $\dashrightarrow$ dest);
7 | | $V_\forall.add$(source), $V_\forall.add$(dest);
8 | | $\Upsilon_a := \Upsilon_a \wedge$ (source $\leftrightarrow$ dest);
9 | **else** `// ξ is an actor; encode transition`
10 | | $V_\forall.add$(GetStateVariable(ξ));
11 | | $\Upsilon_a := \Upsilon_a \wedge$ (EncodeTransition(ξ)) /* $\xi \in Act$ */ ;
12 **for** $v_{out} \in V_{out}$ **do**
$\Upsilon_g := \Upsilon_g \wedge (\nexists i, j : (Res_{v_{out}}.\mathsf{input}_i = \mathsf{true}) \wedge (Res_{v_{out}}.\mathsf{input}_j = \mathsf{false}))$;
13 **foreach** η_m, $m = 1 \ldots k$ **do**
14 | **if** DetectPattern(η_m) $\in \{P5\}$ **then** $\Upsilon_g := \Upsilon_g \wedge \eta_m$;
15 **if** Solve2QBF$(V_\exists, V_\forall, \Upsilon_a \rightarrow \Upsilon_g)$.isSatisable **then**
16 | **return** $\mathcal{S}$ by replacing each $Res_{out}.\mathsf{A}$ by the value of witness in 2QBF;
17 **else return** unknown ;

Step 4 only guarantees *soundness*: If the formula is satisfiable, then the specification is realizable (line 15, 16). Otherwise, unknown is returned (line 17).[5]

As each individual specification of one of the types {P1, P2, P3, P4} is trivially realizable, the reason for rejecting a specification is (1) simultaneous true and false demanded by different sub-specifications, (2) violation of properties over output variables (type P5), and (3) feedback loop within $\mathcal{S}$. Therefore, as Steps 1-4 guarantees non-existence of above three situations, the presented method is *sound*.

Theorem 1. (Soundness) *Let ϕ be a GXW specification, and $\mathcal{S}$ be an actor-based controller as generated by Steps 1-4 from ϕ; then $\mathcal{S}$ realizes ϕ.*

[5] Even without unroll, one can infer relations over universal variables via statically analyzing the specification. As an example, consider two sub-specifications $S1$: $\mathbf{G}(\mathsf{in1} \rightarrow (\mathsf{out}\,\mathbf{W}\,\mathsf{in2}))$ and $S2$: $\mathbf{G}(\mathsf{in2} \rightarrow (\neg\mathsf{out}\,\mathbf{W}\,\mathsf{in1}))$. One can infer that it is impossible for $TrUB^{(1)}$ and $TrUB^{(2)}$ to be simultaneously have state variable $lock =$ true, as both starts with $lock =$ false, and if $S1$ first enters lock ($lock =$ true) due to in1, the $S2$ cannot enter, as release part of $S2$ is also in1. Similar argument follows vice versa.

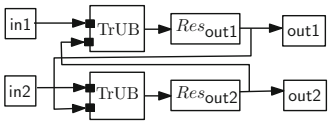

Fig. 7. Incompleteness example.

The GXW synthesis algorithm as described above, however is *incomplete*, as controllers with feedback loops are rejected; that is, whenever output variables listed in the release part of P2 necessitate simultaneous reasoning over two or more output variables. Figure 7 display a controller (with feedback loop) for realizing the specification $\mathbf{G}(\mathsf{in1} \rightarrow (\mathsf{out1}\mathbf{W}\mathsf{out2})) \wedge \mathbf{G}(\mathsf{in2} \rightarrow (\mathsf{out2}\mathbf{W}\mathsf{out1}))$. However, our workflow rejects such a controller even though the given specification is realizable. With further structural restriction over GXW (which guarantees no feedback loop in during construction) and by using unrolling of the generated actor-based controllers, the workflow as presented here can be made to be *complete*, as demonstrated in Sect. 4.4.

4.4 General Properties for GXW Synthesis

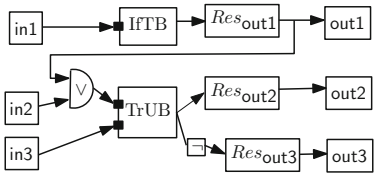

Fig. 8. Control implementation.

Since unrealizability of a GXW specification is due to the conditions (1) simultaneous true and false demanded by different sub-specifications, and (2) violation of properties over output variables (type P5)[6], one can build a counter-strategy[7] by first building a tree that provides input assignments to lead all runs to undesired states violating (1) or (2), then all leafs of the tree violating (1) or (2) are connected to a self-looped final state, in order to accept ω-words. As the input part listed in Table 2 does not involve any output variable, a counter-strategy, if exists, can lead to violation of (1) or (2) within Ω cycles, where Ω is a number sufficient to let each input part of the sub-specification be true in a run.

Lemma 1. For GXW specification $\varrho \rightarrow \bigwedge_{m=1\ldots k} \eta_m$, if (a) ρ^0_{out} is false for all η_m of type P2 and (b) no specification of type P5 exists, then if the specification is not realizable, then there exists a counter-strategy which leads to violation of (1) or (2) in Ω steps, where Ω is bounded by the sum of (i) the number of specifications k, and (ii) the sum of all i value defined within each ϕ^i_{in} of η_m.

When ρ^0_{out} is false for all η_m of type P2, our presented construction guarantees no feedback loop. As no specification of type P5 exists, the selection of

[6] Rejecting feedback loops on the controller structure is only a restriction of our presented method and is not the reason for unrealizability; similar to Fig. 7, feedback loop can possibly be resolved by merging all actors involving feedback to a single actor.

[7] A counter-strategy in LTL synthesis a state machine where the environment can enforce to violate the given property, regardless of all possible moves by the controller [27].

A never influences whether the specification is realizable. Therefore, quantifier alternation is removed. To this end, checking unrealizability is equivalent to non-deterministically guessing Ω input assignments and subsequently, checking if a violation of (1) or (2) appears by executing $\mathcal{S}$. This brings the co-NP result stated in Lemma 2. This also means that under the restriction from Lemma 1, a slight modification of Step 4 to perform unrolling the computation Ω-times makes our synthesis algorithm *complete*.

Lemma 2. Deciding whether a given GXW specification, which also obeys the additional restrictions as stated in Lemma 1, is realizable or not is in co-NP.

For the general case, the bound in Lemma 1 remains valid (as input part is not decided by the output variable). Complexity result is achieved by, without using our construction, directly using finite memory to store and examine all possible control strategies in Ω steps.

Lemma 3. For GXW specification $\varrho \rightarrow \bigwedge_{m=1\ldots k} \eta_m$, if the specification is not realizable, then there exists a counter-strategy which leads to violation of (1) or (2) in Ω steps, where Ω is bounded by condition similar to Lemma 1.

Lemma 4. Deciding whether a given GXW specification is realizable or not is in PSPACE.

The above mentioned bounds are only conditions to detect realizability of a GXW specification, while our presented workflow in Sect. 4 targets generating structured implementations. Still, by unrolling the computation Ω-times, one can detect if a controller, following our regulated structure, exists.

4.5 Extensions

One can extend the presented workflow to allow richer specification than previously presented GXW fragment. Here we outline how these extensions are realized by considering the following sample specification: $\mathbf{G}(\mathsf{in1} \rightarrow \mathsf{out1}) \wedge \mathbf{G}((\mathsf{in2} \vee \mathsf{out1}) \rightarrow ((\mathsf{out2} \wedge \neg\mathsf{out3})\mathbf{W}\,\mathsf{in3}))$. The SDF controller implementation is shown in Fig. 8. First, conjunctions in ϱ_{out} can be handled by considering each output variable separately. E.g., for $\varrho_{out} \equiv \mathsf{out2} \wedge \neg\mathsf{out3}$, in Fig. 8 both are connected to the same TrUB. Second, the use of output variables in "input" part for pattern P1, P2, P3 is also supported, provided that in effect a combinatorial circuit is created (i.e., output variables should always proceed with $\mathbf{X}^i$), and the generated system does not create a feedback loop. E.g., for the antecedent $(\mathsf{in2} \vee \mathsf{out1})$, it is created by wiring the Res_{out1}.out to an OR-gate.

5 Experimental Evaluation

We implemented a tool for GXW synthesis in Java, which invokes DepQBF [21] (Version 5.0) for QBF solving. Table 3 includes experimental results for a representative subset of our PLC benchmark examples. Execution times is recorded

using Ubuntu VM (Virtual Box with 3 GB RAM) running on an Intel i7-3520M 2.9 GHz CPU and 8 GB RAM). Most control problems are solved in less than a second[8]. GXW synthesis always generated a controller without feedback loops for all examples.

Table 3 lists a comparison of execution times of GXW synthesis and the bounded LTL synthesis tool Acacia+ [8] (latest version 2.3). We used the option `--player 1` of Acacia+ for forcing the environment to take a first move, but we did not do manual annotation in order to support compositional synthesis in Acacia+, as it is not needed by our tool. For many of the simpler case studies, the reported runtimes of Acacia+ are similar to GXW synthesis. However, GXW seems to scale much better to more complex case studies with a larger number of input and output variables such as examples 5, 9, 11, 12, 13, 15, 16, 17, 18, 19 in Table 3. The representation of the generated controller in terms of a system of interacting actors in GXW synthesis, however, allows the engineer to trace each sub-specification with corresponding partial implementation. In fact the structure of the controllers generated by GXW is usually similar to reference implementations by the case study providers. In contrast, a controller expressed in terms of single Mealy machine is rather difficult to grasp and to maintain for problems such as example 18 with 13 input and 13 output variables.

6 Related Work

Here we compare GXW synthesis with related GR(1) synthesis (e.g., [5,15,25,31]) and bounded LTL synthesis (e.g., [8,11,28]) techniques.

Synthesis for the GR(1) fragment of LTL is in time polynomial to the number of nodes of a generated game, which is PSPACE when considering exponential blow-up caused by input and output variables. GXW is also in PSPACE, where GXW allows **W** and GR(1) allows **F**. Even though it has been demonstrated that the expressiveness of GR(1) is enough to cover many practical examples, the use of an until logical operator, which is not included in GR(1), proved to be essential for encoding a majority of our PLC case studies. Also, implementations of GR(1) synthesis such as Anzu [15] do not generate structured controllers. Since GR(1) synthesis, however, includes a round-robin arbiter for circulating among sub-specifications, the systematic structuring of controllers underlying GXW synthesis may be applicable for synthesizing structured GR(1) controllers.

Bounded synthesis supports full LTL and is based on a translation of the LTL synthesis problem to safety games. By doing so, one solves the safety game and finds smaller controllers (as demonstrated in synthesis competitions via tools like Simple BDD solver [13], AbsSynthe [9], Demiurge [17]). The result of solving safety games in bounded LTL synthesis usually is a monolithic Mealy (or Moore) machine, whereas our GXW synthesis method of creating SDF actors may be understood as a way of avoiding the expensive construction of the product of machines. Instead, we are generating controllers by means of wiring smaller

[8] Approximately 0.25 seconds is used for initializing JVM in every run.

sub-controllers for specific monitoring and event triggering tasks. The structure of the resulting controllers seem to be very close to what is happening in practice, as a number of our industrial benchmark examples are shipped with reference implementation which are usually structured in a similar way. The size of the representations of generated controllers is particularly important when considering resource-bounded embedded computing devices such as a PLCs. LTL component synthesis, however, has the same worst-case complexity as full LTL synthesis [22].

Table 3. Experimental Result

ID	Description	Source	I/O vars	GXW time (s)	Acacia+ time (s)
1	Automatic door	Ex15 [2]	(4,3)	0.389	0.180
2	Simple conveyor belt	Ex7.1.19 [24]	(3,3)	0.556	0.637
3	Hydraulic ramp	Ex7.1.3 [24]	(5,2)	0.642	0.451
4	Waste water treatment V1	Ex7.1.8 [24]	(6,3)	0.471	0.323
5	Waste water treatment V2	Ex7.1.9 [24]	(8,9)	0.516	5.621
6	Container fusing	Ex10 [2]	(7,6)	0.444	0.425
7	Elevator control mixing plant	Ex7.1.4 [24]	(10,5)	0.484	2.902
8	Lifting platform	Ex21 [16]	(6,3)	0.350	0.645
9	Control of reversal	Ex36 [16]	(7,7)	0.395	2.901
10	Gear wheel	Ex19 [16]	(4,6)	0.447	0.302
11	Two directional conveyor (simplified)	Ex7.1.31.1 [24]	(9,5)	0.789	6.552
12	Garage door control	Ex7.1.25 [24]	(13,5)	0.574	7.002
13	Contrast agent injection	Ex7.1.18 [24]	(6,8)	0.458	3.209
14	Identification	Ex39 [16]	(5,5)	0.430	0.392
15	Monitoring chain elevator	Ex7.1.15 [24]	(10,9)	0.429	9.647
16	Two directional conveyor	Ex7.1.31.1 [24]	(12,5)	0.890	51.553
17	Control of single torque drive (simplified)	Ex7.1.26 [24]	(12,8)	0.538	38.010
18	Gravel transportation via 3 conveyors (simplified)	Ex7.1.31.4 [24]	(13,13)	1.227	> 600 (t.o.)
19	Control of two torque drives (simplified)	Ex7.1.26 [24]	(22,16)	0.790	> 600 (t.o.)

7 Conclusion

We have identified a useful subclass GXW of LTL for specifying a large class of embedded control problems, and we developed a novel synthesis algorithm (in PSPACE) for automatically generating structured controllers in a high-level programming language with synchronous dataflow without cycles. Our experimental results suggest that GXW synthesis scales well to industrial-sized control problems with around 20 input and output ports and beyond.

In this way, GXW synthesis can readily be integrated with industrial design frameworks such as CODESYS [3], Matlab Simulink, and Ptolemy II, and the generated SDF controllers (without cycles) can be statically scheduled and implemented on single and multiple processors [18]. It would also be interesting to use

our synthesis algorithms to automatically generate control code from established requirement frameworks for embedded control software such as EARS [23]. Moreover, our presented method supports traceability between specifications and the generated controller code as required by safety-critical applications. Traceability is also the basis for an incremental development methodology.

One of the main impediments of using synthesis in engineering practice, however, is the lack of useful and automated feedback in case of unrealizable specifications [6,10,20] or realizable specifications with unintended realizations. The use of a stylized specification languages such as GXW seems to be a good starting point for supporting design engineers in identifying and analyzing unrealizable specifications, since there are only a relatively small number of potential sources of unrealizability in GXW specifications. Finally, hierarchical SDF may also be useful for modular synthesis [30].

Acknowledgement. We thank Lacramioara Aştefănoaei for her fruitful feedback during the development of the paper, and CAV reviewers for their constructive comments. This work is supported by the H2020 project openMOS, GA no. 680735.

References

1. Full version available at http://arxiv.org/abs/1605.01153
2. Online training material for PLC programming. http://plc-scada-dcs.blogspot.com/
3. CODESYS - industrial IEC 61131–3 programming framework. http://www.codesys.com/
4. Matlab Simulink. http://www.mathworks.com/products/simulink/
5. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – a new requirements analysis tool with synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010)
6. Bloem, R., Ehlers, R., Jacobs, S., Knighofer, R.: How to handle assumptions in synthesis. In: SYNT, pp. 34–50 (2014). EPTCS 157
7. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: runtime enforcement for reactive systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 533–548. Springer, Heidelberg (2015)
8. 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)
9. Brenguier, R., Prez, G.A., Raskin, J.-F., Sankur, O.: AbsSynthe: abstract synthesis from succinct safety specifications. In: SYNT, pp. 100–116 (2014). EPTCS 157
10. Cheng, C.-H., Huang, C.-H., Ruess, H., Stattelmann, S.: G4LTL − ST: automatic generation of PLC programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 541–549. Springer, Heidelberg (2014)
11. Ehlers, R.: Unbeast: symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
12. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Sachs, S., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)

13. Jacobs, S., Bloem, R., Brenguier, R., Ehlers, R., Hell, T., Knighofer, R. Prez, G.A., Raskin, J.-F., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The first reactive synthesis competition. In: SYNTCOMP 2014 (2014). http://arxiv.org/abs/1506.08726
14. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE (2006)
15. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: a tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
16. Kaftan, J.: Praktische Beispiele mit AC500 von ABB: 45 Aufgaben und Lsungen mit CoDeSys (2014). http://pwww.kaftan-media.com/. ISBN 978-3-943211-05-4
17. Knighofer, R., Seidl, M.: Demiurge 1.2: A SAT-Based Synthesis Tool. Tool description for the SyntComp 2015 competition. http://www.iaik.tugraz.at/content/research/opensource/demiurge/
18. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)
19. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
20. Li, W.-C.: Specification mining: new formalisms, algorithms and applications. Ph.D. thesis. UC Berkeley (2015)
21. Lonsing, F., Biere, A.: DepQBF: a dependency-aware QBF solver. J. Satisfiability Boolean Model. Comput. **7**, 71–76 (2010)
22. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. STTT **15**(5–6), 603–618 (2013)
23. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy Approach to Requirements Syntax (EARS). In: RE, pp. 317–322. IEEE (2009)
24. Petry, J.: IEC 61131–3 mit CoDeSys V3: Ein Praxisbuch fuer SPS-Programmierer. Eigenverlag 3S-Smart Software Solutions. ISBN 978-3-000465-08-6 (2011)
25. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)
26. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
27. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. IEEE (1989)
28. 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)
29. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
30. Tripakis, S., Bui, D., Geilen, M., Rodiers, B., Lee, E.A.: Compositionality in synchronous data flow: modular code generation from hierarchical SDF graphs. ACM Trans. Embed. Comput. Syst. **12**(3), 83:1–83:26 (2013). http://doi.acm.org/10.1145/2442116.2442133, articleno 83, ISSN = 1539-9087
31. Wong, K.-W., Ehlers, R., Kress-Gazit, H.: Correct high-level robot behavior in environments with unexpected events. In: Robotics: Science and Systems X (RSS X) (2014)