Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Uppsala, Sweden, April 22-29, 2017

autoCode4: Structural Controller Synthesis

Chih-Hong Cheng[1], Edward A. Lee[2], and Harald Ruess[1]

[1] fortiss - An-Institut Technische Universität München, Germany
[2] EECS, UC Berkeley, USA

Abstract. autoCode4 synthesizes structured reactive controllers from realizable specifications in the GXW subset of linear temporal logic (LTL). Generated reactive controllers are expressed in terms of an intermediate synchronous dataflow (SDF) format, which is further translated, using an open interface, into SCADE/Lustre and Ptolemy II. Moreover, autoCode4 generates and maintains a traceability relation between individual requirements and generated code blocks, as mandated by current standards for certifying safety-critical control code.

1 Introduction

autoCode4 synthesizes structured and certifiable reactive controllers from a given realizable specification in the GXW [5] subset of linear temporal logic (LTL). It is released under the LGPLv3 open source license, and can be downloaded, including a hands-on tutorial, from

`http://autocode4.sourceforge.net`

autoCode4 is based on structural recursion of GXW input formulas [5] for generating synchronous dataflow (SDF) [12] controllers composed from a set of actors, and for maintaining the traceability between given specifications and the generated code blocks. The underlying synthesis algorithm [5] of autoCode4 differs considerably from previous approaches and tools for reactive synthesis [16, 11, 7, 17, 4, 14]. In contrast to these prevailing automata-based approaches, autoCode4 generates a reactive controller in a structured actor language with high-level behavioral constructs and synchronous dataflow communication between connected actors. This choice of generating *structured controllers* is motivated by the fact that a subset of SDF is compatible with the underlying model of computation for state-of-the-practice design tools including LabVIEW[3] and SCADE[4]. Indeed, autoCode4 includes pre-defined code generators for Lustre/SCADE and for Ptolemy II [9], where C code or a hardware description, say, in Verilog, can be generated subsequently. Structured SDF controllers also support the integration of manually designed or legacy elements. Furthermore, the structure of the generated SDF controller is often instrumental in pinpointing and isolating problematic (e.g. realizable but demonstrating undesired behavior) specifications for validating requirements.

autoCode4 supports the necessary interfaces for integration into existing development tool chains. In this way, autoCode4 has been embedded into the Ptolemy II [9] platform for the design, simulation, and code generation of cyber-physical systems. The open interfaces of autoCode4 are suitable for realizing additional code generators

[3] `http://www.ni.com/labview`
[4] `http://www.ansys.com/Products/Embedded-Software/ANSYS-SCADE-Suite`

for, say, Matlab Simulink[5] or continuous function charts (IEC 61131-3). Moreover, requirement specification languages for embedded control systems, such as EARS [8], may be translated to the GXW input language of autoCode4 .

autoCode4 is unique among reactive synthesis tools in that it maintains the traceability between individual requirements and the generated controller code blocks. Such a traceability relation is mandated by current industrial standards for safety-related developments such IEC 61508 (e.g. industrial automation), DO-178C (aerospace), and ISO-26262 (automotive).

2 Structural Synthesis in a Nutshell

autoCode4 uses the GXW subset of linear temporal logic (LTL) as defined in [5] for specifying the input-output behavior of reactive controllers. This specification language supports a conjunction of input assumptions, invariance conditions on outputs, transition-like reactions of the form $\mathbf{G}(\mathsf{input} \rightarrow \mathbf{X}^i \mathsf{output})$, and 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 (e.g. falling edge $(\mathsf{in} \wedge \mathbf{X}\neg\mathsf{in})$). 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. Output response to input events in GXW is *immediate* and, whenever an event occurs the specification excludes choices of the controller to select among multiple output options (such as setting either out1 or out2 to be true).

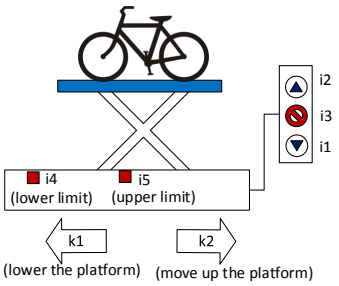

Fig. 1. A hydraulic lifting platform

The design of the GXW language has been guided by expressiveness, useability, and complexity considerations. We demonstrated the expressiveness of GXW by encoding of a large and rather diverse set of (Booleanized) control specifications from the domain of industrial automation [1, 2]. On the other hand, reactive synthesis for GXW is in PSPACE [5] compared to 2EXPTIME for full LTL. Moreover, the restrictions of GXW support the control designer by excluding non-causal, and commonly unrealizable, specifications, where output assignments are supposed to depend on future input values.

Structural synthesis [5] generates synchronous data flow [12] controllers from LTL specifications. Hereby, a controller is structurally composed from a set of interacting actors (behavior expressed in terms of Mealy machines). One of the main steps of the structural synthesis [5] involves checking for potentially conflicting constraints, from

[5] http://www.mathworks.com/products/simulink/

```
## Driving a hydraulic lifting platform
## S0: When i2 is pressed, proceed upwards
[] ((i2 && !i1 && !i3) -> (k2 W (i5 || i1 || i3 || (i1 && i2))))
## S1: When i1 is pressed, proceed downwards
[] ((i1 && !i2 && !i3) -> (k1 W (i4 || i2 || i3 || (i1 && i2))))
## S2: forbidden to drive the motor both upwards and downwards
[] (!(k1 && k2))
## S3: When reaching the upper limit, disallow upward movement
[] (i5 -> !k2)
## S4: When reaching the lower limit, disallow downward movement
[] (i4 -> !k1)
## S5: When both up & down button are pressed, disallow movement
[] ((i1 && i2) -> (!k1 && !k2))
## When stop is pressed, disallow movement
[] (i3 -> (!k1 && !k2))
## S7: Don't do any movement until i1 or i2 is pressed
((!k1 && !k2) W (i1 || i2))

INPUT i1, i2, i3, i4, i5
OUTPUT k1, k2
```

Fig. 2. GXW specification for hydraulic lifting platform.

multiple sub-specifications, for the output assignments, and determining satisfiability of given invariance properties.

3 autoCode4 in Action

We illustrate the use of GXW by means of the simple example of controlling a hydraulic ramp as shown in Fig. 1 (cmp. with [2], Section 7.1.3, for similar scenarios). By pressing button i2 (up) the motor enables upward movement of the platform by setting output variable k2 to true. Button i3 (stop) stops the movement, and i1 (down) triggers the lowering of the hub by setting output variable k1 to true. Sensors i5 and i4 are used for detecting upper and lower limits. If i1 and i2 are simultaneously pressed, one needs to stop the movement. Finally, simultaneous triggering the motor in both directions (i.e., $\mathbf{G}\neg(\mathsf{k1} \wedge \mathsf{k2})$) is disabled. The corresponding GXW specification is depicted in Fig. 2. Lines starting with "##" are comments, and a total of 8 GXW sub-specifications are listed. Sub-specifications are (implicitly) labeled, from top-to-bottom, by the indices 0 to 7.

For the hydraulic lifting platform, Fig. 3 shows the resulting SDF controller synthesized under Ptolemy II. One may run interactive simulation or further use code generation features in Ptolemy II to automatically generate executable code in C or HDL. Due to space limits, we do not show the control structure within each block; instead we refer the reader to Fig. 4 for the corresponding Lustre implementation.

Now, we shortly comment on the requirement-to-implementation traceability using sub-specification S7: $(\neg\mathsf{k1} \wedge \neg\mathsf{k2})\mathbf{W}(\mathsf{i1} \vee \mathsf{i2})$ (the tool also allows generating a traceability report). In Fig. 3, an or-gate called event7 connects i1 and i2. The output of event7 is fed into an InUB-typed actor called Ctrl_7. The output of Ctrl_7 is negated (via Not_7k1 and Not_7k2) to influence output variables k1 and k2 respectively. One can observe that the specification index "7" can be identified in above mentioned blocks due to the naming convention.

autoCode4 may also generate reactive controllers in Lustre [13]. Fig. 4 includes the Lustre v4 code generated from controlling the hydraulic lifting platform. The requirement-to-implementation traceability is similar to the Ptolemy II graphical representation in

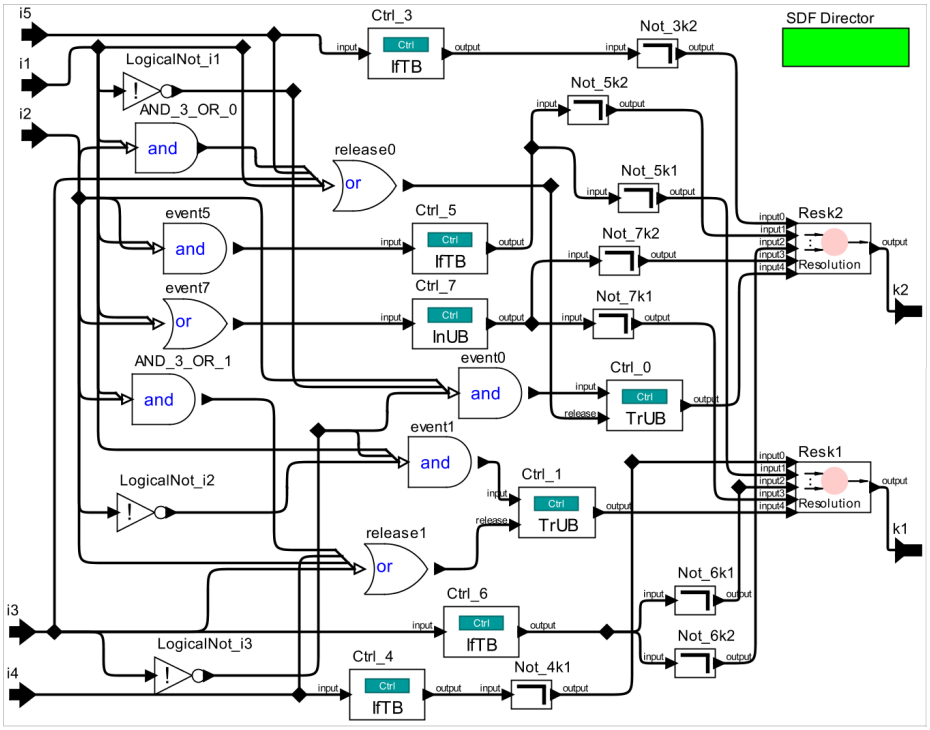

Fig. 3. Control realization as Ptolemy II models

Fig. 3. Notice that parameterized blocks such as `Res5` in Fig. 4 are instantiated twice. This kind of block reuse makes textual representations of generated code (i.e., define once, instantiate multiple times) highly compact.

4 autoCode4 Software Architecture

The software architecture of autoCode4 is depicted in Fig. 5 and follows the general outline of the structural synthesis algorithm as described [5]. Input specifications are analyzed and categorized by the specification analyzer, which also rejects non-GXW specifications. In our running example, `S0` and `S1` are of type TrUB (when event A triggers, do B until event C). `S3`, `S4` and `S5` are categorized as IfTB (when event A triggers, do B), `S7` is InUB (initially, do A until B), and lastly, S2 is an invariance property.

Subsequently, Constraint Builder builds the corresponding SDF structure (via SDF Builder), which is not fully instantiated yet, and constructs a quantified Boolean formula with one quantifier alternation (2QBF) for resolving potential conflicts between individual sub-specifications. In this process, each sub-specification is associated with a set of actors; for example, formula `S7` is associated with actors such as event7 or Ctrl_7. The engine can hash a set of actors that was instantiated previously, to enable actor reuse among multiple sub-specifications. Moreover, blocks such as `Res5` in Fig. 4

```
node TrUB (input, release: bool) returns
(output: int);
var lock: bool;
let lock = if input and not release then true
        else if release then false
        else false -> pre(lock);
  output = if input and not release then 1
         else if release then 2
         else 2 -> if pre(lock) then 1 else 2 ;
tel
```

```
node InUB (release: bool) returns (output: int);
var lock: bool;
let
 lock = if release then false else true ->
pre(lock);
 output = if release then 2
        else 1 -> if pre(lock) then 1 else 2 ;
```

```
node IfTB (input: bool) returns (output: int);
let  output = if input then 1 else 2;       tel
```

```
node TernaryNot (input: int) returns (output:
int);
let output = if input = 1 then 0
     else if input = 0 then 1 else input;
tel
```

```
node Res5 (input0, input1, input2, input3, input4: int; A: bool) returns (output: bool);
let
 output = if input0 = 1 or input1 = 1 or input2 = 1 or input3 = 1 or input4 = 1 then true
     else if input0 = 0 or input1 = 0 or input2 = 0 or input3 = 0 or input4 = 0 then false
     else A;
tel
```

```
node GXWcontroller(Gli1, Gli2, Gli3, Gli4, Gli5 : bool) returns (GOk1, GOk2: bool);
var
 Ctrl_4, Not_7k2, Ctrl_5, Ctrl_6, Not_6k1, Not_5k2, Ctrl_7, Not_7k1, Not_6k2,
 Ctrl_0, Not_4k1, Not_3k2, Ctrl_1, Not_5k1, Ctrl_3: int;
 Resk2, Resk1, event1, event0, release1, release0, event7, event5,
 AND_3_OR_0, AND_3_OR_1: bool;
let
 event0 = (Gli2 and (not Gli1) and (not Gli3));
 event1 = (Gli1 and (not Gli2) and (not Gli3));
 AND_3_OR_0 = (Gli1 and Gli2); release0 = (AND_3_OR_0 or Gli5 or Gli1 or Gli3);
 AND_3_OR_1 = (Gli1 and Gli2); release1 = (AND_3_OR_1 or Gli4 or Gli2 or Gli3);
 event7 = (Gli1 or Gli2);  event5 = (Gli1 and Gli2);
 Ctrl_0 = TrUB(event0 , release0); Ctrl_1 = TrUB(event1 , release1);
 Ctrl_3 = IfTB(Gli5); Ctrl_4 = IfTB(Gli4);
 Ctrl_5 = IfTB(event5); Ctrl_6 = IfTB(Gli3); Ctrl_7 = InUB(event7);
 Not_3k2 = TernaryNot(Ctrl_3);  Not_4k1 = TernaryNot(Ctrl_4);
 Not_5k1 = TernaryNot(Ctrl_5);  Not_5k2 = TernaryNot(Ctrl_5);
 Not_6k1 = TernaryNot(Ctrl_6);  Not_6k2 = TernaryNot(Ctrl_6);
 Not_7k2 = TernaryNot(Ctrl_7);  Not_7k1 = TernaryNot(Ctrl_7);
 Resk1 = Res5(Not_4k1, Ctrl_1, Not_5k1, Not_6k1, Not_7k1, false); GOk1 = Resk1;
 Resk2 = Res5(Ctrl_0, Not_3k2, Not_7k2, Not_5k2, Not_6k2, false); GOk2 = Resk2;
tel
```

Fig. 4. Control realization in Lustre format

are used for determining an output value `A` if none of the sub-specifications constrain such an output.

The generated 2QBF constraint is analyzed for potential conflicts on certain output variables from different sub-specifications, and for analyzing invariance properties. These constraints are of the form: $\exists\, \mathtt{A}_1, \ldots, \mathtt{A}_k\ \forall$ system state s: (Env-Assumption(s) $\wedge$ SDF-dynamics(s)) $\rightarrow$ (No-output-conflict(s) $\wedge$ Invariance(s)). The open design choices are determined by witnesses for the existentially-quantified variables $\mathtt{A}_1$, ... , $\mathtt{A}_k$, as specified in the previous step. Informally, the body of this formula encodes the condition: if the environment assumptions hold and the system adheres to the semantics of the SDF dynamics then there is no output conflict and the specified invariance properties hold. Our 2QBF solver implements an algorithm for alternating two Boolean satisfiability solvers [10] as in SAT4J [3]. It takes the negation of the constraint above with forall-exists top-level quantification. Therefore, whenever the 2QBF solver returns false, the generated counterexample determines a non-conflicting assignment for the uninstantiated variables. Using the definition of `Resk1` and `Resk2` in Fig. 4 in our running example, the variable `A` is set to false in both cases.

The internal SDF controller is stored using the class SDFctrl, and can be traversed for generating control code in Ptolemy II, Lustre, and other SDF-based lan-

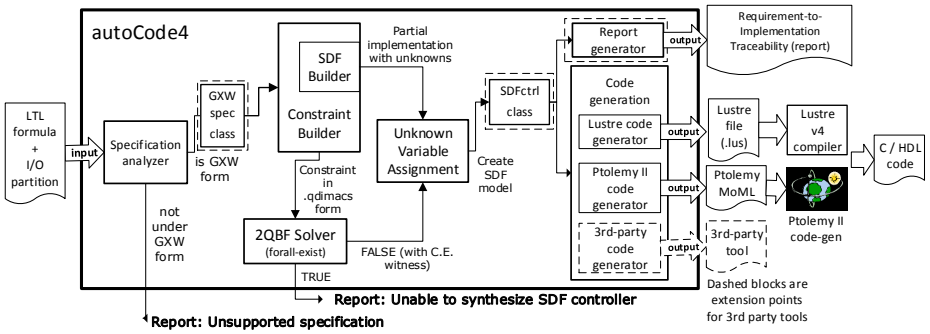

Fig. 5. System architecture of autoCode4 , where dashed elements are extension points.

guages. Finally, autoCode4 uses the Report generator for producing a requirement-to-implementation traceability report.

References

1. Online training material for PLC programming. http://plc-scada-dcs.blogspot.com/.
2. CODESYS - Industrial IEC 61131-3 PLC programming framework http://www.codesys.com/.
3. D. Le Berre, Daniel, and A. Parrain. The SAT4J library, release 2.2, system description. Journal on Satisfiability, Boolean Modeling and Computation (7): 59-64 (2010).
4. A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin. Acacia+, a tool for LTL synthesis. In: *CAV*, volume 7358 of *LNCS*, pages 652–657. Springer, 2012.
5. C.-H. Cheng, Y. Hamza, H. Ruess: Structural Synthesis for GXW Specifications. In: *CAV*, volume 9779 of *LNCS*, pages 95–117. Springer, 2016.
6. C.-H. Cheng, C.-H. Huang, H. Ruess, and S. Stattelmann. G4LTL-ST: Automatic Generation of PLC Programs. In: *CAV*, volume 8559 of *LNCS*, pages 541–549. Springer, 2014.
7. R. Ehlers. Unbeast: Symbolic Bounded Synthesis. In: *TACAS*, volume 6605 of *LNCS*, pages 272–275. Springer, 2011.
8. A. Mavin, P. Wilkinson, A. Harwood, M. Novak: Easy Approach to Requirements Syntax (EARS). In: *RE*, pages 317–322. IEEE, 2009
9. C. Ptolemaeus (ed.). System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014).
10. M. Janota, J. Marques-Silva. Abstraction-based Algorithm for 2QBF. In: *SAT*, volume 6695 of *LNCS*, pages 230–244. Springer, 2011.
11. B. Jobstmann, . Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for Property Synthesis. In: *CAV*, volume 4590 of *LNCS*, pages 258–262. Springer, 2007.
12. E. A. Lee and D. G. Messerschmitt: Synchronous Data Flow. Proceedings of the IEEE, 75(9):1235-1245 (1987).
13. N. Halbwachs, P. Caspi, P. Raymond,, D Pilaud. The Synchronous Data Flow Programming Language LUSTRE. Proceedings of the IEEE, 79(9):1305-1320 (1987).
14. N. Piterman, A. Pnueli, Y. Sa'ar. Synthesis of Reactive(1) Designs. In: *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.
15. A. Pnueli. The Temporal Logic of Programs. In: *FOCS*, pages 46–57. IEEE, 1977.
16. A. Pnueli, R. Rosner. On the Synthesis of a Reactive Module. In: *POPL*, pages 179–190. IEEE, 1989.
17. S. Schewe and B. Finkbeiner. Bounded Synthesis. In: *ATVA*, volume 4762 of *LNCS*, pages 474–488. Springer, 2007.