Specification Format for Reactive Synthesis Problems*

Ayrat Khalimov
Graz University of Technology, Austria

Automatic synthesis from a given specification automatically constructs correct implementation. This frees the user from the mundane implementation work, but still requires the specification. But is specifying easier than implementing? In this paper, we propose a user-friendly format to ease the specification work, in particularly, that of specifying partial implementations. Also, we provide scripts to convert specifications in the new format into the SYNTCOMP format, thus benefiting from state of the art synthesizers.

1 Introduction

Specifying reactive synthesis tasks is not easy. First, writing non-trivial specifications in e.g. linear temporal logic (LTL) requires experience, and even an experienced user of LTL may notice that some properties are easier to implement oneself than to specify. Thus, it is desirable to be able to mix imperative and declarative paradigms when specifying reactive synthesis tasks, which makes a call for a new convenient specification format.

The full set of features of the new specification format might include:

1. *Modularity.* A synthesis task may require to synthesize several communicating modules where each module has its own properties. Thus, the new format should allow for specifying module interfaces and connections between them. These interfaces specify the amount of information each module knows about others.
2. *Imperative and declarative.* Some modules may already be given to the user, and some modules or parts of it may be easier to implement than to specify. Thus, the new format should allow for specifying module implementations.
3. *Conversion to the SYNTCOMP format.* The SYNTCOMP format [9] was recently proposed as the common ground format for reactive synthesis competitions, and at least four synthesizers were competing in 2014. Thus, to let the user to benefit from state of the art synthesizers, the new format should be convertible into the SYNTCOMP format.
4. *Property language agnostic.* The new format should allow the user to choose the best suited language for writing properties: linear temporal logic, linear dynamic logic [14], regular expressions, automata, etc.

These features requirements are our subjective suggestions and arise from the domain of synthesis of reactive systems that usually represent some hardware. The features certainly depend on the synthesis domain: for example, in the case of fault-tolerant algorithms the user also needs to specify the ratio of faulty to normal processes, the type of faults, etc.

In this the paper we:

- propose a specification format for reactive synthesis tasks, and

*This work was supported by the Austrian Science Fund via project RiSE (S11406).

Černý, Kuncak, Madhusudan (Eds.): Fourth Workshop on Synthesis (SYNT 2015)
EPTCS 202, 2016, pp. 112–119, doi:10.4204/EPTCS.202.8

- provide scripts to convert from the new format into the SYNTCOMP format.

The new format can be extended to support features (1), (2), (3), and (4), but the current version has limitations. Some of the limitations are: (i) the user can separate the system into modules, but each module has the full information about others, (ii) only deterministic Büchi automata are allowed for specifying properties, and (iii) assumptions must be safety properties.

The new format is based on the SMV format [7] – it is convenient for describing hardware systems: it allows the user to define finite state machines that operate on variables of enumeration and range types, and to separate the system into modules, etc. Another advantage of using the SMV format as the starting point is that there is a solid support of the SMV format in the AIGER distribution [1], which greatly simplifies the task of the development of the conversion scripts.

Outline. We describe the new format and its restrictions in Section 2. Section 3 describes the conversion scripts and also introduces the SYNTCOMP format extended with liveness which is one of the supported target formats (alongside the standard SYNTCOMP format). Section 4 illustrates the use of the format and of the scripts – we write the specification that describes the task: when given an implementation of a Huffman decoder for the English alphabet, synthesize an encoder for it. Section 5 points to other possible ways of writing specifications and converting them into the SYNTCOMP format. And we conclude in Section 6.

2 Specification Format

We assume that the reader is familiar with the SMV (cf. [7]) and the SYNTCOMP [9] formats. We introduce a new section into the SMV format, and the comments of special form that allow for specifying synthesis problems. The specification in the extended SMV format is then translated into the SYNTCOMP format.

An example of the extended SMV format is shown in Listing 1. [1]

As in the usual SMV format, it consists of modules and the main module. In the main module, variables to be controlled by the system are marked with the comment '`--controllable`' (Mealy-type). The new sections ENV_AUTOMATON_SPEC and SYS_AUTOMATON_SPEC contain definitions of the assumptions and guarantees respectively. Every assumption and guarantee in the corresponding sections is expressed by a file path to a Büchi automaton in the GOAL format [13]. A file path can be preceded by '!' to indicate that the property is the negation of the automaton. These property automata will be converted into SMV modules.

Restrictions

The framework we describe in Section 3 converts a given specification in the extended SMV format into a deterministic game in the AIGER circuit format. AIGER circuits are inherently deterministic and so should be automata used in sections SYS_AUTOMATON_SPEC and ENV_AUTOMATON_SPEC. We require that:

- guarantees automata are deterministic (or determinizable),
- assumptions automata represent safety properties.

These conditions are sufficient (but not necessary) for the game to be deterministic, and are required by the conversion script spec_2_aag.py described in Section 3.

[1] The format is under active development and may slightly differ from the one described here.

Listing 1: Format structure (special elements are in blue color).

```
MODULE helper1(input1,input2)  //we can define and use SMV modules as usually
VAR
  state: 0..100;
DEFINE
  reached42 := state=42;
  ...

MODULE main  // module 'main' contains a specification
VAR
  CPUread: boolean;   // only boolean is allowed

VAR --controllable
  valueOut: boolean;  // only boolean is allowed

VAR
  h: helper1(readA, valueOut);  // we can instantiate modules as usually

DEFINE
  //signals defined in the module can be referred to in the property automata
  a := TRUE;
  b := FALSE;

  writtenA := CPUwrite & valueIn=a & done;
  readA := CPUread & valueOut=a & done;
  is42 := h.reached42;
  ...
  // thus we can use variables 'is42', 'readA', 'writtenA' in property automata below

SYS_AUTOMATON_SPEC // guarantees in the GOAL automata format
  guarantee1.gff;
  !guarantee2.gff; // '!' signals to negate the automaton

ENV_AUTOMATON_SPEC // assumptions in the GOAL automata format
  assumption1.gff;
  !assumption2.gff;
  ...
```

3 Conversion into the SYNTCOMP Format

We will convert specifications in the extended SMV format into standard and extended SYNTCOMP formats. Specifications in the standard SYNTCOMP can be given to any synthesis tool from the SYNTCOMP competition. Specification in the extended format can either be converted into the standard SYNTCOMP format using justice_2_safety.py, or can be given to our synthesizer aisy.py that supports it.

The scripts are available at `https://bitbucket.org/art_haali/spec-framework`.

Standard and extended SYNTCOMP

In this section we remind what the standard SYNTCOMP format is and then introduce the extension.

The standard SYNTCOMP is a circuit in the old AIGER format [2] with special comments that allow for specifying controllable (by the system) and uncontrollable (thus controllable by the environment) signals. Figures 1 and 2 show the standard SYNTCOMP format [9] (ignore the dotted arrows – they are part of the extended format).

The goal is to synthesize the controllable signals (i.e., replace them with combinational circuits that as inputs use the memory and uncontrollable signals) such that the output *bad* never raises. Thus, the

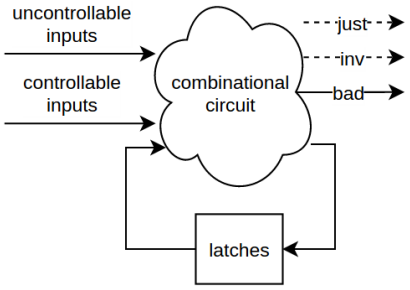


Figure 1: SYNTCOMP specification

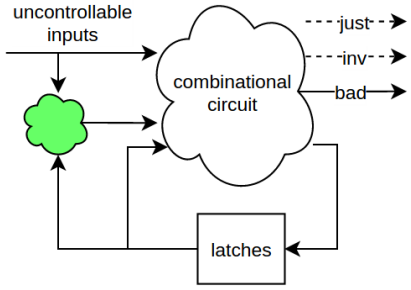


Figure 2: SYNTCOMP model

semantics of the standard SYNTCOMP is $\mathsf{G}\neg bad$, which allows for specifying safety properties.

The natural extension is to allow liveness properties. This is what the extended SYNTCOMP format proposes. It also uses signal *inv* though it does not add the expressiveness. These signals are 'introduced' using the standard capabilities of the new AIGER format [4] (which allows for specifying 'bad' signals, 'invariant' signals, and 'justice' signals). The extended SYNTCOMP is shown on the same figure as the standard one if you take into account the dotted signals.

The semantics of the extended SYNTCOMP format is

$$(\neg bad\,\mathsf{W}\,\neg inv) \wedge (\mathsf{G}\,inv \rightarrow \mathsf{GF}\,just) \tag{1}$$

Note: the meaning of the signal *just* is reversed compared to the new AIGER format [4]: in that case a witness liveness trace satisfies $\mathsf{G}\,inv \wedge \mathsf{GF}\,just$, while in our case it satisfies $\mathsf{G}\,inv \wedge \neg\,\mathsf{GF}\,just$. We reversed the meaning of the signal *just* to be able to specify properties like $\mathsf{G}(r \rightarrow \mathsf{F}\,g)$ ("every request is granted") or $\mathsf{GF}\neg r$ ("request is lowered infinitely often"). Such properties can be represented by deterministic Büchi automata but not by deterministic co-Büchi automata. And we need specification automata to be deterministic to be able to convert them into inherently deterministic AIGER circuits.

Converting specifications into SYNTCOMP

Figure 3 shows how we convert a given specification into SYNTCOMP format.

The main script is spec_2_aag.py:

1. Given a specification in extended SMV format (Section 2), we first convert all the automata in the GOAL format into SMV modules. At this step we might need to complement or determinize a

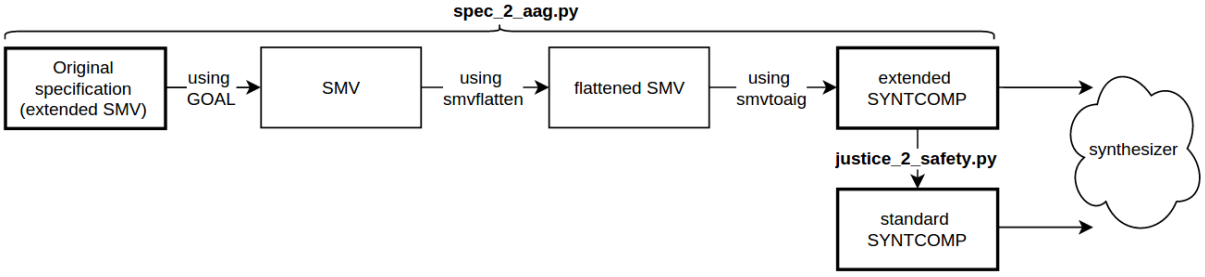


Figure 3: Converting specifications from our extended SMV format into the SYNTCOMP format

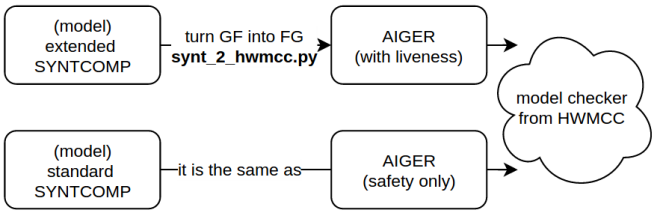
given automaton – this is done using GOAL. Then we parse the result and convert it into an SMV module. Such SMV module contains two special signals: *bad* and $fair$. In such SMV module, signal $fair$ is risen when we visit an accepting state of the automaton, and *bad* is risen when we visit a non-accepting state with a self-loop labelled $True$.

2. The main conversion work – from the SMV format into the extended SYNTCOMP format – is done with scripts smvflatten and svmtoaig from the AIGER distribution [1]. The result of this step is an AIGER file that may contain invariant and justice signals, which is not supported by the current SYNTCOMP format. Thus the current synthesis tools from the competition cannot be used directly.
3. The file in the extended SYNTCOMP format is converted into the standard version (with the single output) using justice_2_safety.py. The conversion requires input positive integer k and is standard: $\mathsf{GF}\ just$ is replaced with $\mathsf{G}(just \vee \mathsf{X}\ just \vee \ldots \mathsf{X}^k\ just)$, where X^k means k repetitions of X.

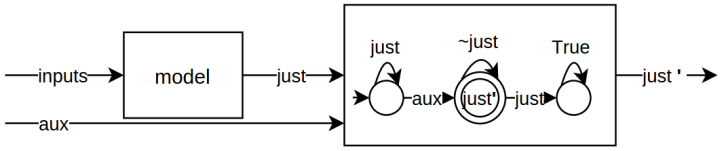
The result of this conversion is specification in either the standard or extended SYNTCOMP formats, and can be given to a synthesizer.

Converting models into AIGER

After the synthesizer produces a model, it can be turned into a benchmark in the standard AIGER format and then be fed to a model checker (e.g., one from the HWMCC competition):



If the input synthesis specification is in the standard SYNTCOMP format, then the model is also in the standard AIGER format and can be fed to a model checker directly. But in the case of the extended SYNTCOMP format we need to translate. Recall the semantics of our extended format (Equation 1): in our case a trace violating a liveness property would satisfy $\neg\mathsf{GF}\ just$, while the AIGER format has $\mathsf{GF}\ just'$. Thus, we convert the model into a model with signal $just'$ such that: if there is a trace that satisfies $\mathsf{GF}\ just'$ then it satisfies $\mathsf{FG}\neg just$. If denote the new model by M', and the original one by M, then: $M' \models \mathsf{EFG}\ just' \rightarrow M \not\models \mathsf{AGF}\ just$. The script synt_2_hwmcc.py does this by introducing a new input *aux* and attaching the automaton as shown below:



4 Example: Synthesizing a Huffman Encoder

This section demonstrates the use of the format and the framework. We implemented a simple synthesizer that solves Büchi games with invariants and safety objectives given in the extended SYNTCOMP format

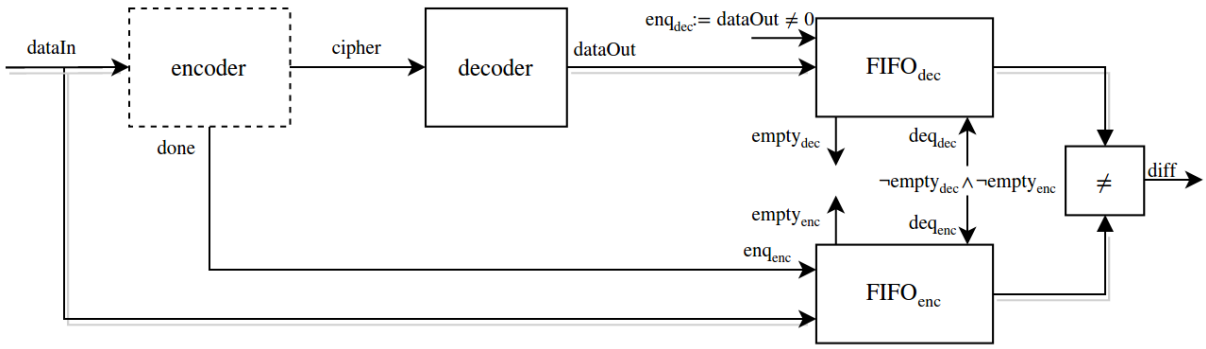


Figure 4: The structure of the SMV specificaton for a Huffman encoder

described in Section 2. The results of the synthesis are then translated into the HWMCC format using script synt_2_hwmcc.py, and then model checked with IIMC [5].

We use the Huffman coding [8] to encode 26 English letters $A...Z$ and the space symbol into bit words of variable length (27 symbols in total). Let us assume that a Huffman decoder that decodes a stream of bits into letters is given [2] — the goal is to synthesize an encoder that works with the decoder.

Figure 4 shows the structure of the SMV specification of the synthesis task.

The dotted module (encoder) is to be synthesized, namely signals *cipher* and *done* (these signals are marked 'controllable' in the specification). The input is *dataIn* and has five bits of width, which is enough to encode 27 symbol: we use numbers 1..27 for encoding the symbols. The outputs of the encoder are boolean signals *cipher* and *done*; the intended meaning of *done* is "the last bit of the cipher is being sent now". The signal *cipher* is read by the decoder, which decodes the cipher and outputs it over *dataOut*; on successful decoding *dataOut* lasts for one tick, after which it is 0 again. The data-signal *dataOut* is then fed to the FIFO module $FIFO_{dec}$, and $FIFO_{enc}$ takes as input *dataIn*. FIFOs values are dequeued whenever they are not empty, and their values are compared. $FIFO_{enc}$ is enqueued whenever *done* is high, and $FIFO_{dec}$ – whenever *dataOut* encodes a letter. A FIFO gets blocked if we enqueue and not dequeue, and the FIFO is not empty currently (i.e., if $enq \wedge \neg deq \wedge empty$ holds).

All modules except dotted module *encoder* are given: FIFOs we coded manually (of size 1); the decoder is taken from the distribution of the model checker VIS [6].

In words, the specification is:

A1. assumption: "input *dataIn* is within range 1..27"

A2. assumption: "*dataIn* does not change until and including the moment when *done* is high"

G1. $\mathsf{G}(done \rightarrow \mathsf{X}enq_{dec})$ [3]

G2. $\mathsf{G}\neg diff$, i.e., if FIFOs are not empty, then they contain the same data

G3. liveness guarantee: $\mathsf{GF}\,done$

The specification in the SMV format is translated into the SYNTCOMP formats (standard safety and extended liveness) as described in Section 2. The semantics is as given in Equation 1 where: *bad* is the violation of any of the safety guarantees, *inv* is the truth of (A1) and (A2) so far, and $just = done$.

[2] Thus the decoder already has the letter frequencies built in.

[3] Strictly speaking this guarantee is not needed for the correct synthesis of the encoder, but without it the meaning of *done* may be different from the intended one ("the last cipher bit is being transferred").

Given the specification in the extended SYNTCOMP format, the synthesizer aisy.py synthesized the model in ≈ 2 minutes; the model has $\approx$ 130k new AND-gates [4]. The cipher synthesized is as expected (coincides with that of the Huffman decoder).

If we translate the specification into the k-safety variant with $k = 10$ (the minimal realizable), then aisy.py needs ≈ 4 minutes for the synthesis and the model has $\approx$ 120k AND-gates. We do not claim that in terms of efficiency the liveness specifications are superior to their safety variant – for this a more thorough research is needed. But the translation of liveness into safety requires a value of k as input: here we provided it manually, while in the general case its upper bound should be restricted and the permitted values should be iterated in some way.

Some final notes on the example. Initially, FIFOs implementations were non blocking, which permits the synthesizer to produce a cipher for a letter that is prefixed with ciphers of other letters (this version of the specification would compare only the last decoded letter). Also, with non-blocking FIFOs and without guarantee G2, the synthesizer produced a cipher that utilized the overflow in the state variable of the decoder. Hence in the general case the synthesized cipher may depend on in the implementation of the decoder and will not work with other implementations.

The benchmarks are available as a part of the conversion scripts distribution; aisy.py is available at `https://bitbucket.org/art_haali/aisy`.

5 Related Work

There are scripts and ways to create specification circuits in the SYNTCOMP format:

The script ltl2aig [10] takes as input specification in LTL format and signals partition and converts it into a circuit in the standard SYNTCOMP format. It does not use tools from the AIGER distribution [1] and supports all the routines natively. It also converts liveness properties into safety variants in the standard way. The limitation is that it does not allow the user to provide partial implementations.

The bundle ltl2smv[7] - smvflatten - smvtoaig [3] can translate SMV files with LTL properties embedded into AIGER format. The idea is:

1. smvflatten accepts a given SMV file with modules and variable types like range and enums, and translates it into boolean SMV file, preserving the original LTLSPEC section.
2. The result is sent to smvtoaig that translates LTLSPEC section into SMV module using ltl2smv, then joins the result, and translates it into AIGER circuit.

I.e, it does what we want but in the context of the model checking. For synthesis we also need:

- to provide the signals partition (into controllable and uncontrollable) – a minor issue, and
- to ensure there are no non-deterministic automata and thus no non-deterministic SMV modules produced at step (2) by ltl2smv. [5] One way to achieve this is to provide a custom implementation of ltl2smv. In hindsight, I think this might be a good way to go.

Finally, in the work in progress paper [12] the authors target a similar goal of providing a rich specification language that benefits from efficient synthesizers. In that work the authors automatically translate often used LTL patterns into the GR(1) fragment of LTL that has an efficient synthesis algorithm [11]. They do not allow for providing partial implementations.

[4] Recall that we synthesize a memory-less strategy, thus introduce only new AND-gates and no additional memory.

[5] This is because we cannot resolve non-determinism by adding the uncontrollable input: the synthesizer is aware of all circuit's signals, thus it may wait for the input to raise and then behave accordingly. I.e., we need to ensure that a system strategy is independent of the auxiliary signal – the partial information, which is not supported by the SYNTCOMP format.

6 Conclusions

In this paper we proposed a format to ease the specification task that allows the user to provide partial implementations, and we built the conversion scripts from the new format into the SYNTCOMP format. Both the specification format and the way we convert into the existing format are subject to discussion:

- Is there a more convenient format of specifications? Is SMV enough or Verilog should be used instead? Should we support GR(1)? Partial information?
- Is there a simpler way to convert from the new format into the SYNTCOMP format?

Acknowledgements. This paper would not be possible without numerous fruitful discussions with Robert Könighofer, Roderick Bloem, and Georg Hofferek. Many thanks to reviewers for valuable suggestions.

References

[1] Armin Biere: *AIGER format and toolbox*. Available at `http://fmv.jku.at/aiger/`.

[2] Armin Biere: *AIGER format version 20070427*. Available at `http://fmv.jku.at/aiger/FORMAT-20070427.pdf`.

[3] Armin Biere: smvflatten. Available at `http://fmv.jku.at/smvflatten/`.

[4] Armin Biere, Keijo Heljanko & Siert Wieringa (2011): *AIGER 1.9 and Beyond*. Available at `http://www.fmv.jku.at/hwmcc11/beyond1.pdf`.

[5] Aaron Bradley, Arlen Cox, Michael Dooley, Zyad Hassan, Fabio Somenzi & Yan Zhang: *IIMC*. Available at `http://ecee.colorado.edu/wpmu/iimc/`.

[6] RobertK. Brayton, GaryD. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, RajeevK. Ranjan, Shaker Sarwary, ThomasR. Staple, Gitanjali Swamy & Tiziano Villa (1996): *VIS: A system for verification and synthesis*. In: *CAV*, *LNCS* 1102, pp. 428–432, doi:10.1007/3-540-61474-5_95.

[7] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani & Armando Tacchella (2002): *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. In: *CAV*, *LNCS* 2404, pp. 359–364, doi:10.1007/3-540-45657-0_29.

[8] D.A. Huffman (1952): *A Method for the Construction of Minimum-Redundancy Codes*. *Proceedings of the IRE* 40(9), pp. 1098–1101, doi:10.1109/JRPROC.1952.273898.

[9] Swen Jacobs (2014): *Extended AIGER Format for Synthesis (v0.1)*. ArXiv:1405.5793.

[10] Guillermo A. Perez: ltl2aig. Available at `https://github.com/gaperez64/acacia_ltl2aig`.

[11] Nir Piterman, Amir Pnueli & Yaniv Saar (2006): *Synthesis of reactive (1) designs*. In: *Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 364–380, doi:10.1007/11609773_24.

[12] Jan Oliver Ringert (2015): *Extensible Support for Specification Patterns in GR(1) Synthesis (Work in Progress)*. Young Researchers' Conference "Frontiers of Formal Methods". Available at `http://ffm2015.rwth-aachen.de/proceedings.php`.

[13] Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Kang-Nien Wu & Wen-Chin Chan (2007): *GOAL: A graphical tool for manipulating Büchi automata and temporal formulae*. In: *TACAS*, Springer, pp. 466–471, doi:10.1007/978-3-540-71209-1_35.

[14] M. Y. Vardi (2011): *The rise and fall of linear time logic*. 2nd Intl Symp. on Games, Automata, Logics and Formal Verification. Invited talk.