A High-Level LTL Synthesis Format: TLSF v1.1 (Extended Version)

Swen Jacobs
Saarland University
Saarbrücken, Germany
jacobs@react.uni-saarland.de

Felix Klein
Saarland University
Saarbrücken, Germany
klein@react.uni-saarland.de

Sebastian Schirmer
Saarland University
Saarbrücken, Germany
s9sescir@stud.uni-saarland.de

We present the Temporal Logic Synthesis Format (TLSF), a high-level format to describe synthesis problems via Linear Temporal Logic (LTL). The format builds upon standard LTL, but additionally allows to use high-level constructs, such as sets and functions, to provide a compact and human-readable representation. Furthermore, the format allows to identify parameters of a specification such that a single description can be used to define a family of problems. Additionally, we present a tool to automatically translate the format into plain LTL, which then can be used for synthesis by a solver. The tool also allows to adjust parameters of the specification and to apply standard transformations on the resulting formula.

1 Introduction

The automatic synthesis of reactive systems from formal specifications has been one of the major challenges of computer science, and an active field of research, since the definition of the problem by Church [6]. For specifications in linear temporal logic the problem is 2EXPTIME-complete, and a number of fundamental approaches to solve this problem have been proposed [5, 24, 23], based on a translation of the specification into a game or an automaton. Recently, there has been a lot of work on solving synthesis problems more efficiently, either by restricting the specification language [4, 22], or by a smart exploration of the search space [15, 9, 12, 26, 14, 13].

However, as already noted by Ehlers [8], it has been very hard to compare different synthesis tools. A major reason for this was the lack of a common language and a benchmark library on which to compare tools. As a consequence, there has also been a lack of incentive for the development of efficient implementations of new synthesis approaches.

To some extent, this has changed with the advent of the reactive synthesis competition (SYNTCOMP) [17, 18], which has been organized in order to encourage the development of mature and efficient synthesis tools. However, SYNTCOMP thus far was restricted to safety specifications in an extension of the AIGER format [16], a low-level format that is not suited for writing expressive specifications by hand. Moreover, AIGER files directly represent a (safety) game, and the translation of a temporal logic specification to a suitable game (or other intermediate representations) is a non-trivial part of the synthesis problem that is removed from the picture if we start from an AIGER specification.

In this paper, we introduce the *temporal logic synthesis format* (TLSF), a high-level format for the specification of synthesis problems. The goal of TLSF is to create a format that (i) makes it

To appear in EPTCS.

convenient to write expressive specifications by hand, and at the same time (ii) is *easy to support by synthesis tools*.

To achieve the first goal, TLSF allows to define synthesis problems with high-level temporal logic specifications, and supports a number of additional features. These include user-defined enumeration types and signal buses, function declarations (including recursion), and the definition of parameters that allow to easily define parameterized families of synthesis problems.

To achieve the second goal, we define a *basic format* that is essentially restricted to linear temporal logic (LTL) without these additional features, and we supply the *Synthesis Format Conversion Tool* (SyFCo) that can compile arbitrary TLSF specifications into the basic format. Hence, for synthesis tools it is sufficient to support the much simpler basic format. Moreover, the SyFCo tool also supports a number of additional features, including the translation to some existing specification formats like Promela LTL [1] or PSL [11], and is easily extensible to other formats.

To demonstrate the features of our specification format, we provide a version of the AMBA arbiter specification in TLSF. In addition to this, a large number of existing benchmarks have already been converted to TLSF, and can be found in our publicly available repository [2].

TLSF will be used as a high-level format in several new tracks of SYNTCOMP in 2016. The goal is to develop and maintain a standard format for synthesis from high-level temporal logic specifications, and to use our repository of benchmarks as a starting point for a growing benchmark library that will be part of SYNTCOMP. The design decisions that went into TLSF are inspired by the findings of Schirmer [25], who compared existing synthesis formats and made a first proposal towards the goals stated above.

Overview. We present the basic version of the Temporal Logic Synthesis Format (TLSF) in Sect. 2. In Sect. 3 we discuss the intended semantics of a specification, defined in terms of different implementation models. The full format is introduced in Sect. 4, followed by an illustration of its main features on an example in Sect. 5. In Sect. 6, we give an overview of the SyFCo Tool. Finally, we discuss possible extensions of the format in Sect. 7.

2 The Basic Format

A specification in the basic format consists of an `INFO` section and a `MAIN` section:

$$\langle info \rangle \langle main \rangle$$

2.1 The INFO Section

The `INFO` section contains the meta data of the specification, like a title and some description[1]. Furthermore, it defines the underlying semantics of the specification (Mealy or Moore / standard or strict implication) and the target model of the synthesized implementation. Detailed information about supported semantics and targets can be found in Sect. 3. Finally, a comma separated list of tags can be specified to identify features of the specification, e.g., the restriction to a specific fragment of LTL. A $\langle tag \rangle$ can be any string literal and is not restricted to any predefined keywords.

[1] We use colored verbatim font to identify the syntactic elements of the specification.

```
INFO {
  TITLE:       "⟨some title⟩"
  DESCRIPTION: "⟨some description⟩"
  SEMANTICS:   ⟨semantics⟩
  TARGET:      ⟨target⟩
  TAGS:        ⟨tag⟩, ⟨tag⟩, ...
}
```

2.2 The MAIN Section

The specification is completed by the MAIN section, which contains the partitioning of input and output signals, followed by the main specification. The specification itself is separated into assumptions on the environment and desired properties of the system, and can additionally be distinguished into initial (INITIALLY/PRESET), invariant (REQUIRE/ASSERT), and arbitrary (ASSUME/GUARANTEE) properties[2]. Multiple declarations and expressions need to be separated by a ';'.

```
MAIN {
  INPUTS     { (⟨boolean signal declaration⟩;)* }
  OUTPUTS    { (⟨boolean signal declaration⟩;)* }
  INITIALLY  { (⟨basic LTL expression⟩;)* }
  PRESET     { (⟨basic LTL expression⟩;)* }
  REQUIRE    { (⟨basic LTL expression⟩;)* }
  ASSERT     { (⟨basic LTL expression⟩;)* }
  ASSUME     { (⟨basic LTL expression⟩;)* }
  GUARANTEE  { (⟨basic LTL expression⟩;)* }
}
```

All subsections except INPUTS and OUTPUTS are optional.

2.3 Basic Expressions

A basic expression e is either a boolean signal or a basic LTL expression. Each basic expression has a corresponding type that is $\mathbb{S}$ for boolean signals and $\mathbb{T}$ for LTL expressions. Basic expressions can be composed to larger expressions using operators. An overview over the different types of expressions and operators is given below.

Boolean Signal Declarations. A signal identifier is represented by a string consisting of lowercase and uppercase letters ('a'-'z', 'A'-'Z'), numbers ('0'-'9'), underscores ('_'), primes (''), and at-signs ('@') and does not start with a number or a prime. Additionally, keywords like X, G or U, as defined in the rest of this document, are forbidden. An identifier is declared as either an input or an output signal. We denote the set of declared input signals as $\mathcal{I}$ and the set of declared output signals as $\mathcal{O}$, where $\mathcal{I} \cap \mathcal{O} = \emptyset$. Then, a boolean signal declaration simply consists of a signal identifier $\langle name \rangle$ from $\mathcal{I} \cup \mathcal{O}$.

Basic LTL Expressions. A basic LTL expression conforms to the following grammar, including truth values, signals, boolean operators and temporal operators. For easy parsing of the basic format, we

[2] In TLSF v1.0 [19], ASSERT was called INVARIANTS, ASSUME was called ASSUMPTIONS, and GUARANTEE was called GUARANTEES (and subsections INITIALLY, PRESET, and REQUIRE did not exist). TLSF v1.1 still supports the old identifiers.

require fully parenthesized expressions, as expressed by the first of the following lines:

$$\begin{aligned}
\varphi &\equiv \texttt{(}\varphi'\texttt{)} \\
\varphi' &\equiv \texttt{true} \mid \texttt{false} \mid s \quad \text{for } s \in \mathcal{I} \cup \mathcal{O} \mid \\
&\quad \texttt{!}\varphi \mid \varphi\ \texttt{\&\&}\ \varphi \mid \varphi\ \texttt{||}\ \varphi \mid \varphi\ \texttt{->}\ \varphi \mid \varphi\ \texttt{<->}\ \varphi \\
&\quad \texttt{X}\ \varphi \mid \texttt{G}\ \varphi \mid \texttt{F}\ \varphi \mid \varphi\ \texttt{U}\ \varphi \mid \varphi\ \texttt{R}\ \varphi \mid \varphi\ \texttt{W}\ \varphi
\end{aligned}$$

Thus, a basic LTL expression is either true, false, or a signal, or composed from these atomic expressions with boolean operators (negation, conjunction, disjunction, implication, equivalence) and temporal operators (next, globally, eventually, until, release, weak until). The semantics of the boolean operators are defined in the usual way, and the temporal operators are defined in Appendix A.1.

3 Targets and Semantics

3.1 Targets

The `TARGET` of the specification defines the implementation model that a solution should adhere to. Currently supported targets are Mealy automata (`Mealy`), whose output depends on the current state and input, and Moore automata (`Moore`), whose output only depends on the current state. The differentiation is necessary since realizability of a specification depends on the target system model. For example, every specification that is realizable under Moore semantics is also realizable under Mealy semantics, but not vice versa. A formal description of both automata models can be found in Appendix A.2.

3.2 Semantics

The `SEMANTICS` of the specification defines how the formula was intended to be evaluated, which also depends on an implementation model. We currently support four different semantics: standard Mealy semantics (`Mealy`), standard Moore semantics (`Moore`), strict Mealy semantics (`Mealy, Strict`), and strict Moore semantics (`Moore,Strict`).

In the following, consider a specification where `INITIALLY` evaluates to the LTL formula θ_e, `PRESET` evaluates to θ_s, `REQUIRE` evaluates to ψ_e, `ASSERT` evaluates to ψ_s, `ASSUME` evaluates to φ_e, and `GUARANTEE` evaluates to φ_s. For specification sections that are not present, the respective formula is interpreted as `true`.

Standard semantics. If the semantics is (non-strict) `Mealy` or `Moore`, and the `TARGET` coincides with the semantics system model, then the specification is interpreted as the formula

$$\theta_e \rightarrow (\theta_s \wedge (\mathsf{G}\,\psi_e \wedge \varphi_e \rightarrow \mathsf{G}\,\psi_s \wedge \varphi_s))$$

in standard LTL semantics (see Appendix A.1). Note that we require that the `PRESET` property θ_s holds whenever the `INITIALLY` condition θ_e holds, regardless of other environment assumptions.

Strict semantics. If the semantics is `Mealy,Strict` or `Moore,Strict`, and the `TARGET` coincides with the semantics system model, then the specification is interpreted under strict implication semantics (as used in the synthesis of GR(1) specifications), which is equivalent to the formula

$$\theta_e \rightarrow (\theta_s \wedge (\psi_s \,\mathcal{W}\, \neg\psi_e) \wedge (\mathsf{G}\,\psi_e \wedge \varphi_e \rightarrow \varphi_s))$$

in standard LTL semantics. In this case, we additionally require that the ASSERT property ψ_s needs to hold at least as long as the REQUIRE condition ψ_e holds.

Note that this gives us an easy way to convert a specification with strict semantics into one with non-strict semantics. For details on strict implication semantics, see Klein and Pnueli [21], as well as Bloem et al. [4], from which we also take our definition and interpretation of the GR(1) fragment.[3]

Conversion between system models. If the implementation model of the SEMANTICS differs from the TARGET of a specification, we use a simple conversion to get a specification that is realizable in the target system model iff the original specification is realizable in the original system model: a specification in Moore semantics can be converted into Mealy semantics by prefixing all occurrences of input atomic propositions with an additional X-operator. Similarly, we can convert from Mealy semantics to Moore semantics by prefixing outputs with a X-operator.

4 The Full Format

In the full format, a specification consists of three sections: the INFO section, the GLOBAL section and the MAIN section. The GLOBAL section is optional.

$$\langle info \rangle [\langle global \rangle] \langle main \rangle$$

The INFO section is the same as in the basic format, defined in Sect. 2.1. The GLOBAL section can be used to define parameters, and to bind identifiers to expressions that can be used later in the specification. The MAIN section is used as before, but can use extended sets of declarations and expressions.

We define the GLOBAL section in Sect. 4.1, and the changes to the MAIN section compared to the basic format in Sect. 4.2. The extended set of expressions that can be used in the full format is introduced in Sect. 4.3, enumerations, extended signal and function declarations in Sect. 4.4 and 4.6, and additional notation in Sect. 4.7–4.9.

4.1 The GLOBAL Section

The GLOBAL section consists of the PARAMETERS subsection, defining the identifiers that parameterize the specification, and the DEFINITIONS subsection, that allows to define functions, enumerations and to bind identifiers to complex expressions. Multiple declarations need to be separated by a ';'. The section and its subsections are optional.

```
GLOBAL {
  PARAMETERS {
    (⟨identifier⟩ = ⟨numerical expression⟩;)*
  }
  DEFINITIONS  {
    ((⟨function declaration⟩ | ⟨enum declaration⟩ | ⟨identifier⟩ = ⟨expression⟩);)*
  }
}
```

[3] Note that in the conversion of [4], the formula is strengthened by adding the formula $\mathsf{G}(\mathsf{H}\,\psi_e \rightarrow \psi_s)$, where $\mathsf{H}\,\varphi$ is a Past-LTL formula and denotes that φ holds everywhere in the past. However, it is easy to show that our definition of strict semantics matches the definition of [4]. We prefer this notion, since it avoids the introduction of Past-LTL.

4.2 The MAIN Section

Like in the basic format, the `MAIN` section contains the partitioning of input and output signals, as well as the main specification. However, signal declarations can now contain signal buses, and LTL expressions can use parameters, functions, and identifiers defined in the `GLOBAL` section.

```
MAIN {
  INPUTS     { (⟨signal declaration⟩;)* }
  OUTPUTS    { (⟨signal declaration⟩;)* }
  INITIALLY { (⟨LTL expression⟩;)* }
  PRESET     { (⟨LTL expression⟩;)* }
  REQUIRE    { (⟨LTL expression⟩;)* }
  ASSERT     { (⟨LTL expression⟩;)* }
  ASSUME     { (⟨LTL expression⟩;)* }
  GUARANTEE { (⟨LTL expression⟩;)* }
}
```

As before, all subsections except `INPUTS` and `OUTPUTS` are optional.

4.3 Expressions

An expression e is either a boolean signal, an n-ary signal (called bus), an enumeration type, a numerical expression, a boolean expression, an LTL expression, or a set expression. Each expression has a corresponding type that is either one of the basic types: $\mathbb{S}, \mathbb{U}, \mathbb{E}, \mathbb{N}, \mathbb{B}, \mathbb{T}$, or a recursively defined set type $\mathcal{S}_{\mathbb{X}}$ for some type $\mathbb{X}$.

As before, an identifier is represented by a string consisting of lowercase and uppercase letters ('`a`'-'`z`', '`A`'-'`Z`'), numbers ('`0`'-'`9`'), underscores ('`_`'), primes ('`'`'), and at-signs ('`@`') and does not start with a number or a prime. In the full format, identifiers are bound to expressions of different type. We denote the respective sets of identifiers by $\Gamma_{\mathbb{S}}$, $\Gamma_{\mathbb{U}}$, $\Gamma_{\mathbb{E}}$, $\Gamma_{\mathbb{N}}$, $\Gamma_{\mathbb{B}}$, $\Gamma_{\mathbb{T}}$, and $\Gamma_{\mathcal{S}_{\mathbb{X}}}$. Finally, basic expressions can be composed to larger expressions using operators. In the full format, we do not require fully parenthesized expressions. If an expression is not fully parenthesized, we use the precedence order given in Table 1. An overview over the all types of expressions and operators is given below.

Numerical Expressions. A numerical expression $e_{\mathbb{N}}$ conforms to the following grammar:

$$\begin{aligned} e_{\mathbb{N}} \quad \equiv \quad & i \quad \text{for } i \in \Gamma_{\mathbb{N}} \mid n \quad \text{for } n \in \mathbb{N} \mid e_{\mathbb{N}} \texttt{ + } e_{\mathbb{N}} \mid e_{\mathbb{N}} \texttt{ - } e_{\mathbb{N}} \mid e_{\mathbb{N}} \texttt{ * } e_{\mathbb{N}} \mid e_{\mathbb{N}} \texttt{ / } e_{\mathbb{N}} \mid e_{\mathbb{N}} \texttt{ \% } e_{\mathbb{N}} \\ & \texttt{|}e_{\mathcal{S}_{\mathbb{X}}}\texttt{|} \mid \texttt{MIN } e_{\mathcal{S}_{\mathbb{N}}} \mid \texttt{MAX } e_{\mathcal{S}_{\mathbb{N}}} \mid \texttt{SIZEOF } s \quad \text{for } s \in \Gamma_{\mathbb{U}} \end{aligned}$$

Thus, a numerical expression either represents an identifier (bound to a numerical value), a numerical constant, an addition, a subtraction, a multiplication, an integer division, a modulo operation, the size of a set, the minimal/maximal value of a set of naturals, or the size (i.e., width) of a bus, respectively. The semantics are defined in the usual way.

Set Expressions. A set expression $e_{\mathcal{S}_{\mathbb{X}}}$, containing elements of type $\mathbb{X}$, conforms to the following grammar:

$$\begin{aligned} e_{\mathcal{S}_{\mathbb{X}}} \quad \equiv \quad & i \quad \text{for } i \in \Gamma_{\mathcal{S}_{\mathbb{X}}} \mid \texttt{\{}e_{\mathbb{X}}\texttt{,}\, e_{\mathbb{X}}\texttt{,}\, \ldots\texttt{,}\, e_{\mathbb{X}}\texttt{\}} \mid \texttt{\{}e_{\mathbb{N}}\texttt{,}\, e_{\mathbb{N}}\texttt{..}\, e_{\mathbb{N}}\texttt{\}} \mid \\ & e_{\mathcal{S}_{\mathbb{X}}}\texttt{ (+) }e_{\mathcal{S}_{\mathbb{X}}} \mid e_{\mathcal{S}_{\mathbb{X}}}\texttt{ (*) }e_{\mathcal{S}_{\mathbb{X}}} \mid e_{\mathcal{S}_{\mathbb{X}}}\texttt{ (\textbackslash) }e_{\mathcal{S}_{\mathbb{X}}} \end{aligned}$$

Precedence	Operator	Description	Arity	Associativity

1	`+[·]` (`SUM[·]`)
	`*[·]` (`PROD[·]`)
	`|···|` (`SIZE`)
	`MIN`
	`MAX`
	`SIZEOF`
		sum
		product
		size
		minimum
		maximum
		size of a bus
			unary	

2	`*` (`MUL`)	multiplication	binary	left-to-right
3	`/` (`DIV`) `%` (`MOD`)	integer division modulo	binary	right-to-left
4	`+` (`PLUS`) `-` (`MINUS`)	addition difference	binary	left-to-right
5	`(*)[·]` (`CAP[·]`) `(+)[·]` (`CUP[·]`)	intersection union	unary	
6	`(\)` (`(-)`,`SETMINUS`)	set difference	binary	right-to-left
7	`(*)` (`CAP`)	intersection	binary	left-to-right
8	`(+)` (`CUP`)	union	binary	left-to-right
9	`==` (`EQ`) `!=` (`/=`, `NEQ`) `<` (`LE`) `<=` (`LEQ`) `>` (`GE`) `>=` (`GEG`)	equality inequality smaller than smaller or equal than greater then greater or equal than	binary	left-to-right
10	`IN` (`ELEM`, `<-`)	membership	binary	left-to-right

11	`!` (`NOT`)
	`X`
	`F`
	`G`
	`&&[·]` (`AND[·]`, `FORALL[·]`)
	`||[·]` (`OR[·]`, `EXISTS[·]`)
		negation
		next
		finally
		globally
		conjunction
		disjunction
			unary	

12	`&&` (`AND`)	conjunction	binary	left-to-right
13	`||` (`OR`)	disjunction	binary	left-to-right
14	`->` (`IMPLIES`) `<->` (`EQUIV`)	implication equivalence	binary	right-to-left
15	`W`	weak until	binary	right-to-left
16	`U`	until	binary	right-to-left
17	`R`	release	binary	left-to-right
18	`~`	pattern match	binary	left-to-right
19	`:`	guard	binary	left-to-right

Table 1: The table lists the precedence, arity and associativity of all expression operators. Also consider the alternative names in brackets which can be used instead of the symbolic representations.

Thus, the expression $e_{\mathcal{S}_\mathbb{X}}$ either represents an identifier (bound to a set of values of type $\mathbb{X}$), an explicit list of elements of type $\mathbb{X}$, a list of elements specified by a range (for $\mathbb{X} = \mathbb{N}$), a union of two sets, an intersection or a difference, respectively. The semantics of a range expression `{`x`,`y`..`z`}` are defined for $x < y$ via:

$$\{n \in \mathbb{N} \mid x \leq n \leq z \wedge \exists j.\, n = x + j \cdot (y - x)\}.$$

The semantics of all other expressions are defined as usual. Sets contain either positive integers, boolean expressions, LTL expressions, buses, signals, or other sets of a specific type.

Boolean Expressions. A boolean expression $e_\mathbb{B}$ conforms to the following grammar:

$$\begin{aligned} e_\mathbb{B} \quad \equiv \quad & i \ \text{ for } i \in \Gamma_\mathbb{B} \mid e_\mathbb{X}\ \texttt{IN}\ e_{\mathcal{S}_\mathbb{X}} \mid \texttt{true} \mid \texttt{false} \mid \texttt{!}\, e_\mathbb{B} \mid \\ & e_\mathbb{B}\ \texttt{\&\&}\ e_\mathbb{B} \mid e_\mathbb{B}\ \texttt{||}\ e_\mathbb{B} \mid e_\mathbb{B}\ \texttt{->}\ e_\mathbb{B} \mid e_\mathbb{B}\ \texttt{<->}\ e_\mathbb{B} \mid \\ & e_\mathbb{N}\ \texttt{==}\ e_\mathbb{N} \mid e_\mathbb{N}\ \texttt{!=}\ e_\mathbb{N} \mid e_\mathbb{N}\ \texttt{<}\ e_\mathbb{N} \mid e_\mathbb{N}\ \texttt{<=}\ e_\mathbb{N} \mid e_\mathbb{N}\ \texttt{>}\ e_\mathbb{N} \mid e_\mathbb{N}\ \texttt{>=}\ e_\mathbb{N} \end{aligned}$$

Thus, a boolean expression either represents an identifier (bound to a boolean value), a membership test, true, false, a negation, a conjunction, a disjunction, an implication, an equivalence, or an equation between two positive integers (equality, inequality, less than, less or equal than, greater than, greater or equal than), respectively. The semantics are defined in the usual way. Note that signals are not allowed in a boolean expression, but only in an LTL expression.

LTL Expressions. An LTL expression φ conforms to the same grammar as a boolean expression, except that it additionally includes signals and temporal operators.

$$\begin{aligned} \varphi \quad \equiv \quad & \ldots \mid i \ \text{ for } i \in \Gamma_\mathbb{T} \mid s \ \text{ for } s \in \Gamma_\mathbb{S} \mid b\texttt{[}e_\mathbb{N}\texttt{]} \ \text{ for } b \in \Gamma_\mathbb{U} \mid \\ & b_0\ \texttt{==}\ b_1 \ \text{ for } b_j \in \Gamma_\mathbb{U} \text{ and } b_{1-j} \in \Gamma_\mathbb{E} \mid b_0\ \texttt{!=}\ b_1 \ \text{ for } b_j \in \Gamma_\mathbb{U} \text{ and } b_{1-j} \in \Gamma_\mathbb{E} \mid \\ & \texttt{X}\ \varphi \mid \texttt{G}\ \varphi \mid \texttt{F}\ \varphi \mid \varphi\ \texttt{U}\ \varphi \mid \varphi\ \texttt{R}\ \varphi \mid \varphi\ \texttt{W}\ \varphi \end{aligned}$$

Thus, an LTL expression additionally can represent an identifier bound to an LTL formula, a signal, an $e_\mathbb{N}$-th signal of a bus, a next operation, a restriction of a bus to a set of enumeration valuations via equality or inequality, a globally operation, an eventually operation, an until operation, a release operation, or a weak until operation, respectively. Note that every boolean expression is also an LTL expression, thus we allow the use of identifiers that are bound to boolean expressions as well. A formal definition of the semantics of the temporal operators is given in Appendix A.1. The semantics of expressions involving bus operations is defined in the subsequent sections.

4.4 Enumerations

An enumeration declaration conforms to the following grammar:

$$\texttt{enum}\ \langle \textit{enumtype} \rangle\ \texttt{=}\ \Big(\langle \textit{identifier} \rangle\ \texttt{:}\ (\texttt{0} \mid \texttt{1} \mid \texttt{*})^n \big(\texttt{,}\ (\texttt{0} \mid \texttt{1} \mid \texttt{*})^n\big)^* \Big)^+$$

for some arbitrary but fix positive integer $n > 0$. As an example consider the enumeration `Positions`, which declares the enumeration identifiers `LEFT`, `MIDDLE`, `RIGHT`, and `UNDEF` as members of $\Gamma_\mathbb{E}$:

```
enum Position =
  LEFT:   100
  MIDDLE: 010
```

```
RIGHT:  001
UNDEF:  11*, 1*1, *11
```

We use `0` to identify the absent signal, `1` to identify the present signal and `*` for either of both. Each identifier then refers to at least one concrete signal valuation sequence. Multiple values can be denoted by sequences with a `*`, as well as by comma separated lists. Furthermore, the identifier of each declared valuation has to be unique. Not all possible valuations have to be identified.

Enumeration identifiers can only be used in comparisons against buses inside an LTL expression, where we require that the corresponding bus has the same width as the valuation compared to. It defines a boolean constraint on the bus, restricting it to the different valuations, bound to the identifier, e.g., the expressions `b == RIGHT` and `!b[0] && !b[1] && b[2]` are semantically equivalent, as well as `b /= UNDEF` and `!((b[0] && b[1]) || (b[0] && b[2]) || (b[1] && b[2]))`.

4.5 Signals and Buses

A single signal declaration consists of the name of the signal. As for the basic format, signals are declared as either input or output signals, denoted by $\mathcal{I}$ and $\mathcal{O}$, respectively. A bus declaration additionally specifies a signal width, i.e., a bus represents a finite set of signals. The signal width is either given by a numerical value or via an enumeration type.

$$\langle name \rangle \;\mid\; \langle name \rangle \texttt{[}e_{\mathbb{N}}\texttt{]} \;\mid\; \langle enumtype \rangle \langle name \rangle$$

Semantically, a signal declaration `s` specifies a signal $s \in \mathcal{I} \cup \mathcal{O}$, where a bus declaration `b[n]` specifies n signals `b[0]`, `b[1]`, …, `b[n-1]`, with either `b[i]` $\in \mathcal{I}$ for all $0 \leq i < n$, or `b[i]` $\in \mathcal{O}$ for all $0 \leq i < n$. A bus specified via an enumeration type has the same width as the valuations of the corresponding enumeration.

Buses which are declared using an enumeration type, where not all valuations are related to an identifier[4] induce an implicit constraint on the corresponding signals: if the bus corresponds to a set of input signals, then the global requirement that no other than the defined valuations appear on this bus is imposed. If it corresponds to a set of output signals, then the equivalent global invariant is imposed.

Finally, note that we use `b[i]` to access the i-th value of b, i.e., we use the same syntax as for the declaration itself[5]. Also note that for the declared signals s, we have $s \in \mathcal{I} \cup \mathcal{O} \subseteq \Gamma_{\mathbb{S}}$, and for the declared buses b, we have $b \in \Gamma_{\mathbb{U}}$.

4.6 Function Declarations

As another feature, one can declare (recursive) functions of arbitrary arity inside the `DEFINITIONS` section. Functions can be used to define simple macros, but also to generate complex formulas from a given set of parameters. A declaration of a function of arity n has the form

$$\langle \textit{function name} \rangle \texttt{(}\langle arg_1 \rangle \texttt{,} \langle arg_2 \rangle \texttt{,} \ldots \texttt{,} \langle arg_n \rangle\texttt{) = } (e_c)^+,$$

where $\langle arg_1 \rangle, \langle arg_2 \rangle, \ldots, \langle arg_n \rangle$ are fresh identifiers that can only be used inside the sub-expressions e_c. An expression e_c conforms to the following grammar:

$$e_c \;\equiv\; e \mid e_{\mathbb{B}} \texttt{ : } e \mid e_{\mathbb{P}} \texttt{ : } e \qquad \text{where} \quad e \;\equiv\; e_{\mathbb{N}} \mid e_{\mathbb{B}} \mid e_{\mathcal{S}_{\mathbb{X}}} \mid \varphi$$

[4] See e.g. the `000` valuation of the example of Sect. 4.5

[5] C-Array Syntax Style

Thus, a function can be bound to any expression e, parameterized in its arguments, which additionally may be guarded by some boolean expression $e_{\mathbb{B}}$, or a pattern match $e_{\mathbb{P}}$. If the regular expression $(e_c)^+$ consists of more than one expression e_c, then the function binds to the first expression whose guard evaluates to `true` (in the order of their declaration). Furthermore, the special guard `otherwise` can be used, which evaluates to `true` if and only if all other guards evaluate to `false`. Expressions without a guard are implicitly guarded by `true`. All sub-expressions e_c need to have the same type $\mathbb{X}$. For every instantiation of a function by given parameters, we view the resulting expression $e_{\mathbb{X}}$ as an identifier in $\Gamma_{\mathbb{X}}$, bound to the result of the function application.

Pattern Matching. Pattern matches are special guards of the form

$$e_{\mathbb{P}} \equiv \ \varphi \sim \varphi',$$

which can be used to describe different behavior depending on the structure of an LTL expression. Hence, a guard $e_{\mathbb{P}}$ evaluates to `true` if and only if φ and φ' are structurally equivalent, with respect to their boolean and temporal connectives. Furthermore, identifier names that are used in φ' need to be fresh, since every identifier expression that appears in φ' is bound to the equivalent sub-expression in φ, which is only visible inside the right-hand-side of the guard. Furthermore, to improve readability, the special identifier `_` (wildcard) can be used, which always remains unbound. To clarify this feature, consider the following function declaration:

```
fun(f) =
  f ~ a U _: a
  otherwise: X f
```

The function *fun* gets an LTL formula f as a parameter. If f is an until formula of the form $\varphi_1 \,\mathcal{U}\, \varphi_2$, then $\mathit{fun}(f)$ binds to φ_1, otherwise $\mathit{fun}(f)$ binds to $\mathsf{X} f$.

4.7 Big Operator Notation

It is often useful to express parameterized expressions using "big" operators, e.g., we use Σ to denote a sum over multiple sub-expressions, Π to denote a product, or $\bigcup$ to denote a union. It is also possible to use this kind of notion in this specification format. The corresponding syntax looks as follows:

$$\langle op \rangle \texttt{[} \langle id_0 \rangle \texttt{ IN } e_{\mathcal{S}_{\mathbb{X}_0}} \texttt{,} \langle id_1 \rangle \texttt{ IN } e_{\mathcal{S}_{\mathbb{X}_1}} \texttt{, } \ldots \texttt{, } \langle id_n \rangle \texttt{ IN } e_{\mathcal{S}_{\mathbb{X}_n}} \texttt{]}\ e_{\mathbb{X}}$$

Let x_j be the identifier represented by $\langle id_j \rangle$ and S_j be the set represented by $e_{\mathcal{S}_{\mathbb{X}_j}}$. Further, let $\bigoplus$ be the mathematical operator corresponding to $\langle op \rangle$. Then, the above expression corresponds to the mathematical expression:

$$\bigoplus_{x_0 \in S_0} \bigoplus_{x_1 \in S_1} \cdots \bigoplus_{x_n \in S_n} (e_{\mathbb{X}})$$

Note that $\langle id_0 \rangle$ is already bound in expression $e_{\mathcal{S}_{\mathbb{X}_1}}$, $\langle id_1 \rangle$ is bound in $e_{\mathcal{S}_{\mathbb{X}_2}}$, and so forth. The syntax is supported by every operator $\langle op \rangle \in \{\texttt{+}, \texttt{*}, \texttt{(+)}, \texttt{(*)}, \texttt{\&\&}, \texttt{||}\}$.

4.8 Syntactic Sugar

To improve readability, there is additional syntactic sugar, which can be used beside the standard syntax. Let n and m be numerical expressions, then

- `X[`n`]` φ denotes a stack of n next operations, e.g.:

  `X[3] a` $\equiv$ `X X X a`

- `F[`n`:`m`]` φ denotes that φ holds somewhere between the next n and m steps, e.g.:

  `F[2:3] a` $\equiv$ `X X(a || X a)`

- `G[`n`:`m`]` φ denotes that φ holds everywhere between the next n and m steps, e.g.:

  `G[1:3] a` $\equiv$ `X(a && X(a && X a))`

- $\langle op \rangle$`[...,`$n \circ_1 \langle id \rangle \circ_2 m$`,...]` e_X denotes a big operator application, where $n \circ_1 \langle id \rangle \circ_2 m$ with $\circ_1, \circ_2 \in \{$`<`, `<=`$\}$ denotes that $\langle id \rangle$ ranges from n to m. The inclusion of n and m depends on the choice of $\circ_1$ and $\circ_2$, respectively. Thus, the notation provides an alternative to membership in combination with set ranges, e.g.:

  `&&[0 <= i < n] b[i]` $\equiv$ `&&[i IN {0,1..n-1}] b[i]`

4.9 Comments

It is possible to use C style comments anywhere in the specification, i.e., there are single line comments initialized by `//` and multi line comments between `/*` and `*/`. Multi line comments can be nested.

5 Example: A Decomposed AMBA Arbiter

To get a feeling for the interplay of the aforementioned features, we present a specification of an arbiter for ARM's *Advanced Microcontroller Bus Architecture* (AMBA) [3] in TLSF, decomposed into multiple components as depicted in Figure 1. Inputs of the system are requests (HBUSREQ) from masters that want to access the bus, and a ready signal (HREADY) from the clients that the masters want to talk to. Additionally, each master has a signal for locking the bus (HLOCK), and different types of locked accesses can be requested (via HBURST). The main output of the system is the number of the master that currently owns the bus (HMASTER, in a binary encoding), and a signal for whether the bus is locked (HMASTLOCK). Additionally, there are outputs for the next master that will get access to the bus (HGRANT, unary encoding).

Our encoding is inspired by existing encodings [20], but also includes some new design aspects with respect to the decomposition. We only consider the TLSF encoding of the components DECODE, ENCODE and ARBITER in detail. Encodings of the remaining components can be found in App. A.3.

First, consider the DECODE component, whose encoding is depicted in Figure 3. The component reads the different values of the HBURST bus and splits them up into separate, mutually exclusive signals. Clearly, enumerations are perfectly suited to describe such a behavior.

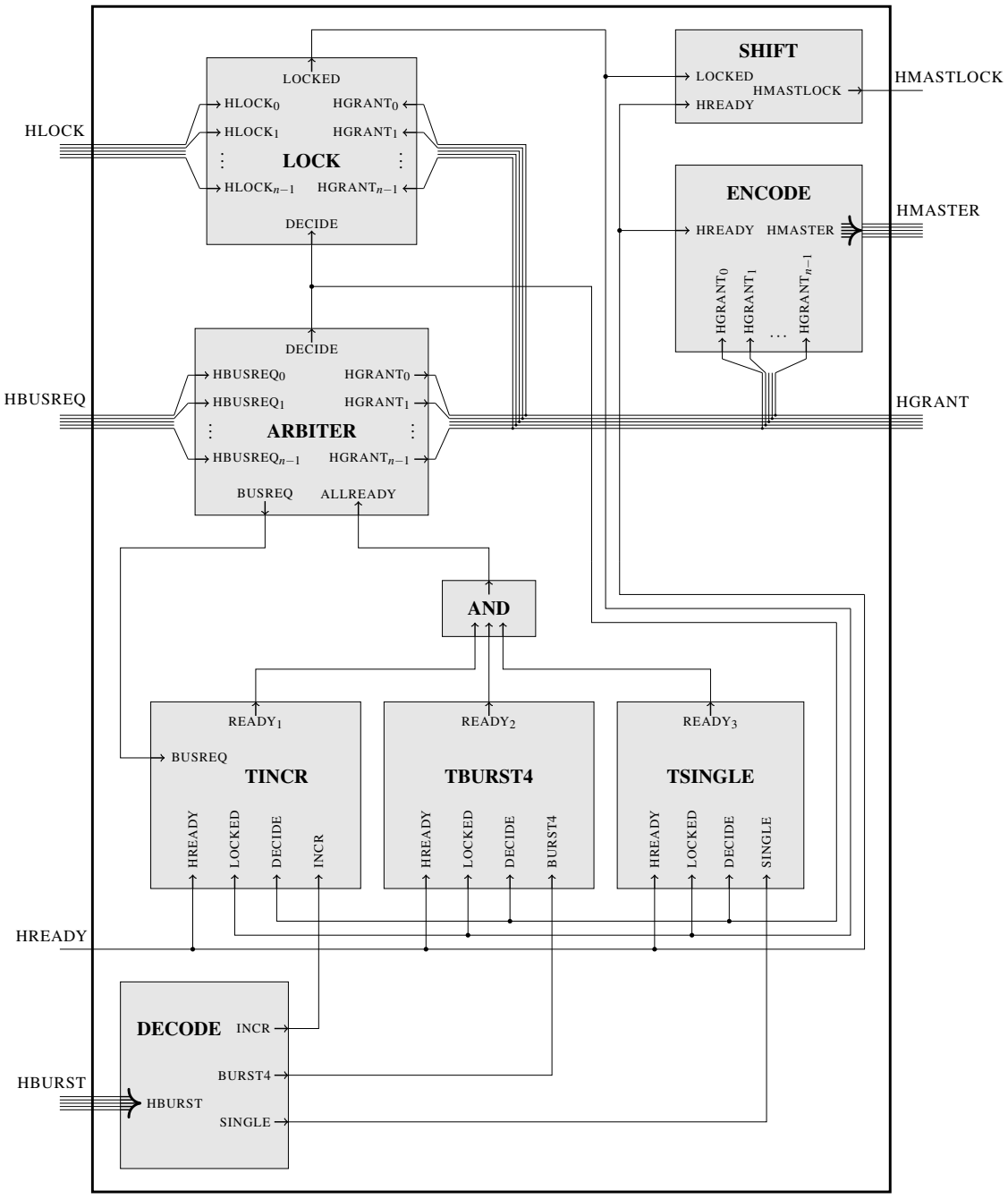

Figure 1: Decomposition of the AMBA AHB arbiter.

Next, consider the ARBITER component, granting the bus to the different masters. The component selects a new master, whenever all other components completed their tasks (signaled by ALLREADY). Furthermore, every master requesting the bus is eventually granted access to it, where every request needs the be held until the access is granted. Additionally, we require that every assignment of a new master triggers the DECIDE flag, which has to be raised one time step in advance to inform other components early about the change. Finally, the request signal of the granted bus is mirrored by

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: Arbiter"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
GLOBAL {
  PARAMETERS {
    n = 2;
  }
  DEFINITIONS {
    // mutual exclusion
    mutual(b) =
      ||[i IN {0, 1 .. (SIZEOF b) - 1}]
        &&[j IN {0, 1 .. (SIZEOF b) - 1} (\) {i}]
          !(b[i] && b[j]);
  }
}
MAIN {
  INPUTS {
    HBUSREQ[n];
    ALLREADY;
  }
  OUTPUTS {
    HGRANT[n];
    BUSREQ;
    DECIDE;
  }
  INITIALLY {
    // the component is initially idle
    ALLREADY;
  }
  ASSUME {
    // the component is not eventually disabled
    G F ALLREADY;
  }
  ASSERT {
    // always exactely one master is granted
    mutual(HGRANT) && ||[0 <= i < n] HGRANT[i];
    // if not ready, the grants stay unchanged
    &&[0 <= i < n]
      (!ALLREADY -> (X HGRANT[i] <-> HGRANT[i]));
    // every request is eventually granted
    &&[0 <= i < n]
      (HBUSREQ[i] -> F (!HBUSREQ[i] || HGRANT[i]));
    // the BUSREQ signal mirrors the HBUSREQ[i]
    // signal of the currently granted master i
    &&[0 <= i < n]
      (HGRANT[i] -> (BUSREQ <-> HBUSREQ[i]));
    // taking decisions requires to be idle
    !ALLREADY -> !DECIDE;
    // granting another master triggers a decision
    DECIDE <-> ||[0 <= i < n]
      !(X HGRANT[i] <-> HGRANT[i]);
    // if there is no request, master 0 is granted
    (&&[0 <= i < n] !HBUSREQ[i]) && DECIDE
       -> X HGRANT[0];
  }
}
```

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: Encode"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
GLOBAL {
  PARAMETERS {
    n = 2;
  }
  DEFINITIONS {
    // mutual exclusion
    mutual(b) =
      ||[i IN {0, 1 .. (SIZEOF b) - 1}]
        &&[j IN {0, 1 .. (SIZEOF b) - 1} (\) {i}]
          !(b[i] && b[j]);
    // checks whether a bus encodes the numerical
    // value v in binary
    value(bus,v) = value'(bus,v,0, SIZEOF  bus);
    value'(bus,v,i,j) =
      i >= j         : true
      bit(v,i) == 1 : value'(bus,v,i+1,j)
                      && bus[i]
      otherwise      : value'(bus,v,i+1,j)
                      && !bus[i];
    // returns the i-th bit of the numerical
    // value v
    bit(v,i) =
      i <= 0     : v % 2
      otherwise : bit(v/2,i-1);
    // discrete logarithm
    log2(x) =
      x <= 1     : 1
      otherwise : 1 + log2(x/2);
  }
}
MAIN {
  INPUTS {
    HREADY;
    HGRANT[n];
  }
  OUTPUTS {
    // the output is encoded in binary
    HMASTER[log2(n-1)];
  }
  REQUIRE {
    // a every time exactely one grant is high
    mutual(HGRANT) && ||[0 <= i < n] HGRANT[i];
  }
  ASSERT {
    // output the binary encoding of i, whenever
    // i is granted and HREADY is high
    &&[0 <= i < n] (HREADY ->
      (X value(HMASTER,i) <-> HGRANT[i]));
    // when HREADY is low, the value is copied
    !HREADY -> &&[0 <= i < log2(n-1)]
      (X HMASTER[i] <-> HMASTER[i]);
  }
}
```

Figure 2: The ARBITER component (left) and the ENCODE component (right) of the decomposed AMBA AHB arbiter.

the BUSREQ output. The encoding of the component is depicted in Figure 2 on the left. It uses straightforward formulations of the aforementioned properties in TLSF, which integrate the behavior informally described above. Note that the whole encoding is parameterized in the number of masters n.

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: Decode"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}

GLOBAL {
  DEFINITIONS {
    enum hburst =
      Single: 00
      Burst4: 10
      Incr:   01
  }
}
```

```
MAIN {
  INPUTS {
    hburst HBURST;
  }
  OUTPUTS {
    SINGLE;
    BURST4;
    INCR;
    }
  ASSERT {
    HBURST == Single -> SINGLE;
    HBURST == Burst4 -> BURST4;
    HBURST == Incr -> INCR;
    !(SINGLE && (BURST4 || INCR)) && !(BURST4 && INCR);
  }
}
```

Figure 3: The DECODE component of the decomposed AMBA AHB arbiter.

As our final example, consider the encoding of the ENCODE component depicted on the right side of Figure 2. The component identifies the master currently holding the bus via a binary number, encoded logarithmically in the number of masters. Furthermore, the component is only enabled as long as the HREADY input is high. We observe that the translation from unary to binary can be easily described using a function mapping the unary values to the corresponding binary ones. Inspecting the encoding shows that the semantics of the function are derivable straightforwardly from the declaration, due to the close relation to the equivalent mathematical representation.

6 The SyFCo Tool

We created the Synthesis Format Conversion Tool (SyFCo) [2] that can interpret the high level constructs of the format and supports transformation of the specification to plain LTL. The tool has been designed to be modular with respect to the supported output formats and semantics. Furthermore, the tool can identify and manipulate parameters, targets and semantics of a specification on the fly, and thus allows comparative studies, as it is for example needed in the reactive synthesis competition.

The main features of the tool can be summarized as follows:

- Evaluation of high level constructs in the full format to reduce full TLSF to basic TLSF.
- Transformation to other existing specification formats, like Promela LTL [1], PSL [11], Unbeast [7], Wring [27], or SLUGS [10].
- Syntactical analysis of membership in $GR(k)$ for any k, modulo boolean identities[6].
- On the fly adjustment of parameters, semantics or targets.
- Preprocessing of the resulting LTL formula, including

[6] We use the setup of [4] to identify the transition structure and the $GR(k)$ winning condition.

- conversion to negation normal form,
- replacement of derived operators, and
- pushing/pulling next, eventually, or globally operators inwards/outwards.

7 Extensions

The format remains open for further extensions, which allow more fine-grained control over the specification with respect to a particular synthesis problem. At the time of writing, the following extensions were under consideration:

- Compositionality: The possibility to separate specifications into multiple components, which then can be used as building blocks to specify larger components. E.g., it should be possible to express the whole decomposed specification, as it is depicted in Figure 1 (including the interconnections), in a single specification file in TLSF.
- Partial Implementations: a specification that is separated into multiple components might also contain components that are already implemented. Implemented components could be given in the AIGER format that is already used in SYNTCOMP [16].
- Libraries: Several functions and definitions are often shared between components, e.g., the function *mutual* of the example in Sect. 5. Hence, it is more useful to ship them via libraries instead of redeclaring them each time.
- LTL Fragment Detection: Our tool currently only supports detection of $GR(k)$. We aim to support the detection of further relevant fragments, like for example Liveness or Safety.

Acknowledgments

We thank Roderick Bloem, Rüdiger Ehlers, Bernd Finkbeiner, Ayrat Khalimov, Robert Könighofer, Nir Piterman, and Leander Tentrup for comments on TLSF and drafts of this document.

References

[1] *Promela Manual Pages (Promela LTL)*. Available at `http://spinroot.com/spin/Man/ltl.html`.

[2] *Synthesis Format Conversion Tool*. Available at `https://github.com/reactive-systems/syfco`.

[3] ARM Ltd. (1999): *AMBA specification (rev. 2)*. Available at `http://www.arm.com`.

[4] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2012): *Synthesis of Reactive(1) Designs*. *J. Comput. Syst. Sci.* 78(3), pp. 911–938, doi:10.1016/j.jcss.2011.08.007.

[5] J.R. Büchi & L.H. Landweber (1969): *Solving sequential conditions by finite-state strategies*. *Trans. Amer. Math. Soc.* 138, pp. 295–311, doi:10.2307/1994916.

[6] Alonzo Church (1962): *Logic, arithmetic and automata*. In: *Proceedings of the international congress of mathematicians*, pp. 23–35, doi:10.2307/2270398.

[7] Rüdiger Ehlers (2010): *Unbeast - Symbolic Bounded Synthesis*. Available at `https://react.cs.uni-saarland.de/tools/unbeast`.

[8] Rüdiger Ehlers (2011): *Experimental Aspects of Synthesis*. In: *iWIGP*, *EPTCS* 50, pp. 1–16, doi:10.4204/EPTCS.50.1.

[9] Rüdiger Ehlers (2012): *Symbolic bounded synthesis*. Formal Methods in System Design 40(2), pp. 232–262, doi:10.1007/s10703-011-0137-x.

[10] Rüdiger Ehlers, Vasumathi Raman & Cameron Finucane (2013): *slugs - SmalL bUt Complete GROne Synthesizer*. Available at `https://github.com/VerifiableRobotics/slugs`.

[11] Cindy Eisner & Dana Fisman (2006): *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems, Springer-Verlag, doi:10.1007/978-0-387-36123-9.

[12] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2011): *Antichains and compositional algorithms for LTL synthesis*. Formal Methods in System Design 39(3), pp. 261–296, doi:10.1007/s10703-011-0115-3.

[13] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2013): *Exploiting structure in LTL synthesis*. STTT 15(5-6), pp. 541–561, doi:10.1007/s10009-012-0222-5.

[14] Bernd Finkbeiner & Swen Jacobs (2012): *Lazy Synthesis*. In: VMCAI, LNCS 7148, Springer, pp. 219–234, doi:10.1007/978-3-642-27940-9_15.

[15] Bernd Finkbeiner & Sven Schewe (2013): *Bounded synthesis*. STTT 15(5-6), pp. 519–539, doi:10.1007/s10009-012-0228-z.

[16] Swen Jacobs (2014): *Extended AIGER Format for Synthesis*. CoRR abs/1405.5793. Available at `http://arxiv.org/abs/1405.5793`.

[17] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2016): *The First Reactive Synthesis Competition (SYNTCOMP 2014)*. STTT. Accepted for publication.

[18] Swen Jacobs, Roderick Bloem, Romain Brenguier, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup & Adam Walker (2015): *The Second Reactive Synthesis Competition (SYNTCOMP 2015)*. In: SYNT 2015, EPTCS 202, pp. 27–57, doi:10.4204/EPTCS.202.4.

[19] Swen Jacobs & Felix Klein (2016): *A High-Level LTL Synthesis Format: TLSF v1.0*. CoRR abs/1601.05228. Available at `http://arxiv.org/abs/1601.05228`.

[20] Barbara Jobstmann (2007): *Applications and Optimizations for LTL Synthesis*. Ph.D. thesis, Graz University of Technology.

[21] Uri Klein & Amir Pnueli (2010): *Revisiting Synthesis of GR(1) Specifications*. In: HVC 2010. Revised Selected Papers, LNCS 6504, Springer, pp. 161–181, doi:10.1007/978-3-642-19583-9_16.

[22] Andreas Morgenstern & Klaus Schneider (2011): *A LTL Fragment for GR(1)-Synthesis*. In: iWIGP, EPTCS 50, pp. 33–45, doi:10.4204/EPTCS.50.3.

[23] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: POPL, ACM Press, pp. 179–190, doi:10.1145/75277.75293.

[24] Michael O. Rabin (1969): *Decidability of second-order theories and automata on infinite trees*. Trans. Amer. Math. Soc. 141, pp. 1–35, doi:10.1090/S0002-9947-1969-0246760-1.

[25] Sebastian Schirmer (2015): *A Specification Format for Reactive Synthesis*. Bachelor Thesis. Saarland University, Computer Science.

[26] Saqib Sohail & Fabio Somenzi (2013): *Safety first: a two-stage algorithm for the synthesis of reactive systems*. STTT 15(5-6), pp. 433–454, doi:10.1007/s10009-012-0224-3.

[27] Fabio Somenzi & Roderick Bloem (2000): *Efficient Büchi Automata from LTL Formulae*. In: CAV, LNCS 1855, Springer, pp. 248–263, doi:10.1007/10722167_21.

A Appendix

A.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is a temporal logic, defined over a finite set of atomic propositions AP. The syntax of LTL conforms to the following grammar:

$$\varphi \;:=\; \mathit{true} \mid p \in \mathrm{AP} \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

The semantics of LTL are defined over infinite words $\alpha = \alpha_0\alpha_1\alpha_2\cdots \in (2^{\mathrm{AP}})^\omega$. A word α satisfies a formula φ at position $i \in \mathbb{N}$:

- $\alpha, i \vDash \mathit{true}$
- $\alpha, i \vDash p$ iff $p \in \alpha_i$
- $\alpha, i \vDash \neg\varphi$ iff $\alpha, i \nvDash \varphi$
- $\alpha, i \vDash \varphi_1 \vee \varphi_2$ iff $\alpha, i \vDash \varphi_1$ or $\alpha, i \vDash \varphi_2$
- $\alpha, i \vDash \mathsf{X}\varphi$ iff $\alpha, i+1 \vDash \varphi$
- $\alpha, i \vDash \varphi_1 \,\mathcal{U}\, \varphi_2$ iff $\exists n \geq i.\ \alpha, n \vDash \varphi_2$ and $\forall i \leq j < n.\ \alpha, j \vDash \varphi_1$

A word $\alpha \in 2^{\mathrm{AP}}$ satisfies a formula φ iff $\alpha, 0 \vDash \varphi$. Beside the standard operators, we have the following derived operators:

- $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$
- $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$
- $\mathsf{F}\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$
- $\mathsf{G}\varphi \equiv \neg\mathsf{F}\neg\varphi$
- $\varphi_1 \,\mathcal{R}\, \varphi_2 \equiv \neg(\neg\varphi_1 \,\mathcal{U}\, \neg\varphi_2)$
- $\varphi_1 \,\mathcal{W}\, \varphi_2 \equiv (\varphi_1 \,\mathcal{U}\, \varphi_2) \vee \mathsf{G}\varphi_1$

A.2 Mealy and Moore Automata

A Mealy automaton is a tuple $\mathcal{M}_e = (\mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda_e)$, where

- $\mathcal{I}$ is a finite set of input letters,
- $\mathcal{O}$ is a finite set of output letters,
- Q is finite set of states,
- $q_0 \in Q$ is the initial state,
- $\delta\colon Q \times \mathcal{I} \rightarrow Q$ is the transition function, and
- $\lambda_e\colon Q \times \mathcal{I} \rightarrow \mathcal{O}$ is the output function.

Hence, the output depends on the current state of the automaton and the last input letter.

A Moore automaton is a tuple $\mathcal{M}_o = (\mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda_o)$, where $\mathcal{I}, \mathcal{O}, Q, q_0$ and δ are defined as for Mealy automata. However, the output function $\lambda_o\colon Q \rightarrow \mathcal{O}$ determines the current output only on the current state of the automaton, but not the current input.

A.3 TLFS encoding of Shift, TSingle, TIncr, TBurst4 and Lock

Shift.

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: Shift"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
MAIN {
  INPUTS { HREADY; LOCKED; }
  OUTPUTS { HMASTLOCK; }
  ASSERT {
    // if HREADY is high, the component copies LOCKED to HMASTLOCK, shifted by one time step
    HREADY -> (X HMASTLOCK <-> LOCKED);
    // if HREADY is low, the old value of HMASTLOCK is copied
    !HREADY -> (X HMASTLOCK <-> HMASTLOCK);
  }
}
```

TSingle.

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: TSingle"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
MAIN {
  INPUTS { SINGLE; HREADY; LOCKED; DECIDE; }
  OUTPUTS { READY3; }
  INITIALLY {
    // initially no decision is taken
    !DECIDE;
  }
  PRESET {
    // at startup, the component is ready
    READY3;
  }
  REQUIRE {
    // decisions are only taken if the component is ready
    !READY3 -> X !DECIDE;
  }
  ASSUME {
    // a slave cannot block the bus
    G F HREADY
  }
  ASSERT {
    // for each single, locked transmission, the bus is locked for one time step
    DECIDE ->
      X[2] (((SINGLE && LOCKED) -> (!READY3 U (HREADY && !READY3 && X READY3))) &&
            (!(SINGLE && LOCKED) -> READY3));
    // the component stays ready as long as there is no decision
    READY3 && X !DECIDE -> X READY3;
    // if there is a decision the component blocks the bus for at least two time steps
    READY3 && X DECIDE -> G[1:2] ! READY3;
  }
}
```

TIncr.

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: TIncr"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
MAIN {
  INPUTS { INCR; HREADY; LOCKED; DECIDE; BUSREQ; }
  OUTPUTS { READY1; }
  INITIALLY { !DECIDE; }
  PRESET { READY1; }
  REQUIRE {
    // decisions are only taken if the component is ready
    !READY1 -> X !DECIDE;
  }
  ASSUME {
    // slaves and masters cannot block the bus
    G F HREADY && G F !BUSREQ;
  }
  ASSERT {
    // for each incremental, locked transmission, the bus is locked as long as requested
    DECIDE ->
      X[2] (((INCR && LOCKED) -> (!READY1 W (HREADY && !BUSREQ))) &&
            (!(INCR && LOCKED) -> READY1));
    // the component stays ready as long as there is no decision
    READY && X !DECIDE -> X READY1;
    // if there is a decision the component blocks the bus for at least two time steps
    READY1 && X DECIDE -> G[1:2] ! READY1;
  }
}
```

TBurst4.

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: TBurst4"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
MAIN {
  INPUTS { BURST4; HREADY; LOCKED; DECIDE; }
  OUTPUTS { READY2; }
  INITIALLY { !DECIDE; }
  PRESET { READY2; }
  REQUIRE {
    // decisions are only taken if the component is ready
    !READY2 -> X !DECIDE;
  }
  ASSUME {
    // a slave block the bus
    G F HREADY;
  }
  ASSERT {
    // for each burst4, locked transmission, the bus is locked for four time steps
    DECIDE ->
      X[2] (((BURST4 && LOCKED) -> (!READY2 U (HREADY && !READY2 && X (!READY2 U (HREADY &&
             !READY2 && X (!READY2 U (HREADY && !READY2 && X (!READY2 U (HREADY &&
             !READY2 && XREADY2)))))))) && (!(BURST4 && LOCKED) -> READY2))
    // the component stays ready as long as there is no decision
    READY2 && X !DECIDE -> X READY2;
    // if there is a decision the component blocks the bus for at least two time steps
    READY2 && X DECIDE -> G[1:2] ! READY2;
  }
}
```

Lock.

```
INFO {
  TITLE:       "AMBA AHB Arbiter"
  DESCRIPTION: "Component: Lock"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}
GLOBAL {
  PARAMETERS {
    n = 2;
  }
  DEFINITIONS {
    // mutual exclusion
    mutual(b) =
      ||[i IN {0, 1 .. (SIZEOF b) - 1}]
        &&[j IN {0, 1 .. (SIZEOF b) - 1} (\) {i}]
          !(b[i] && b[j]);
  }
}
MAIN {
  INPUTS {
    DECIDE;
    HGRANT[n];
    HLOCK[n];
  }
  OUTPUTS {
    LOCKED;
  }
  REQUIRE {
    // a every time exactely one grant is high
    mutual(HGRANT) && ||[0 <= i < n] HGRANT[i];
  }
  ASSERT {
    // whenever a decicion is taken, the LOCKED signal is updated to
    // the HLOCK value of the granted master
    &&[0 <= i < n] (DECIDE && X HGRANT[i] -> (X LOCKED <-> X HLOCK[i]));
    // otherwise, the value is copied
    !DECIDE -> (X LOCKED <-> LOCKED);
  }
}
```