`ppLTLTT`: Temporal testing for pure-past linear temporal logic formulae[⋆]

Shaun Azzopardi, David Lidell, Nir Piterman, and Gerardo Schneider

University of Gothenburg, Gothenburg, Sweden

Abstract. This paper presents `ppLTLTT`, a tool for translating pure-past linear temporal logic formulae into temporal testers in the form of automata. We show how `ppLTLTT` can be used to easily extend existing LTL-based tools, such as LTL-to-automata translators and reactive synthesis tools, to support a richer input language. Namely, with `ppLTLTT`, tools that accept LTL input are also made to handle pure-past LTL as atomic formulae. While the addition of past operators does not increase the expressive power of LTL, it opens up the possibility of writing more intuitive and succinct specifications. We illustrate this intended use of `ppLTLTT` for `Slugs`, `Strix`, and `Spot`'s command line tool `LTL2TGBA` by describing three corresponding wrapper tools `pSlugs`, `pStrix`, and `pLTL2TGBA`, that all leverage `ppLTLTT`. All three wrapper tools are designed to seamlessly fit this paradigm, by staying as close to the respective syntax of each underlying tool as possible.

Keywords: Past Linear Temporal Logic · Temporal Testers · Omega-Automata · Reactive Synthesis

1 Introduction

Linear temporal logic (LTL) is a popular choice of specification language for both the formal verification and the synthesis of programs. It has been established that LTL with past (pLTL) can be exponentially more succinct than LTL [13], and perhaps more importantly, it allows for arguably more natural specifications of real-world properties, reducing the risk of incorrectly formulating them. As a fictional but plausible example, consider a program that may flag for two different errors, represented by the variables err_1 and err_2. We may wish to express that a termination signal, represented by the variable end, should be triggered as soon as both errors have occurred, and only then. This can be done in pLTL with the formula,

$$\mathbf{G}\,((\mathbf{O}\,err_1 \land \mathbf{O}\,err_2 \land \widetilde{\mathbf{Y}}\,\mathbf{H}\,\neg end) \Leftrightarrow end). \tag{1}$$

[⋆] This research is supported by the Swedish research council (VR) project (No. 2020-04963) and the ERC Consolidator grant DSynMA (No. 772459).

An equivalent LTL formula is,

$$\begin{gathered}\mathbf{G}\,(end \Rightarrow \mathbf{X}\,\mathbf{G}\,\neg end) \wedge \\ ((\neg err_1 \wedge \neg err_2 \wedge \neg end)\,\mathbf{W}\,((err_1 \wedge ((\neg err_2 \wedge \neg end)\,\mathbf{W}\,(err_2 \wedge end))) \vee \\ (\neg err_1 \wedge \neg err_2 \wedge \neg end)\,\mathbf{W}\,((err_2 \wedge ((\neg err_1 \wedge \neg end)\,\mathbf{W}\,(err_1 \wedge end))))))).\end{gathered}$$

The above formula becomes significantly more complex when we add more errors that should trigger termination (in fact, it is factorial in the number of errors [9]). Doing the same for its pLTL counterpart only requires adding conjuncts of the form $\mathbf{O}\,(err_i)$. The past has been also suggested as a way to increase the expressiveness of fragments of LTL that can be handled more efficiently for synthesis, such as GR(1) [4].

Despite the above considerations, there is a lack of general-purpose tool support for pLTL. For example, LamaConv only allows for the translation of pLTL to two-way Büchi (or parity(3)) automata [1], which limits opportunities for further processing. FRET supports a limited notion of past in its specification language [10], and SpeAR provides a natural language interface for writing pure-past LTL requirements and checking them for logical consistency [8]. But neither FRET nor SpeAR enable the usage of pure-past LTL beyond their workflows. GOAL, with its capability to translate full pLTL to different types of ω-automata, comes close to providing general support [17]. However, while feature-rich, GOAL was designed to be used for educational purposes [18], and these translations are not implemented with performance in mind. The lack of support for past is particularly glaring for reactive synthesis tools, where it is vital to express specifications as concisely as possible, due to the high computational complexity of synthesis.

An interesting fragment of pLTL is LTL augmented atomically with pure-past LTL (ppLTL) formulae, which we call *LTL+pp*. The property of exponential succinctness of pLTL w.r.t. LTL is maintained by this fragment (the example formula that proves it for pLTL is also in LTL+pp [13], as is the example above). This fragment is arguably more intuitive than full pLTL, since it does not require reasoning that in a complex manner mixes different time directions, by disallowing the occurrence of future temporal operators under past temporal operators in the syntax tree. Moreover, as we briefly describe in Section 4, LTL+pp allows for a straightforward compositional approach to constructing corresponding automata.

In the next section, we describe the syntax and semantics of pLTL and of the fragments LTL+pp and ppLTL, and define temporal testers [15]. Following that, we describe `ppLTLTT`, a tool for generating temporal testers from ppLTL formulae, and then propose this tool as the basis for a toolchain to allow the input of existing tools for LTL-based tasks (e.g., for automata generators and reactive synthesis) to be expanded from LTL to LTL+pp. We further describe the application of our approach to `Slugs` [7], `Strix` [12,14], and `Spot`'s command line tool `LTL2TGBA` [5], and describe three corresponding wrapper tools `pSlugs`, `pStrix` and `pLTL2TGBA`, that all leverage `ppLTLTT`. We describe some experiments performed to investigate the viability of this approach. ppLTLTT and the three wrapper tools are available at GitHub[1].

2 Linear Temporal Logic with Past and Temporal Testers

Formulae of pLTL are constructed from a set of propositional variables, Boolean values and operators, and the temporal operators $\mathbf{X}$, $\mathbf{U}$, $\mathbf{Y}$, and $\mathbf{S}$.

Definition 1 (Syntax of pLTL). *Given a set of propositional variables AP, the well-formed formulae of pLTL are generated by the following grammar:*

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \mathbf{Y}\,\varphi \mid \varphi\,\mathbf{S}\,\varphi,$$

where $p \in AP$.

Formulae of pLTL are evaluated over infinite words, which are sequences of truth assignments to the variables in AP. We call such a truth assignment a *valuation*. We write $(\sigma, t) \models \varphi$ to denote that the infinite word σ models φ at time t.

Definition 2 (Semantics of pLTL). *Let $\sigma = \sigma_0\sigma_1 \cdots \in (2^{AP})^\omega$ be an infinite word over a set of propositional variables AP, φ a pLTL formula, and $t \in \mathbb{N}$. The semantic entailment relation $\models$ is defined by,*

$$\begin{array}{lcl}
(\sigma,t) \models \top & & \\
(\sigma,t) \models p & \Leftrightarrow & p \in \sigma_t \\
(\sigma,t) \models \neg\varphi & \Leftrightarrow & (\sigma,t) \not\models \varphi \\
(\sigma,t) \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & (\sigma,t) \models \varphi_1 \text{ and } (\sigma,t) \models \varphi_2 \\
(\sigma,t) \models \mathbf{X}\,\varphi & \Leftrightarrow & (\sigma,t+1) \models \varphi \\
(\sigma,t) \models \varphi_1\,\mathbf{U}\,\varphi_2 & \Leftrightarrow & \exists k \geq t\,.\,((\sigma,k) \models \varphi_2 \wedge \forall j \in [t,k)\,.\,(\sigma,j) \models \varphi_1) \\
(\sigma,t) \models \mathbf{Y}\,\varphi & \Leftrightarrow & t > 0 \text{ and } (\sigma,t-1) \models \varphi \\
(\sigma,t) \models \varphi_1\,\mathbf{S}\,\varphi_2 & \Leftrightarrow & \exists k \leq t\,.\,((\sigma,k) \models \varphi_2 \wedge \forall j \in (k,t]\,.\,(\sigma,j) \models \varphi_1)
\end{array}$$

The rest of the standard Boolean and temporal operators can be formulated in this language in the usual manner. We assume the reader is familiar with the derivation of other Boolean operators, and only present the following derived temporal operators:

$$\begin{array}{rlrl}
\mathbf{F}\,\varphi & \coloneqq \top\,\mathbf{U}\,\varphi & \mathbf{O}\,\varphi & \coloneqq \top\,\mathbf{S}\,\varphi \\
\mathbf{G}\,\varphi & \coloneqq \neg\mathbf{F}\,\neg\varphi & \mathbf{H}\,\varphi & \coloneqq \neg\mathbf{O}\,\neg\varphi \\
\varphi_1\,\mathbf{W}\,\varphi_2 & \coloneqq \varphi_1\,\mathbf{U}\,\varphi_2 \vee \mathbf{G}\,\varphi_1 & \varphi_1\,\widetilde{\mathbf{S}}\,\varphi_2 & \coloneqq \varphi_1\,\mathbf{S}\,\varphi_2 \vee \mathbf{H}\,\varphi_1 \\
\varphi_1\,\mathbf{R}\,\varphi_2 & \coloneqq \varphi_2\,\mathbf{W}\,(\varphi_1 \wedge \varphi_2) & \widetilde{\mathbf{Y}}\,\varphi & \coloneqq \mathbf{Y}\,\varphi \vee \neg\mathbf{Y}\,\top
\end{array}$$

We define the fragments of *pure-past LTL* (ppLTL) and *LTL with pure-past* subformulae as atoms (LTL+pp).

[1] https://github.com/DoppeD/ppLTLTT

Definition 3 (ppLTL and LTL+pp). *Given a set of propositional variables AP, the well-formed formulae of ppLTL (ψ) and LTL+pp (φ) are generated by the following grammar:*

$$\psi ::= \top \mid p \mid \neg\psi \mid \psi \wedge \psi \mid \mathbf{Y}\,\psi \mid \psi\,\mathbf{S}\,\psi$$
$$\varphi ::= \psi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi.$$

The semantics is the same as that of pLTL.

Definition 4 (Temporal testers). *Let φ be a ppLTL formula and z a propositional variable that does not appear in φ. A temporal tester $T_z(\varphi) = (S, s_0, \delta)$ for φ is a deterministic Büchi automaton with alphabet $2^{Var(\varphi)\cup\{z\}}$, that recognizes exactly the formula $\mathbf{G}\,(z \Leftrightarrow \varphi)$, where S is its set of states, of which all are accepting, s_0 is its initial state and δ its transition relation.*

Since temporal testers contain no sink states, the variable z acts as a monitor for the truth value of φ for every prefix of an input word. In the sequel, we will refer to z in the above definition as the *monitor variable* of the temporal tester.

We refer the reader to [15] for a more in-depth presentation of temporal testers, and of pLTL and its properties.

3 ppLTLTT

We present `ppLTLTT`, a tool that translates ppLTL formulae into temporal testers, which are output in Hanoi Omega-Automata format [3]. For example, a temporal tester for the simple formula $\varphi := \mathbf{Y}\,p$ generated by `ppLTLTT` is represented in Figure 1.

In addition to the Boolean operators $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$, and $\oplus$ (exclusive or), `ppLTLTT` supports both the primitive and the derived past operators described in Section 2.

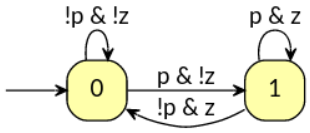

Fig. 1: Temporal tester for the formula $\varphi = \mathbf{Y}\,\psi$, generated by `ppLTLTT`.

3.1 ppLTL to Temporal Testers

The tool takes a pure-past LTL formula φ and constructs a temporal tester for it. Each state of the temporal tester corresponds to the subset of subformulae of φ that are true when a run reaches that state. Accordingly, every transition, updates the set of true subformulae after reading one more input letter. The

Algorithm 1: The ppLTLTT algorithm

```
Function BuildTT($\varphi$, $z$):
    $AP \leftarrow \mathrm{Var}(\varphi)$
    $Q, q_0, \delta \leftarrow \emptyset$
    $P_\varphi \leftarrow \{\psi \mid \mathbf{Y}\,\psi \in \mathrm{Sub}(\varphi) \vee \exists \psi_1, \psi_2 \,.\, \psi \in \{\mathbf{O}\,\psi_1, \psi_1\,\mathbf{S}\,\psi_2\} \cap \mathrm{Sub}(\varphi)\}$
    $P_\varphi \leftarrow P_\varphi \cup \{\neg\psi \mid \widetilde{\mathbf{Y}}\,\psi \in \mathrm{Sub}(\varphi) \vee \exists \psi_1, \psi_2 \,.\, \psi \in \{\mathbf{H}\,\psi_1, \psi_1\,\widetilde{\mathbf{S}}\,\psi_2\} \cap \mathrm{Sub}(\varphi)\}$
    $S \leftarrow S.\mathrm{push}(q_0)$
    while $\neg(S.\mathit{empty})$ do
        $s \leftarrow S.\mathrm{pop}$
        if $s \notin Q$ then
            $Q \leftarrow Q \cup \{s\}$
            forall $v \in 2^{AP}$ do
                $s' \leftarrow \{\psi \in P_\varphi \mid [\![\psi, s, v]\!] = \top\}$
                $S \leftarrow S.\mathrm{push}(s')$
                if $[\![\varphi, s, v]\!] = \top$ then
                    $\delta \leftarrow \delta \cup \{(s, v \cup \{z\}, s')\}$
                else
                    $\delta \leftarrow \delta \cup \{(s, v, s')\}$
        return $(Q, q_0, \delta)$
```

construction is detailed in Algorithm 1, which uses the evaluation function in Algorithm 2. Given a ppLTL formula φ, Algorithm 1 first collects all subformulae that appear immediately under a $\mathbf{Y}$ and all subformulae that appear in $\mathbf{O}$ and $\mathbf{S}$ subformulae (line 4). Moreover, it collects the negation of subformulae that appear immediately under a $\widetilde{\mathbf{Y}}$ and all subformulae that appear in $\mathbf{H}$ and $\widetilde{\mathbf{S}}$ subformulae (line 5). States of the constructed temporal tester will then be subsets of these collected formulae (P_φ), where a subformula $\psi \in P_\varphi$ is in a given state s iff all prefixes that reach s satisfy, at their final position, $\mathbf{Y}\,\psi$. For each subformula we choose the polarity that is suitable for our choice of identifying the initial state q_0 as the empty set of formulae. At the beginning of a trace before having read even the first letter, every formula of the form $\mathbf{Y}\,\psi$, $\mathbf{O}\,\psi$, or $\psi_1\,\mathbf{S}\,\psi_2$ does not hold. Conversely, every formula of the form $\widetilde{\mathbf{Y}}\,\psi$, $\mathbf{H}\,\psi$, or $\psi_1\,\widetilde{\mathbf{S}}\,\psi_2$ does hold. Thus, by choosing to follow the negations of the latter we can start with the initial state $q_0 = \emptyset$ (line 3).

The algorithm then proceeds to construct a temporal tester for φ incrementally, starting from a stack of states consisting of only the initial state. At each step, all possible transitions are considered (line 11), and for each such transition the set of formulae of P_φ that are true after the transition are collected (line 12), capturing the next state s'. This state is added to the state stack, and z added to the transition label only if the full formula φ is true at that time point (lines 14-17).

The evaluation function from Algorithm 2 is used to determine when a formula is true on a transition from a state s (see lines 12 and 14), by using the knowledge of what happened now (the transition label v) and what held

Algorithm 2: The evaluation function

```
Function $[\![\psi, s, v]\!]$:
    switch $\psi$ do
        /* We omit the cases for Boolean connectives */
        case $p$ do
            return $(p \in v)$
        case $\mathbf{Y}\,\psi_1$ do
            return $(\psi_1 \in s)$
        case $\widetilde{\mathbf{Y}}\,\psi_1$ do
            return $(\neg\psi_1 \notin s)$
        case $\mathbf{O}\,\psi_1$ do
            return $([\![\psi_1, s, v]\!] \vee (\mathbf{O}\,\psi_1 \in s))$
        case $\mathbf{H}\,\psi_1$ do
            return $([\![\psi_1, s, v]\!] \wedge (\neg\mathbf{H}\,\psi_1 \notin s))$
        case $\psi_1\,\mathbf{S}\,\psi_2$ do
            return $(([\![\psi_1, s, v]\!] \wedge (\psi_1\,\mathbf{S}\,\psi_2 \in s)) \vee [\![\psi_2, s, v]\!])$
        case $\psi_1\,\widetilde{\mathbf{S}}\,\psi_2$ do
            return $(([\![\psi_1, s, v]\!] \wedge (\neg(\psi_1\,\widetilde{\mathbf{S}}\,\psi_2) \notin s)) \vee [\![\psi_2, s, v]\!])$
```

true before (the formulae from P_φ in s). This algorithm exploits the expansion law for the temporal operators, which implies that the truth value of each (pure-past) temporal subformula at each time step is completely determined by its truth value in the previous state, together with the current valuation. For example, the expansion of the Since operator (lines 13-14 in Algorithm 2) is $\varphi_1\,\mathbf{S}\,\varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge \mathbf{Y}\,(\varphi_1\,\mathbf{S}\,\varphi_2))$. We omit the cases for the Boolean connectives in the algorithm; these are as expected.

As states are represented by subsets of subformulae of φ, the algorithm returns an automaton with at most 2^n states, where n is the size of the formula. Moreover, since the transitions from a state correspond in a one-to-one manner to valuations of the propositional variables in φ, the automaton is deterministic.

3.2 Implementation Notes

The tool is implemented in Haskell. Subformulae are collected by traversing the abstract syntax tree of the input formula. Each unique subformula with a top-level past operator is annotated with an index, as is every propositional variable. Each state (set of subformulae) is internally represented as an `Integer`[2], where bit i represents the truth value of the subformula with index i.

[2] Note that Haskell `Integer`s are of arbitrary precision; the input formula's size is only limited by the computer's memory.

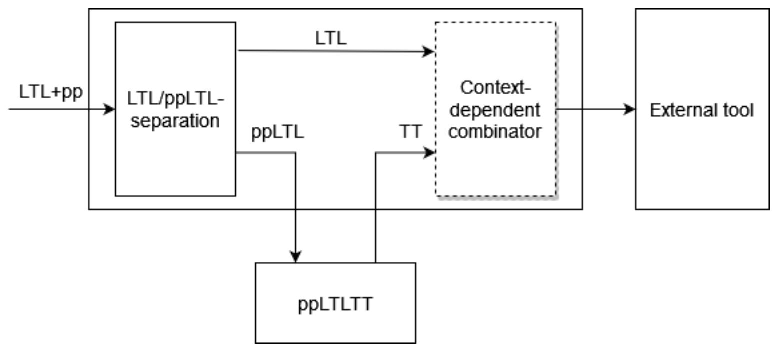

Fig. 2: The proposed toolchain.

4 Adding Past to Existing Tools

ppLTLTT is intended to be used as part of a toolchain to extend existing LTL-based tools to LTL+pp, as illustrated in Figure 2. A given LTL+pp specification is first separated into an LTL and a ppLTL part by replacing each pure-past subformula with a fresh variable. Each such subformula and its corresponding monitor variable are then translated into a temporal tester by ppLTLTT. The resulting temporal tester is combined with the LTL specification in a way that is dependent on the target tool, to which the result is then passed on. By using ppLTLTT in this way, it is possible to extend existing tools that interact with LTL specifications to support a larger fragment of pLTL, namely LTL+pp, in a straightforward manner.

As a proof of concept, we implemented this toolchain for three existing tools: Slugs, Strix, and Spot's command-line tool LTL2TGBA, which we describe below. These are (if not the best then among the best) state-of-the-art tools for different usage of LTL: Slugs handles GR(1) synthesis, Strix handles general LTL synthesis, and Spot converts LTL to automata and implements many automata transformations. We begin by explaining the encoding of the *characteristic LTL formula* of a given temporal tester.

4.1 Encoding Temporal Testers in LTL

Let $T_z(\varphi) = (S, s_0, \delta)$ be a temporal tester generated for the ppLTL formula φ, with monitor variable $z \notin \mathrm{Var}(\varphi)$ and alphabet $AP = \mathrm{Var}(\varphi) \cup \{z\}$. For every state s, let $\beta(s)$ denote its encoding as a Boolean formula[3]. We encode $T_z(\varphi)$ as

[3] The auxiliary tools described in Section 4 default to a binary encoding, but users can opt for a one-hot encoding instead.

the following LTL formula:

$$\beta(s_0) \wedge \bigwedge_{s \in S} \bigwedge_{v \in 2^{AP}} \mathbf{G}\left(\beta(s) \wedge \bigwedge_{p \in v \cap \mathrm{Var}(\varphi)} p \quad \wedge \bigwedge_{q \in \mathrm{Var}(\varphi) \setminus v} \neg q \Rightarrow [\![z]\!] \wedge \mathbf{X}\left(\beta(\delta(s, v))\right)\right) \quad (2)$$

$$\text{where } [\![z]\!] = z \text{ if } z \in v \text{ and } [\![z]\!] = \neg z \text{ otherwise}$$

Note that we here treat δ as a function, to simplify the presentation.

In practice, instead of generating complete temporal testers, we do not generate and encode transitions in which the truth value of the monitor variable z does not match the current truth value of the formula φ, which is entirely determined by the valuation $v \cap \mathrm{Var}(\varphi)$ together with the current state.

Consider the temporal tester in Figure 1. It consists of two states s_0 and s_1 (0 and 1 in the figure). Using a binary encoding, these are represented by a single propositional variable s. With the state encoding $\beta(s_0) = \neg s, \beta(s_1) = s$, the tester is encoded as,

$$\neg s \wedge \mathbf{G}(\neg s \wedge p \Rightarrow \neg z \wedge \mathbf{X}\, s) \wedge \mathbf{G}(\neg s \wedge \neg p \Rightarrow \neg z \wedge \mathbf{X} \neg s)$$
$$\wedge \mathbf{G}(s \wedge p \Rightarrow z \wedge \mathbf{X}\, s) \wedge \mathbf{G}(s \wedge \neg p \Rightarrow z \wedge \mathbf{X} \neg s).$$

4.2 Adding Past to `Slugs`: `pSlugs`

`Slugs` [7] is a tool for GR(1) synthesis [16]. It requires input GR(1) specifications written in the `slugsin` format, using prefix notation. A more syntactically expressive structured format is also available; specifications written in this format must be converted into `slugsin` before being passed to `Slugs`.

Given a `slugsin` specification, `pSlugs` converts the pure-past subformulae into temporal testers as described at the start of Section 4, and encodes each into the specification in the manner described in Section 4.1. To encode the temporal tester in `slugsin`, we allocate the required number of Boolean variables to encode the states (in binary or unary as explained). These new variables are added as output variables, while the initialization and transition invariants mentioned in Section 4.1 are added to the system's initialization and transition invariants, respectively. As `Slugs` treats specifications as well-separated [11], there is no problem with the controller breaking safety.

4.3 Adding Past to `Strix`: `pStrix`

`Strix` [12,14] is a reactive synthesis tool for full LTL. It takes as input an LTL formula and a designation of input and output variables. The corresponding wrapper tool `pStrix` is broadly identical to `pSlugs` in function and interface, but the temporal testers generated from its input are encoded directly as conjuncts in the form of Equation 2. To avoid issues with well-separation of the resulting specification, the final format of the formula given to `Strix` is the conjunction of the formulae relating to the newly allocated output variables with the LTL formula resulting from the removal of pure-past. That is, if φ is an LTL+pp formula with non-overlapping pure-past subformulae $\psi_1, \ldots, \psi_n$, then the final specification for `Strix` is $\varphi[z_1/\psi_1, ..., z_n/\psi_n] \wedge \bigwedge_{i \in [1..n]} T_{z_i}(\psi_i)$.

4.4 Adding Past to LTL2TGBA: pLTL2TGBA

LTL2TGBA [5] is a component of Spot [6]. It is a command-line tool that translates LTL formulae into various kinds of automata. Although the wrapper tool pLTL2TGBA follows the same initial steps of formula separation and conversion into temporal testers as pSlugs and pStrix, it combines the results differently. While pSlugs and pStrix syntactically manipulate the input, pLTL2TGBA works directly with the automata by making use of autfilt (another Spot command-line tool). Once the input has been separated, the LTL part is translated by LTL2TGBA into the desired automaton type. This *LTL-automaton* is then composed with the temporal testers by taking their product using autfilt. Finally, monitor variables become redundant and we instruct autfilt to remove them.

To exemplify the process, let φ be the pLTL formula presented in the introduction (1). The pure-past subformula will be replaced with a fresh variable z by pLTL2TGBA, resulting in the two formulae,

$$\begin{aligned}\varphi_f &= \mathbf{G}\,(z \Leftrightarrow \mathit{end})\\ \varphi_p &= z \Leftrightarrow \mathbf{O}\,\mathit{err}_1 \wedge \mathbf{O}\,\mathit{err}_2 \wedge \widetilde{\mathbf{Y}}\,\mathbf{H}\,\neg\mathit{end}.\end{aligned}$$

The formula φ_f is translated into an automaton $A(\varphi_f)$ by LTL2TGBA, while φ_p is translated into a temporal tester $T_z(\varphi_p)$ by ppLTLTT. The two are then combined by autfilt to obtain an automaton whose language is exactly the models of φ_f,

$$\mathbf{G}\,(z \Leftrightarrow \mathit{end}) \wedge \mathbf{G}\,(z \Leftrightarrow \mathbf{O}\,\mathit{err}_1 \wedge \mathbf{O}\,\mathit{err}_2 \wedge \widetilde{\mathbf{Y}}\,\mathbf{H}\,\neg\mathit{end}),$$

which is clearly equivalent to φ.

5 Experimental Evaluation

All experiments in this section were performed on a Dell Latitude 5420, with an Intel Core i7-1185G7 clocked at 3GHz, and 32GB of DDR4 RAM clocked at 3200MHz, running 64-bit Ubuntu 22.04.1 LTS. For comparisons with other tools we use the latest versions at time of writing. For Goal we use the version dated 2020-05-06, for Strix v.21.0.0, for Spot v.2.11.5, and for Slugs we use the code in commit dc2b1e0 from [2].

For pStrix and pSlugs we use arbiter specifications as test cases, a commonly used example for synthesis. An arbiter is conceived of as a controller granting access to resources as they are requested by clients. For n clients, there are request variables $r_1, r_2, \ldots, r_n$ and grant variables $g_1, g_2, \ldots, g_n$. The requirements of the controller are that a) only one grant is given at a time, b) it is strongly fair, and c) it will not grant access to a resource if there is no open request for it. We can express these conditions as follows:

$$\bigwedge_{i\neq j} \mathbf{G}(\neg g_i \vee \neg g_j) \wedge \bigwedge_{i}(\mathbf{GF}r_i \Rightarrow \mathbf{GF}g_i) \wedge \bigwedge_{i} \mathbf{G}(g_i \Rightarrow \mathbf{Y}(\neg g_i\,\mathbf{S}\,r_i)).$$

No. of Clients	# of Added Variables	Synthesis (s)	No. of States
Arbiter with Strong Fairness, `pStrix`			
1	2	0.03s	2
2	4	0.04s	8
3	6	0.6s	48
4	8	1.04s	384
5	10	16.37s	3840
6	12	`OOM`	N/A
Arbiter, `pSlugs`			
1	4	0.02s	6
2	8	0.03s	72
3	12	0.10s	552
4	16	1.46s	3904
5	20	34.30s	26720
6	24	714.43s	180096

Table 1: Results of synthesizing arbiters with `pStrix` and `pSlugs`.

Arbiter with Strong Fairness, `pLTL2TGBA` and Goal timings						
No. of Clients	`pLTL2TGBA`	`ltl2aut`	`ltl2aut+`	`couvreur`	`ltl2buchi`	`modella`
1	0.058s	0.646s	0.587s	0.622s	0.629s	0.651s
2	0.071s	5.108s	2.956s	9.864s	5.690s	11.779s
3	15.143s	`TO`	`TO`	`TO`	`TO`	`TO`

Table 2: Timings of translating arbiters to NBA with `pLTL2TGBA` and different algorithms of Goal.

The multi-variable strong fairness condition cannot be expressed in GR(1), however. In `pSlugs`, we replace it with $\mathbf{GF}(\neg r_i \,\widetilde{\mathbf{S}}\, g_i)$, which is equivalent to the requirement $\mathbf{G}(r_i \Rightarrow \mathbf{F} g_i)$. That is, every request should eventually be followed by a grant. It is well known how to encode future-time formulae of the latter form in GR(1) by adding an additional variable [4]. In our case, however, we use the LTL+pp equivalent, which is automatically handled by `ppLTLTT`:

$$\bigwedge_{i \neq j} \mathbf{G}(\neg g_i \vee \neg g_j) \wedge \bigwedge_i \mathbf{GF}(r_i \,\widetilde{\mathbf{S}}\, g_i) \wedge \bigwedge_i \mathbf{G}(g_i \Rightarrow \mathbf{Y}(\neg g_i \,\mathbf{S}\, r_i)).$$

Table 1 shows the result of synthesizing arbiters with a varying number of clients using `pStrix` and `pSlugs`. It shows how many variables were added to each specification, the time it took to synthesize the translated specification, and the number of controller states. `OOM` indicates that the program ran out of memory. For general reactive synthesis with `pStrix` (or Strix) `OOM` is to be expected with larger formulae, given the 2EXPTIME-complete complexity of reactive synthesis.

As Goal is the only tool that we are aware of able to translate pLTL to Büchi automata, we compare the performance of `pLTL2TGBA` to Goal in translating arbiters to nondeterministic Büchi automata, using the same specifications as for pStrix. Because Goal offers a choice of several translation algorithms, we only include the five most performant. We set a timeout of ten minutes; TO indicates that the process did not finish within this time limit. The results are shown in Table 2.

6 Conclusion

We have presented ppLTLTT, a tool for translating pure-past linear temporal logic formulae into temporal testers in the form of automata. We have integrated ppLTLTT with three existing LTL-based tools, namely Slugs, Strix and Spot's command-line tool LTL2TGBA, with the aim, among other things, of making controller synthesis more scalable. As future work we intend to optimize the encoding of temporal testers in LTL, and add features such as allowing the user to have fine grained control over how pure-past subformulae are abstracted.

References

1. Lamaconv—logics and automata converter library. https://www.isp.uni-luebeck.de/lamaconv, institute for Software Engineering and Programming Languages, University of Lübeck. Accessed on: 14th October 2022
2. Slugs. https://github.com/VerifiableRobotics/slugs, verifiable Robotics Research Group, Cornell University. Accessed on: 14th October 2022
3. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The Hanoi Omega-Automata Format. In: CAV'15. pp. 479–486. Springer (2015)
4. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
5. Duret-Lutz, A.: LTL translation improvements in Spot 1.0. International Journal on Critical Computer-Based Systems **5**(1/2), 31–54 (Mar 2014)
6. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: CAV'22. LNCS, vol. 13372, pp. 174–187. Springer (2022)
7. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) Synthesis. In: CAV'16. pp. 333–339. Springer (2016)
8. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: Spear v2.0: Formalized past ltl specification and analysis of requirements. In: NASA Formal Methods. pp. 420–426. Springer (2017)
9. Grekula, O.: SeqLTL and ωLTL — Tight witnesses for composing LTL formulas. Master's thesis, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden (2023)
10. Katis, A., Mavridou, A., Giannakopoulou, D., Pressburger, T., Schumann, J.: Capture, analyze, diagnose: Realizability checking of requirements in fret. In: CAV'22. pp. 490–504. Springer (2022)
11. Klein, U., Pnueli, A.: Revisiting synthesis of GR(1) specifications. In: HVC'10. LNCS, vol. 6504, pp. 161–181. Springer (2010)
12. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica **57**(1-2), 3–36 (2020)
13. Markey, N.: Temporal Logic with Past is Exponentially More Succinct. Bulletin-European Association for Theoretical Computer Science **79**, 122–128 (2003)
14. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: CAV'18. LNCS, vol. 10981, pp. 578–586. Springer (2018)
15. Piterman, N., Pnueli, A.: Temporal Logic and Fair Discrete Systems, pp. 27–73. Springer (2018)
16. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: VMCAI'06. pp. 364–380. Springer (2006)
17. Tsai, M.H., Tsay, Y.K., Hwang, Y.S.: Goal for games, omega-automata, and logics. In: CAV'13. pp. 883–889. Springer (2013)
18. Tsay, Y.K., Chen, Y.F., Tsai, M.H., Wu, K.N., Chan, W.C.: Goal: A graphical tool for manipulating büchi automata and temporal formulae. In: Grumberg, O., Huth, M. (eds.) TACAS'07. pp. 466–471. Springer (2007)