Proceedings of the 23rd International Conference on Principles of Knowledge Representation and Reasoning
KR in the Wild

AIGLE: A Tool for Compact, Legible AIGER Circuits from Safety Specifications

Matías Brizzio[1,2] , **Andoni Rodríguez**[1,2] , **César Sánchez**[1] , **Renzo Degiovanni**[3]
[1]IMDEA Software Institute, Spain
[2]Universidad Politécnica de Madrid, Spain
[3]Luxembourg Institute of Science and Technology (LIST), Luxembourg

Abstract

Automated logic circuit design enhances chip performance, energy efficiency, and reliability, with applications in model-checking, reactive synthesis, and hyperproperty verification. AIGER circuits are a standard format for these domains, used in hardware model-checking, synthesis competitions such as Syntcomp, symbolic synthesis algorithms, and the verification of security properties in neural networks and safety-critical systems.

Traditionally, AIGER circuits are generated from Linear Temporal Logic (LTL) specifications through complex pipelines, such as translating LTL to SMV or to automata and then to AIGER. These pipelines guarantee functional equivalence but produce large circuits with auto-generated labels that obscure the specification's meaning. In applications like symbolic reactive synthesis, model-checking, and neural network verification, understanding latches and outputs is critical for debugging and tool improvement.

In this tool paper, we introduce AIGLE, a novel tool that generates compact AIGER circuits directly from LTL[X] or Past-LTL specifications. Our approach uses linear-size translation from LTL[X] to Past-LTL, which produces highly legible circuits. Compared to tools like py-aiger, our tool reduces gate counts——often by thousands——improving readability and synthesis speed. Our empirical evaluation demonstrates smaller, more understandable circuits and faster synthesis, offering a scalable, engineer-friendly solution for formal methods applications.

1 Introduction

The AIGER format (Biere 2007; Jacobs 2014) encodes And-Inverter Graphs (AIGs) and serves as the *lingua franca* for hardware model-checking, reactive synthesis, and hyperproperty verification (Wu, Fowze, and Forte 2022; Finkbeiner 2016; Brizzio et al. 2025). AIGs are the core data structure for symbolic algorithms and the standard benchmark format in major competitions such as SYNTCOMP and HWMCC (syn ; HWM). Beyond traditional hardware verification, AIGs have become critical in emerging domains, including the verification of information-flow hyperproperties, such as non-interference, in neural networks and safety-critical systems (Langedijk, Jumelet, and Zuidema 2025; Horak et al. 2021; Schmitt et al. 2021; Hahn et al. 2020).

In these domains, Linear Temporal Logic (LTL) remains the de-facto standard for specifying safety properties—ensuring that undesirable behaviors never occur (Bonassi et al. 2025; Artale et al. 2023b; Li, Vasile, and Belta 2017). Its fragment, Past-LTL (pLTL), enhances the modularity and succinctness of specifications without increasing expressive power (Markey 2004; Bonassi et al. 2025), making it particularly suitable for defining monitors and safety envelopes in multi-agent systems. Generating AIGER circuits from LTL or pLTL specifications is a foundational step in formal verification pipelines. However, despite the ubiquity of the format, current circuit generation techniques rely on legacy pipelines (e.g., from LTL to SMV, and then to AIG) that obfuscate the relationship between the specification and the resulting circuit. Tools like `ltl2smv` (Clarke, Grumberg, and Hamaguchi 1997) (distributed with NuSMV (Cavada et al. 2005)) followed by `smvtoaig` (Biere 2007), or direct approaches like `py-aiger` (Vazquez-Chanlatte and Rabe a), produce bloated circuits with auxiliary variables and auto-generated labels disconnected from the original requirements, hampering debugging, interpretability, and downstream synthesis performance.

The closest tool, *py-aiger-past-ltl* (Vazquez-Chanlatte and Rabe b), is widely used (Wu, Fowze, and Forte 2022; Coenen et al. 2022; Rodríguez and Sánchez 2023; Hahn et al. 2020) but suffers from three limitations: (1) it lacks native support for the safety fragment LTL[X], forcing manual translation; (2) it produces redundant latches by failing to exploit structural equivalences; and (3) it lacks a formal proof of its translation soundness.

In this work, we introduce AIGLE (Aiger Generation from LTL[X] spEcifications), a tool designed to bridge the gap between high-level logical specifications and low-level circuit representations. Unlike previous approaches, AIGLE implements a direct, structure-aware translation from LTL[X] and pLTL to AIGER. By leveraging a Next-Normal Form (NNF) normalization and AST-based subformula sharing, AIGLE generates circuits that are not only sound and complete but also minimal and human-readable.

We address the following research questions:

- **(RQ1) Compactness & Interpretability:** Can a structure-aware translation significantly reduce gate/latch counts compared to existing pipelines?
- **(RQ2) Downstream Performance:** Does the reduced

circuit complexity translate to faster realizability and synthesis times?
- **(RQ3) Efficiency & Practicality:** Can AIGLE generate compact circuits efficiently enough to enable the scalable creation of modern, interpretable benchmarks?

The rest of the paper is structured as follows: Section 2 establishes the formal preliminaries. Sections 3 and 4 detail the translation algorithms and implementation. Section 5 presents the empirical evaluation, and Section 6 concludes.

1.1 Related Work

Practical Knowledge Representation. AIGLE is a tool paper that follows the spirit of recent contributions in the *KR in the Wild* track (Alviano and Reiners 2024; Geh et al. 2024; Ivliev et al. 2024; Xiang et al. 2024), which emphasizes the deployment of reasoning systems in industrial-scale or practical domains. Like these works, our focus is on bridging the gap between high-level logical formalisms and efficient, interpretable execution backends.

AIGER in Formal Methods. The AIGER format (Biere 2007) represents And-Inverter Graphs (AIGs) as directed acyclic graphs. AIGs are the backbone of modern Electronic Design Automation (EDA), supporting functional reasoning, design quality assessment, and verification (Fang et al. 2025; Liu et al. 2024; Sun et al. 2025). In the research community, they serve as the standard input for competitions (SYNTCOMP, HWMCC) and are increasingly adopted in neuro-symbolic verification to check hyperproperties like robustness and information flow (Rodríguez and Sánchez 2023; Horak et al. 2021; Wu, Fowze, and Forte 2022; Langedijk, Jumelet, and Zuidema 2025).

Translation Pipelines and Inefficiencies. Efficient translation of temporal logic to AIGER is critical for system representation. However, traditional approaches rely on multi-step, heterogeneous pipelines. For instance, tools often convert LTL to SMV (via `ltl2smv`), which is then flattened to AIG (via `smvtoaig`) (Cavada et al. 2014). Some workflows are even more convoluted, chaining `ltl2smv`, `smvtoaig`, and `fsmv2aig` (Cimatti et al. 2020). While functionally correct, these chains destroy the structural mapping between the formula and the circuit, resulting in *"bloated"* representations with thousands of auxiliary variables and opaque labels. Alternative methods based on non-deterministic Büchi automata incur exponential blow-up (Kupferman and Rosenberg 2010), while restricted languages like Speculoos (Brenguier) sacrifice expressiveness. The closest tool to AIGLE is `py-aiger-past-ltl` (Vazquez-Chanlatte and Rabe b), used as a backend in symbolic execution frameworks (Coenen et al. 2022) and in Transformer-based approaches that learn to predict satisfying traces for LTL formulas (Hahn et al. 2020). However, it requires manual translation of future safety properties into pLTL, lacks a formal proof of correctness, and empirically generates a fresh latch for every temporal operator instance, ignoring structural equivalences (see Section 5). AIGLE addresses all three limitations through a formally verified translation from LTL[X] that optimizes latch usage via syntactic normalization.

Reactive Synthesis and LTL transformation. Reactive synthesis for pLTL is EXPTIME-complete in contrast to 2EXPTIME-complete for LTL (Giacomo et al. 2020). While general LTL-to-pLTL translation incurs in an exponential blow-up (Giacomo et al. 2020; Bonassi et al. 2025), the LTL[X] fragment admits a linear-size translation (Markey 2003; Giacomo et al. 2020). The more expressive LTL[X,F] fragment, by contrast, requires a singly exponential pastification targeting $\mathbf{F}$(pLTL) (Artale et al. 2023b), which retains the future operator $\mathbf{F}$, complicates pure-pLTL toolchains such as py-aiger-past-ltl, and loses the complexity advantage. Since many safety specifications rely solely on LTL[X], AIGLE exploits this efficient, linear translation.

Circuit Interpretability. The opacity of generated circuits has become a pressing issue. A recent study (Dong et al. 2025) emphasized the need for *compact, readable circuits* to stress-test model checkers effectively. Similarly, Nazari *et al.* (Nazari, Amini, and Raghothaman 2025) and Nadeem *et al.* (Nadeem et al. 2025) have proposed *post-hoc* techniques to decompose or explain obscure AIGER circuits. AIGLE complements these efforts but shifts the paradigm from post-processing to *correct-by-construction* interpretability. By generating labeled, minimal circuits directly from the specification, we eliminate the need for costly reverse-engineering of the circuit's logic. This becomes especially relevant as Large Language Models are increasingly used to translate natural language into formal specifications, with recent work targeting LTL learning from demonstrations (Xu, Feng, and Miao 2024), dataset generation for LTL-to-natural-language translation (Ma et al. 2025), and LLM-assisted formalization pipelines (Gupta et al. 2024). Since non-experts cannot debug opaque circuits, AIGLE's named latches let logical errors in AI-generated specifications be traced back to specific sub-formulas, closing the loop between LLM output and formal verification.

2 Preliminaries

Linear Temporal Logic with Past Linear Temporal Logic (LTL) is the de-facto standard for specifying reactive systems (Pnueli 1977). We consider the safety fragment LTL[X], extended with past operators to form Past LTL (pLTL) (Lichtenstein, Pnueli, and Zuck 1985). While pLTL is expressively equivalent to LTL, it offers exponential succinctness (Markey 2003).

Let AP be a finite set of atomic propositions. The syntax of pLTL restricted to the future operators $\mathbf{X}$ (Next) and $\mathbf{G}$ (Globally) is defined by:

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi\wedge\varphi \mid \varphi\vee\varphi \mid \mathbf{X}\varphi \mid \mathbf{G}\varphi \mid \mathbf{Y}\varphi \mid \mathbf{Z}\varphi \mid \varphi\mathbf{S}\varphi$$

where $p \in \mathsf{AP}$. The semantics are defined over an infinite trace $\sigma = \sigma_0\sigma_1 \cdots \in (2^{\mathsf{AP}})^\omega$. We denote satisfaction at time $t \in \mathbb{N}$ as $(\sigma, t) \models \varphi$. Boolean connectives are interpreted as usual. The temporal operators are defined as:

$(\sigma,t) \models \mathbf{X}\varphi$ iff $(\sigma, t+1) \models \varphi$
$(\sigma,t) \models \mathbf{G}\varphi$ iff $\forall k \geq t.(\sigma,k) \models \varphi$
$(\sigma,t) \models \mathbf{Y}\varphi$ iff $t > 0$ and $(\sigma, t-1) \models \varphi$
$(\sigma,t) \models \mathbf{Z}\varphi$ iff $t = 0$ or $(\sigma, t-1) \models \varphi$
$(\sigma,t) \models \varphi\mathbf{S}\psi$ iff $\exists r \leq t.\big((\sigma,r) \models \psi \wedge \forall k \in (r,t].(\sigma,k) \models \varphi\big)$

We write $\sigma \models \varphi$ if $(\sigma, 0) \models \varphi$. A key distinction for safety specifications is between the *strong yesterday* ($\mathbf{Y}$), which is false at $t = 0$, and the *weak yesterday* ($\mathbf{Z}$), which holds vacuously at $t = 0$. This distinction enables conflict-free initialization of latches. We use standard derived operators: $\mathbf{O}\varphi \equiv \top \mathbf{S} \varphi$ (Once) and $\mathbf{H}\varphi \equiv \neg \mathbf{O} \neg \varphi$ (Historically).

Next-Normal Form (NNF) To maximize structural sharing, we normalize LTL[X] formulas into Next-Normal Form (NNF). A formula is in NNF if negations apply only to atomic propositions and $\mathbf{X}$ operators apply only to literals (atoms, negated atoms, or constants $\top, \bot$).

The conversion preserves equivalence and is performed in two steps. First, negations are pushed to the atoms using the function $n(\cdot)$. This process relies on De Morgan's laws and the self-duality of the next operator in infinite linear traces ($\neg \mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi$):

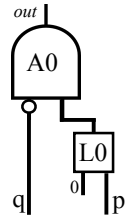
$$\begin{aligned} n(p) &= p & n(\neg(\varphi \vee \psi)) &= n(\neg\varphi) \wedge n(\neg\psi) \\ n(\neg p) &= \neg p & n(\neg(\varphi \wedge \psi)) &= n(\neg\varphi) \vee n(\neg\psi) \\ n(\mathbf{X}\varphi) &= \mathbf{X} n(\varphi) & n(\varphi \vee \psi) &= n(\varphi) \vee n(\psi) \\ n(\neg\mathbf{X}\varphi) &= \mathbf{X} n(\neg\varphi) & n(\varphi \wedge \psi) &= n(\varphi) \wedge n(\psi) \\ n(\neg\neg\varphi) &= n(\varphi) \end{aligned}$$

Second, we push the $\mathbf{X}$ operators towards the literals using the function $h(\cdot)$, exploiting the distributivity of $\mathbf{X}$ over Boolean connectives:

$$\begin{aligned} h(\mathbf{X}^k p) &= \mathbf{X}^k p & h(\mathbf{X}^k(\varphi \vee \psi)) &= h(\mathbf{X}^k\varphi) \vee h(\mathbf{X}^k\psi) \\ h(\mathbf{X}^k \neg p) &= \mathbf{X}^k \neg p & h(\mathbf{X}^k(\varphi \wedge \psi)) &= h(\mathbf{X}^k\varphi) \wedge h(\mathbf{X}^k\psi) \end{aligned}$$

This conversion preserves equivalence and yields a formula of linear size in which identical temporal obligations across different subformulas are mapped to syntactically identical structures: if two of them normalize to the same expression, they share a single latch rather than spawning multiple latches. Consider the formula $\mathbf{X}(s \vee \mathbf{X}(s \vee \mathbf{X}s))$: prior to normalization it appears to contain three nested obligations on s, each potentially requiring its own state variable. After applying h, it becomes $\mathbf{X}s \vee \mathbf{XX}s \vee \mathbf{XXX}s$, a disjunction whose literals chain through shared latches: the latch built for $\mathbf{X}s$ is reused inside $\mathbf{XX}s$, which is reused inside $\mathbf{XXX}s$, so only one fresh latch is added per level of nesting. This is the mechanism enabling the AST-based sharing described in Section 3 and the compactness results of Section 5.

And-Inverter Graphs (AIG) and AIGER Format. An And-Inverter Graph (AIG) is a directed acyclic graph (DAG) for describing a logic circuit (Biere 2007). An AIG contains AND gates, inverters, and terminal nodes for inputs and outputs. To support sequential logic, *latches* are added to AIGs. A latch is a memory element that stores a Boolean state, initialized to 0 (`false`) by default, and updates its value at each discrete time step based on a next-state function. The AIGER format provides a unified text representation for these circuits using a specific integer encoding for literals: even integers $l \in \{2, 4, \dots\}$ represent variables (inputs, latches, gates), while odd integers $l + 1$ represent their logical negation (inverters). Fig. 1 shows a circuit and its ASCII AIGER specification. The first line specifies the counts of literals and components. The second and third lines assign the inputs p and q to literals 2 and 4, respectively. The fourth line indicates a latch (implicitly literal 6) initialized to 0, with its next state defined by literal 2. This latch implements $\mathbf{Y}p$: at step $i + 1$ it holds the value of p at step i. Subsequent lines define the output (literal 8) and an AND gate taking literal 6, i.e., $\mathbf{Y}p$ and the negation of literal 4 (represented as $4 + 1 = 5$, i.e. $\neg q$) as inputs. The gate therefore computes $\mathbf{Y}p \wedge \neg q$




```
1  aag 4 2 1 1 1
2  2 // p variable
3  4 // q variable
4  6 2 0 //L0
5  8 //output
6  8 6 5 // A0
```

Figure 1: *AIGER circuit and its spec for* $\mathbf{Y}p \wedge \neg q$

In the context of synthesis, the SYNTCOMP convention for safety circuits requires the output to be `false` if the circuit complies with the formula (Jacobs 2014). For example, the formula $p \rightarrow \mathbf{X}q$, translated to pLTL as $\mathbf{Y}p \rightarrow q$, is negated and represented as $\mathbf{Y}p \wedge \neg q$ to ensure the output is `false` when safe. AIGLE adopts this standard convention.

AIGER circuits operates as a synchronous sequential machine driven by an implicit global clock. At each discrete step i: (1) the current input symbol $\sigma_i \in 2^I$ is read from the environment, (2) the combinational layer (AND-gates and inverters) computes new values from σ_i and the current latch valuation $\rho_i : L \rightarrow \mathbb{B}$, and (3) each latch $a \in L$ updates its stored value to $\rho_{i+1}(a) = f_a(\sigma_i, \rho_i)$, ready for the next step. This is the model of computation defined in Definition 1.

Definition 1 (AIGER Circuit) *An AIGER circuit is a tuple* $C = (I, L, o, f, f_{\text{and}})$*, where:*

- I *is a finite set of input signals,*
- L *is a finite set of latches,*
- o *is a single output signal,*
- $f = \{f_v : 2^I \times \mathbb{B}^{|L|} \rightarrow \mathbb{B}\}_{v \in L}$ *is a family of* next-state functions*, one per latch, each mapping the current input symbol and latch valuation to the next Boolean value of that latch, and*
- f_{and} *is a set of combinational AND gate definitions, each of the form* $g \leftrightarrow a \wedge b$ *for AIGER literals* g*,* a*,* b*.*

The circuit evaluates over an infinite trace $\sigma \in (2^I)^\omega$ *as follows. The initial latch valuation sets* $\rho_0(a) =$ `false` *for all* $a \in L$*. At each subsequent step,* $\rho_{i+1}(a) = f_a(\sigma_i, \rho_i)$ *for each latch* $a \in L$*. The output at step* i *is given by* $f_o(\sigma_i, \rho_i)$*, where* $f_o : 2^I \times \mathbb{B}^{|L|} \rightarrow \mathbb{B}$ *is the Boolean function computed by the combinational circuit rooted at* o*.*

Interpretability. In the context of automated circuit generation from temporal specifications, we define *interpretability* based on two criteria:

- **Compactness:** The circuit should minimize the number of components (gates and latches), reducing the cognitive load required to analyze the logic.

- **Semantic Mapping:** There must be a bijective correspondence between the circuit's latches and the temporal subformulas of the specification (Def 2)—e.g., a latch l_i explicitly encoding $\mathbf{Y}p$. Boolean subformulas, realized statelessly by AND-gates and inverters, do not require latches and are excluded from the mapping. The globally operator $\mathbf{G}$ of the input LTL[X] specification is unfolded into the safety condition at the output level following the SYNTCOMP convention, and thus also generates no latches.

We formalize what a *temporal subformula* is:

Definition 2 (Temporal Subformula) *A* temporal subformula *of a pLTL formula ψ is any subformula whose outermost operator is a temporal operator—that is, $\mathbf{Y}$, $\mathbf{Z}$, $\mathbf{H}$, or $\mathbf{S}$. Boolean subformulas (conjunctions, disjunctions, and negations of literals) are not temporal subformulas.*

Standard pipelines often violate these criteria by introducing auxiliary variables that lack high-level semantic meaning, obscuring the connection between the specification and the implementation.

3 AIGLE Framework

The technical contributions of this framework are: (i) the linear-size pastification of Section 3.1 and its formal correctness proof (Thm. 1); (ii) an NNF-driven AST sharing scheme that establishes a correspondence between temporal subformulas and latches; and (iii) a `step`-latch initialization compatible with the SYNTCOMP convention. We illustrate the framework on the safety LTL[X] specification $((\mathbf{X}p \vee p) \rightarrow \mathbf{XX}q) \wedge (\mathbf{X}p \rightarrow \mathbf{XX}q)$.

State-of-the-art tools, such as py-aiger-past-ltl (Vazquez-Chanlatte and Rabe b), suffer from three critical limitations: (1) they require engineers to manually translate LTL[X] to pLTL before circuit generation; (2) they fail to produce compact, legible AIGER circuits, and instead generate redundant latches and auto-generated numeric labels that bear no semantic relation to the original formula, severely hindering debugging in safety-critical systems; and (3) they violate the standard convention—used by most tools in academia and industry, including SYNTCOMP—that all latches are initialized to `false`. Instead, py-aiger-past-ltl initializes latches to `true` for operators that hold at $t = 0$ (e.g., $\mathbf{Z}$). This mismatch causes false negatives when its output is fed into other tools: the synthesizer may incorrectly declare *unrealizable* due to inconsistent initial states, directly impacting tool performance and, in case they are used to create benchmarks, the validity of these.

Fig. 3(a) shows the AIGER circuit generated by py-aiger-past-ltl for our example. The circuit uses four latches with meaningless numeric labels. The root cause is that the tool treats each nested $\mathbf{X}$ ($\mathbf{X}p$, $\mathbf{XX}q$) as a separate state variable, introducing redundant latches. In contrast, Fig. 3(b) shows AIGLE 's circuit, which uses only two latches with labels reflecting the formula's structure. AIGLE achieves this by optimizing nested temporal operators and reusing latches for repeated subformulas, leveraging pLTL backward-looking operators to reduce state variables. This example demonstrates AIGLE 's advantages: reduced latch counts (RQ1), improved legibility through meaningful labels (RQ2), and efficient translation by avoiding redundant state variables (RQ3). For larger specifications, py-aiger-past-ltl's inefficiencies scale poorly, increasing synthesis times, as shown in Section 5. AIGLE overcomes the aforementioned issues through a distinct pipeline: first it translates LTL[X] formulas to equivalent pLTL representations in *linear time*, mapping all nested $\mathbf{X}$ operators to compact state variables, then it directly converts the resulting pLTL to AIGER circuits, minimizing latches and ensuring interpretable labels via a bijection between latches and temporal subformulas; the workflow appears in Fig. 2.

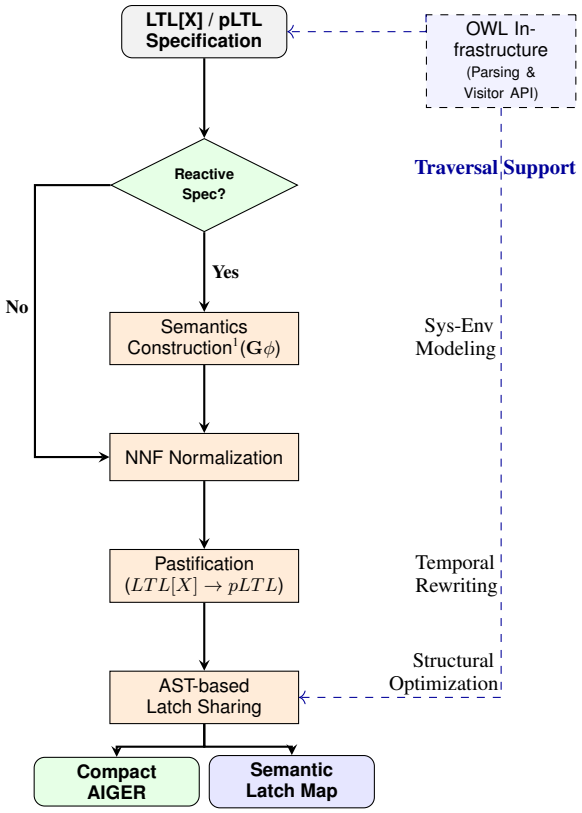


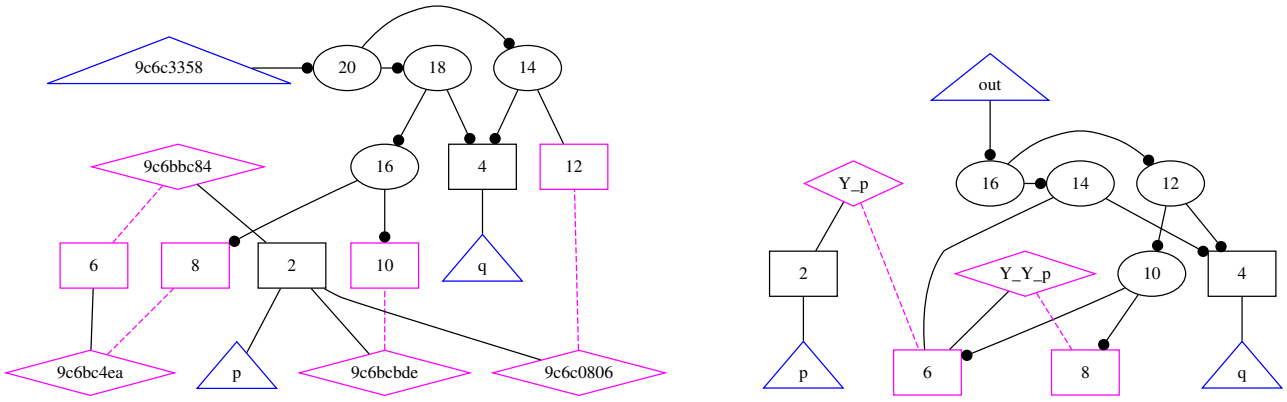
Figure 2: AIGLE detailed workflow. The tool detects reactive specifications to apply safety semantics. OWL provides the parsing infrastructure, while AIGLE performs the core temporal and structural optimizations.

3.1 Translating LTL[X] to pLTL

The first step in AIGLE 's pipeline translates an LTL[X] specification into an equivalent pLTL formula. To do that, AIGLE normalizes the LTL[X] formula to Next-Normal Form (NNF), pushing nested $\mathbf{X}^k$ operators to disjunctions and conjunctions, as detailed in Section 2. This normalization detects and caches repeated temporal subformulas, reducing latches.

For example, the non-NNF formula $\varphi_1 : \mathbf{X}(s \vee \mathbf{X}(s \vee$

[1]We use the standard semantics for reactive synthesis as described in (Meyer, Sickert, and Luttenberger 2018).



(a) Circuit generated by py-aiger-past-ltl.

(b) Circuit generated by AIGLE.

Figure 3: Comparison of size and legibility for py-aiger-past-ltl vs. AIGLE.

$\mathbf{X}s)$) is rewritten as the NNF formula $\varphi_2 : \mathbf{X}s \vee \mathbf{XX}s \vee \mathbf{XXX}s$, enabling latch reuse. Similarly, for the pLTL formula $(\mathbf{YY}p \wedge \mathbf{Y}q) \vee \mathbf{Y}(p \vee q)$, AIGLE normalizes to $(\mathbf{YY}p \wedge \mathbf{Y}q) \vee (\mathbf{Y}p \vee \mathbf{Y}q)$. Fig. 4 illustrates this optimization, showing AIGLE 's reuse of $\mathbf{Y}p$ and $\mathbf{Y}q$ nodes, reducing latches compared to py-aiger-past-ltl's redundant nodes. After normalization, AIGLE translates the LTL[X] formula into pLTL via a function $\mathcal{T}(\varphi, k)$, where the parameter k captures how far the formula looks into the future.

Definition 3 (Maximum X-nesting depth) *The* maximum $\mathbf{X}$-nesting depth $k(\varphi)$ *of an LTL[X] formula φ is the largest $m \geq 0$ such that $\mathbf{X}^m \ell$ is a subformula of φ for some literal ℓ (so $k(\varphi) = 0$ when φ contains no $\mathbf{X}$).*

The translation $\mathcal{T}(\varphi, k)$ produces a pLTL formula that, evaluated at position $t + k$, is equivalent to φ evaluated at t. It is defined recursively as:

- $\mathcal{T}(p, k) = \mathbf{Z}^k p$, for a proposition $p \in \mathsf{AP} \cup \{\top, \bot\}$
- $\mathcal{T}(\neg \ell, k) = \mathbf{Z}^k \neg \ell$, for a literal $\ell \in \mathsf{AP} \cup \{\top, \bot\}$
- $\mathcal{T}(\varphi_1 \circ \varphi_2, k) = \mathcal{T}(\varphi_1, k) \circ \mathcal{T}(\varphi_2, k)$, for $\circ \in \{\wedge, \vee\}$
- $\mathcal{T}(\mathbf{X}^m \varphi, k) = \mathbf{Z}^{k-m} \varphi$, for $0 \leq m \leq k$, where φ is a literal (p or $\neg p$) or constant

Correctness has two parts: (1) *pointwise equivalence*, $(\sigma, t) \models \varphi \iff (\sigma, t + k) \models \mathcal{T}(\varphi, k)$, accounting for the temporal shift; and (2) *compatibility* at initial positions $0 \leq j < k$, ensuring $\mathbf{G}\mathcal{T}(\varphi, k)$ holds from the trace's start. The following examples justifies the temporal shift in the translation.

Example 1 *Consider an infinite trace $\sigma = [\neg p, \neg p, p, p, \ldots]$ starting at $t = 0$. For $\varphi = \mathbf{XX}p$ with $k = 2$, the translation is $\mathcal{T}(\mathbf{XX}p, 2) = \mathbf{Z}^0 p = p$. At $t = 0$, $(\sigma, 0) \models \mathbf{XX}p$ holds since $p \in \sigma(2)$, but $(\sigma, 0) \models p$ fails since $\neg p \in \sigma(0)$. The shifted equivalence holds: $(\sigma, 0) \models \mathbf{XX}p \iff (\sigma, 2) \models p$, as $p \in \sigma(2)$. For global equivalence, $\sigma \models \mathbf{GXX}p$ requires $p \in \sigma(t + 2)$ for all $t \geq 0$, holding from position 2 onward, whereas $\mathbf{G}p$ fails at positions 0 and 1. Lemma 2 ensures initial position compatibility.*

We now introduce all lemmas and theorems that establish the *soundness and completeness* of our encoding and translation. All proofs, for the interested readers, due to lack of space, are presented in the tool web-page. The intutition is as follows. LTL[X] formulas with $\mathbf{X}$ specify future properties, while pLTL references past states. To align these, we evaluate the pLTL formula at time $t+k$, where k is the maximum $\mathbf{X}$ nesting depth. For $\varphi = \mathbf{XX}p$, at $t = 0$, this checks p at $t = 2$. Translating to $\mathbf{Z}^0 p$ and evaluating at $t + 2$ aligns the trace position. Lemma 1 ensures $(\sigma, t) \models \varphi$ if and only if $(\sigma, t + k) \models \mathcal{T}(\varphi, k)$, while initial positions $0 \leq j < k$ needed to ensure global equivalence under $\mathbf{G}$, are proven by Lemma 2.

Lemma 1 *Let φ be an LTL[X] formula with maximum $\mathbf{X}$-nesting depth k, and let $\sigma \in (2^{AP})^\omega$. For all $t \in \mathbb{N}$: $(\sigma, t) \models \varphi$ if and only if $(\sigma, t + k) \models \mathcal{T}(\varphi, k)$.*

Lemma 2 *Let φ be an LTL[X] formula with maximum $\mathbf{X}$-nesting depth k, and let $\sigma \in (2^{AP})^\omega$. For all $0 \leq j < k$: $(\sigma, j) \models \mathcal{T}(\varphi, k)$.*

Theorem 1 *Let φ be an LTL[X] formula with maximum $\mathbf{X}$-nesting depth k, and let $\sigma \in (2^{AP})^\omega$. Then: $\sigma \models \mathbf{G}\varphi$ if and only if $\sigma \models \mathbf{G}\mathcal{T}(\varphi, k)$.*

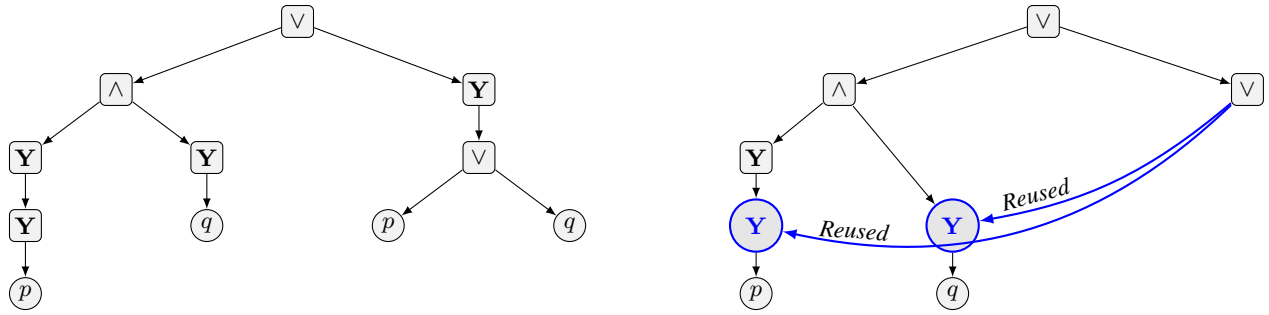
3.2 Encoding pLTL into AIGER

After obtaining a pLTL formula ψ, AIGLE constructs an AIGER circuit $C = (I, L, o, f, f_{and})$ and a semantic mapping M. The construction ensures that:

- All latches are initialized to 0 (`false`),
- The output is 0 iff ψ holds, following the SYNTCOMP convention (Jacobs 2014),
- Temp.Subformula (Def 2) sharing via AST hashing minimizes the number of latches and gates.

To bridge the gap between high-level logic and hardware, we define the translation function $[\![\varphi]\!]_C$, which returns an AIGER literal (an even integer for the signal, odd for its negation). We denote literal negation as $\bar{l} = l \oplus 1$. The circuit is extended using two core operations:

- $C.\texttt{newG}(a, b)$ creates a fresh AND gate g with inputs a and b, effectively $g \leftrightarrow a \wedge b$.
- $C.\texttt{newLat}(id, next, init)$ creates a latch $v \in L$ with the given next-state function and initial value.



(a) py-aiger-past-ltl's AST for φ.

(b) AIGLE's AST for φ, reusing latches.

Figure 4: Comparison of ASTs for $\varphi = (\mathbf{YY}p \wedge \mathbf{Y}q) \vee \mathbf{Y}(p \vee q)$, showing AIGLE's reuse of latches.

The translation is performed recursively as shown in Alg. 1. It employs a *cache* to ensure that each unique temporal subformula is translated only once, implementing AST-based sharing. This is critical for AIGLE's compactness. Additionally, it produces a mapping M that maintains the semantic link between hardware latches and their corresponding subformulas, enabling interpretability.

Alg. 1 constructs C from ψ using the `step` latch to handle initial conditions. In AIGER, all latches must initialize to 0. To encode operators like $\mathbf{Z}\varphi$ (which is `true` at $t = 0$ regardless of φ), we use the `step` latch: at $t = 0$, `step` is 0, so $\overline{C.\text{step}}$ is 1, forcing the gate to output 1. From $t = 1$ onwards, `step` becomes 1, and the circuit correctly tracks the value of the latch assigned to φ. By sharing temporal subformulas through AST hashing, AIGLE minimizes both latch count and gate density, as evidenced in Fig. 4.

4 AIGLE: Structure and Usage

AIGLE *synthesizes* compact AIGER circuits from pLTL specifications. It is implemented as a hybrid tool combining *Python 3.11* for logic processing and *Java* for external library integration (Fig. 2).

Implementation. The core translation from pLTL to AIGER is implemented in Python, providing fine-grained control over circuit construction and subformula sharing. Key components include:

- **pLTL Parsing**: We employ a **Parsing Expression Grammar (PEG)** approach via the `parsimonious` library (Rose). This choice ensures a clear correspondence between the pLTL formal grammar and the internal Abstract Syntax Tree (AST) while maintaining efficient linear-time parsing.
- **AIGER Construction**: Alg. 1 is implemented with a custom Python AIGER builder. By leveraging AST-based hashing, AIGLE ensures that equivalent temporal subformulas are mapped to the same hardware components, maximizing gate reuse.
- **LTL[X] and TLSF Support**: To support the **Temporal Logic Synthesis Format (TLSF)** (Jacobs, Klein, and Schirmer 2016)—the standard in synthesis competitions—AIGLE integrates two external tools: SYFCO (Jacobs, Klein, and Schirmer), used to extract the underlying LTL formula from TLSF, and the OWL library (Kretínský, Meggendorfer, and Sickert 2018). Crucially, while we leverage OWL as a robust infrastructure for parsing and AST manipulation (via its visitor API), the **core translation algorithm** that converts these formulas to pLTL is a native implementation of AIGLE, following the formalization in Sec. 3.1.

Algorithm 1: `EncodeToAIGER`(ψ): pLTL $\rightarrow$ AIGER

Input: pLTL [Z] formula ψ
Output: AIGER circuit C, latch map M

$\{p_0, \ldots, p_{n-1}\} \leftarrow \text{atoms}(\psi)$; $cache \leftarrow \emptyset$; $M \leftarrow \emptyset$;
$C.\text{in} \leftarrow \{p_i \mapsto 2 + 2i \mid 0 \le i < n\}$ // atoms $\rightarrow$ AIGER inputs
$C.\text{step} \leftarrow C.\texttt{newLat}(step\_id, 1, 0)$ // 0 at $t = 0$ then 1
$l \leftarrow [\![\psi]\!]_C$; $C.\text{out} \leftarrow \bar{l}$; // output 0 if ψ holds
return (C, M);

Function `Enc`(φ, C)**:** // recursive impl. of $[\![\varphi]\!]_C$
  if $cache[\varphi] \neq$ null **then return** $cache[\varphi]$;
  switch φ **do**
    case $\top$ **do** $l \leftarrow 1$;
    case $\bot$ **do** $l \leftarrow 0$;
    case $p_i \in Var$ **do** $l \leftarrow C.\text{in}[p_i]$;
    case $\neg\varphi_1$ **do** $l \leftarrow \overline{[\![\varphi_1]\!]_C}$;
    case $\varphi_1 \wedge \varphi_2$ **do**
      $l \leftarrow C.\texttt{newG}([\![\varphi_1]\!]_C, [\![\varphi_2]\!]_C)$;
    case $\mathbf{Y}\varphi_1$ **do** $l \leftarrow C.\texttt{newLat}(id, [\![\varphi_1]\!]_C, 0)$;
    case $\mathbf{H}\varphi_1$ **do**
      $l \leftarrow C.\texttt{newG}([\![\varphi_1]\!]_C, [\![\mathbf{Z}(\mathbf{H}\varphi_1)]\!]_C)$;
    case $\varphi_1 \$ \varphi_2$ **do**
      $l \leftarrow C.\texttt{newG}([\![\varphi_1]\!]_C, [\![\mathbf{Z}(\varphi_2 \$ \varphi_1)]\!]_C)$;
    case $\varphi_1 \vee \varphi_2$ **do**
      $l \leftarrow \overline{C.\texttt{newG}(\overline{[\![\varphi_1]\!]_C}, \overline{[\![\varphi_2]\!]_C})}$;
    case $\mathbf{Z}\varphi_1$ **do**
      $l \leftarrow \overline{C.\texttt{newG}(\overline{C.\texttt{newLat}(id, [\![\varphi_1]\!]_C, 0)}, \overline{C.\text{step}})}$;
  if φ *creates a latch* **then** $M \leftarrow M \cup \{id \mapsto \varphi\}$;
  $cache[\varphi] \leftarrow l$; **return** l;

Supported Input Formats. AIGLE supports two distinct input modes:

1. Forward Safety LTL (TLSF). For standard safety specifications involving the **Next** (`X`) and **Globally** (`G`) operators, AIGLE strictly adheres to the TLSF grammar (Jacobs, Klein, and Schirmer 2016).

2. Past LTL (Native). For direct specifications using past-temporal operators, AIGLE accepts formulas according to the following grammar:

```
phi ::= implies | since | or | and | unary | atom |
        "tt" | "ff" | "(" phi ")"
implies ::= "(" phi " -> " phi ")"
since ::= "(" phi " $ " phi ")"
and ::= "(" phi (" && " phi)+ ")"
or ::= "(" phi (" || " phi)+ ")"
unary ::= "Y" phi | "Z" phi | "H" phi | "~" phi
atom ::= [a-zA-Z_][a-zA-Z0-9_]* | "\"" [^\"]* "\""
```

This grammar supports **Yesterday** (`Y`), **Weak Yesterday** (`Z`), **Since** (`$`), and **Historically** (`H`), as well as standard Boolean connectives (`&&`, `||`, `->`). The tilde (`~`) denotes **Negation**. *Full parenthesization is required.* Example: `((Y p && Z (H q)) || ~Y (p || r))`.

Usage. AIGLE is invoked from the command line:

```
python3.11 aigle.py "Y p" --output circuit.
    aag
```

Key options:

`--input-type pltl` (default): Input is a pLTL formula.

`--input-type tlsf` Input is a TLSF specification; AIGLE converts it using a Java backend.

`-o, --output` Target AIGER file path.

`-m, --map-file` Generates a human-readable mapping of latches to subformulas.

`-v, --verbose` Enables debugging logs.

Example executions:

```
# Past-LTL formula
python3.11 aigle.py "(Y(p1) && Z(H(q)))"
    ↪ -o out.aag

# TLSF specification
python3.11 aigle.py --input-type tlsf
    ↪ spec.tlsf -o out.aag

# Reading formula from stdin
echo "Y(p1)" | python3.11 aigle.py
    ↪ --output out.aag
cat formula.pltl | python3.11 aigle.py
    ↪ --output out.aag
```

The resulting `.aag` file is fully compliant with SYNTCOMP tools and can be used directly by solvers such as ABC, YOSYS, or BIPAR.

5 Empirical Evaluation

We evaluate AIGLE by addressing the research questions (**RQ1**–**RQ3**) outlined in Section 1. We focus on quantifying the reduction in circuit complexity, the resulting impact on formal verification performance, and the tool's scalability.

Experimental Setup. We evaluated AIGLE following the architecture in Sec. 4. Realizability checks—used to validate the correctness of the synthesized circuits—were performed using ABSSYNTHE (Brenguier et al. 2014), a high-performance solver in the SYNTCOMP AIGER track. Experiments were conducted on Intel Xeon 2.6 GHz nodes with 16 GB RAM under GNU/Linux. To ensure reproducibility, our replication package—including the source code, a pre-configured Docker, benchmarks, and evaluation scripts—is available at: https://sites.google.com/view/aigle-tool.

RQ1: Circuit Compactness and Interpretability. AIGLE minimizes circuit size through pastification and AST-based subformula sharing (Section 3.2). We compare against `py-aiger-pltl` (Vazquez-Chanlatte and Rabe b), the current state-of-the-art for pLTL-to-AIGER translation. Since no standardized benchmarks exist for high-level LTL-to-AIGER translation (standard benchmarks are already in AIGER), we follow the methodology of recent LTL evaluations (Artale et al. 2023a; Brizzio et al. 2023) and generate 10 independent sets $(S_1, \ldots, S_{10})$ of 100 pLTL formulas each, following best practices for randomized algorithm evaluation (Arcuri and Briand 2011).

Dataset construction. Formulas are generated recursively according to the pLTL grammar: beginning from atomic propositions $p_0, p_1, \ldots$, the generator applies at each step a randomly chosen operator—unary temporal (**Y**, **Z**, **H**), binary temporal (**S**), negation, or Boolean connective—according to a fixed probability distribution. AND-type connectives are favored, reflecting the AIGER AND-only basis, while disjunction and implication appear with lower probability to exercise the De Morgan rules. Recursion terminates when the formula reaches a target operator count sampled uniformly from $[50, 200]$. The resulting benchmark contains $N = 1{,}000$ formulas (mean operator count $\mu = 125$, std. dev. $\sigma = 43$); all pLTL operators appear across the dataset, ensuring every translation rule is exercised.

Fig. 5 compares total circuit size (inputs + latches + gates), a metric used in recent interpretability studies (Dong et al. 2025; Nazari, Amini, and Raghothaman 2025; Nadeem et al. 2025). AIGLE achieves an *average reduction of 96%*. Fig. 6 breaks down the improvement: AIGLE uses **95% fewer latches** and **97% fewer AND gates**. This reduction yields circuits with *typically under 100 nodes*—orders of magnitude smaller than the output of existing tools, and small enough for direct human inspection. Together with the semantic mapping evidence (AIGLE's latches carry names like $\mathbf{Y}\_p$ and $\mathbf{Y}\_\mathbf{Y}\_p$ reflecting the temporal subformulas they encode), these results empirically validate both axes of interpretability defined in Section 2.

Ablation Analysis. To evaluate the impact of AIGLE's internal optimizations, we performed an ablation study by disabling AST-based sharing. In this configuration, AIGLE produces circuits structurally equivalent to the baseline `py-aiger-pltl`, with gate counts increasing by up to two orders of magnitude in complex formulas. This con-

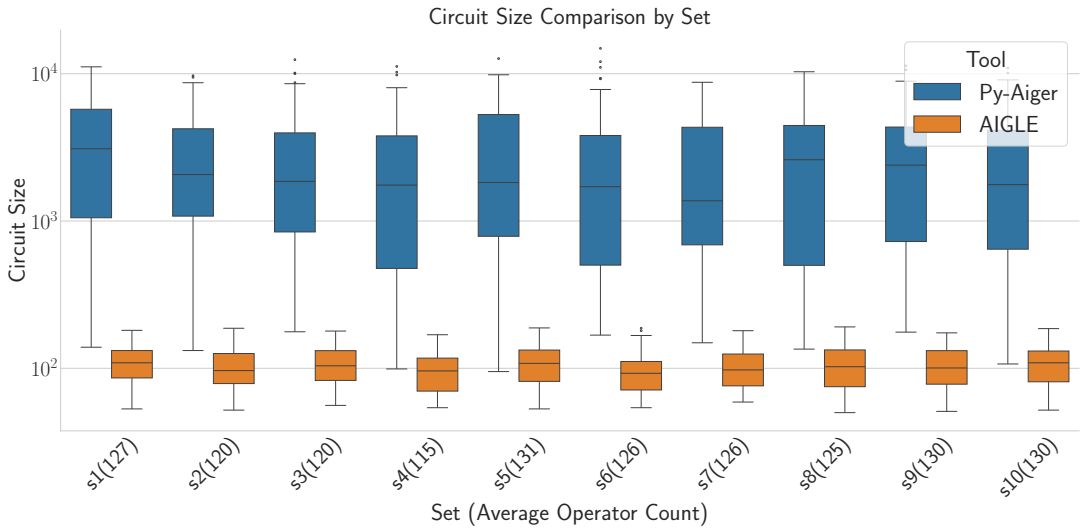


Figure 5: Circuit size (inputs + latches + gates) for 1,000 random pLTL formulas (50–200 operators) across ten independent sets.

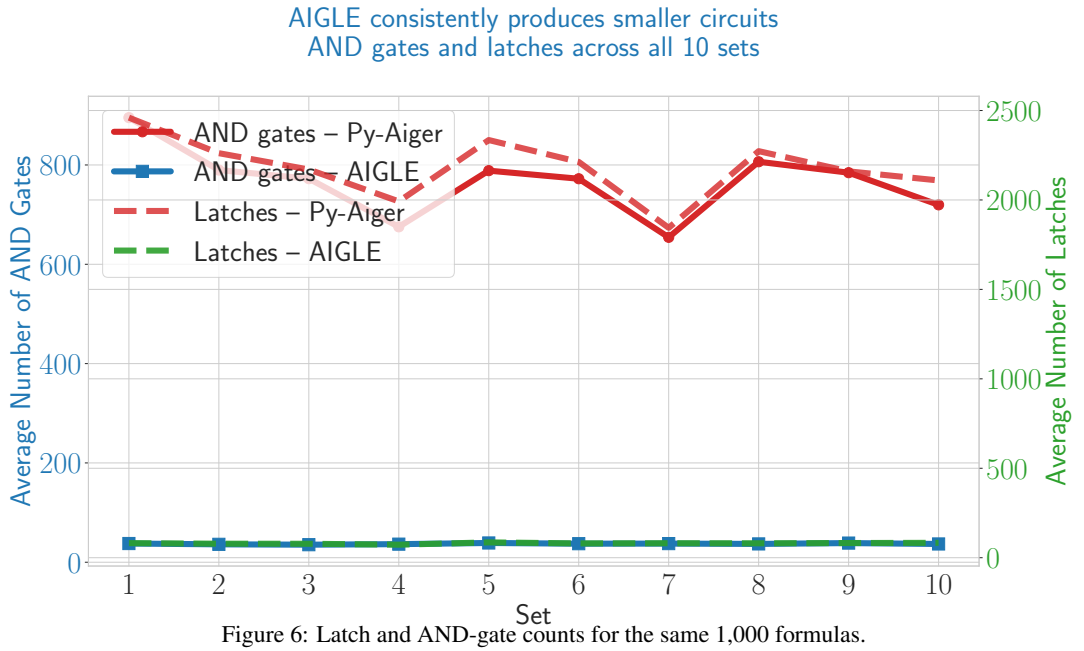


Figure 6: Latch and AND-gate counts for the same 1,000 formulas.

firms that AST-based sharing is the primary driver of the compactness results illustrated in Fig. 5. Crucially, NNF normalization (Section 2) is a key enabler for effective sharing: by pushing negations to the leaves and **X** operators toward literals, NNF ensures that semantically equivalent subformulas achieve a canonical syntactic representation in the AST.

RQ1—Circuit Compactness and Interpretability

AIGLE reduces circuit size by **96%**, latches by **95%**, and AND gates by **97%** compared to py-aiger-pltl. The resulting circuits are compact enough to support interpretability and downstream analysis.

RQ2: Realizability Performance. Figs. 7 and 8 show realizability time using ABSSYNTHE for all 1,000 formulas. Each point represents one formula, with operator count on the x-axis, synthesis time (log-scale, up to 600s) on the y-axis, and point size encoding circuit size logarithmically; sets are distinguished by hue. Marginal density plots flank the axes: the top panel shows the distribution of formula sizes per set, and the right panel the distribution of synthesis times per set. In this context, a specification φ is *realizable* if there exists a strategy (controller) that, for every input sequence, produces an execution satisfying φ; equivalently, a winning strategy exists in the induced safety game on the AIGER circuit. We focus on realizability because the AIGER format is the *de-facto* standard for reactive syn-

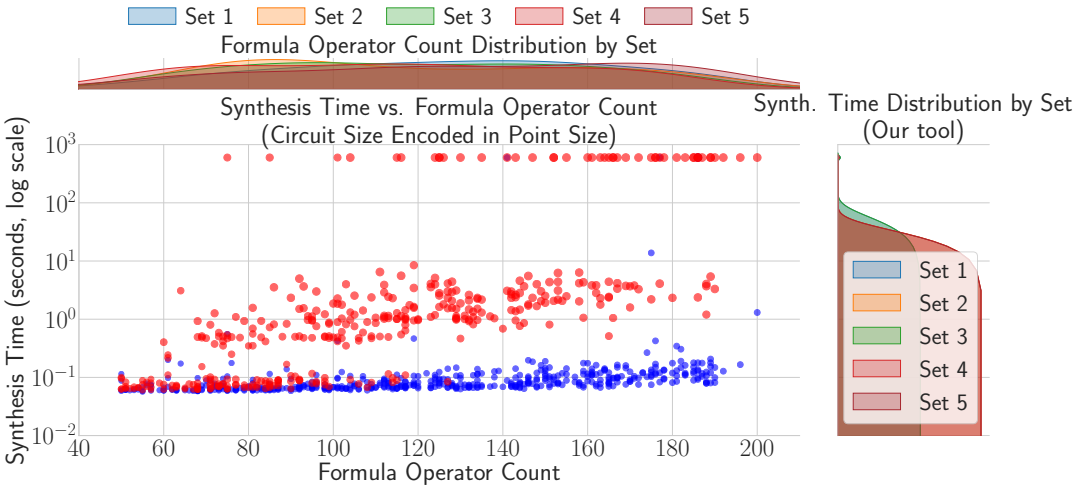


Figure 7: Sets S_1–S_5. **Blue:** AIGLE (our tool). **Red:** `py-aiger-pltl`.

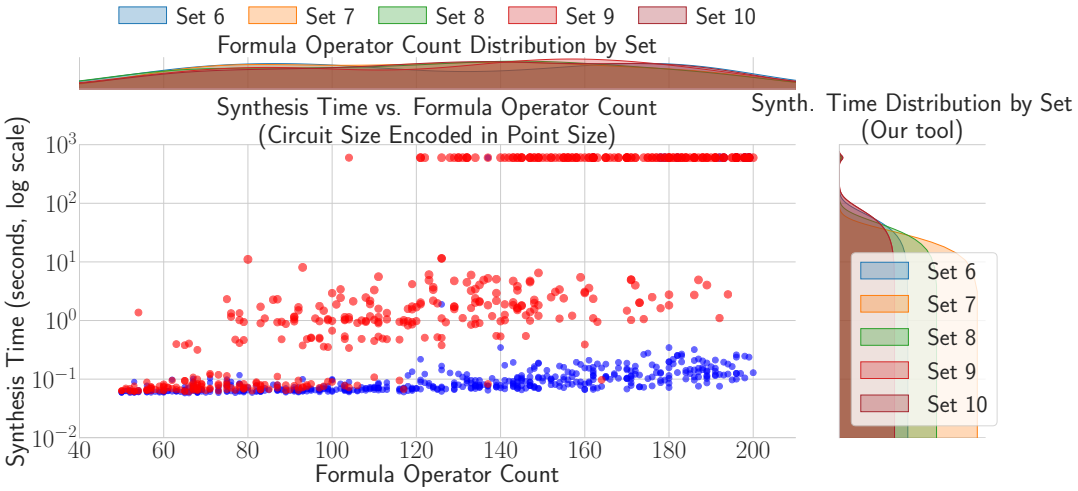


Figure 8: Sets S_6–S_{10}. **Blue:** AIGLE (our tool). **Red:** `py-aiger-pltl`.

thesis competitions, where the compactness of the monitor circuit is critical: redundant latches and gates directly enlarge the state space, often causing exponential blow-ups in solving time. By measuring realizability, we demonstrate that AIGLE's structural optimizations significantly reduce the computational burden for downstream synthesis tools.

Using AIGLE (blue), ABSSYNTHE solves *every single instance in under one second*—forming a razor-thin line at the bottom of the log-scale plot. In contrast, `py-aiger-pltl` (red) exhibits exponential growth: most formulas time out after 600 seconds, with even the fastest ones orders of magnitude slower. This yields an avg. speedup of *more than* 600×.

RQ2—Realizability Performance

AIGLE enables *sub-second realizability* for formulas up to 200 op., achieving a speedup of *more than 600*×.

RQ3: Efficiency and Practical Deployment. AIGLE generates circuits for formulas with hundreds of operators in milliseconds. This efficiency is critical not only for real-time verification but for the evolution of the field's benchmarks. Currently, most safety benchmarks in competitions like SYNTCOMP are distributed only in AIGER format. While functional, these circuits are often *"opaque"* and difficult to reverse-engineer or use for fine-grained tool analysis. By providing a direct path from high-level LTL[X] and pLTL to compact AIGER, AIGLE enables researchers to create new, traceable benchmarks. The results of RQ1 directly support this: AIGLE consistently produces circuits under 100 nodes with semantically labeled latches (Figs. 5–6), demonstrating that traceability is preserved at scale—a property absent in standard SYNTCOMP benchmarks, where the original specification is discarded.

RQ3—Efficiency and Practicality

AIGLE generates compact circuits *in milliseconds*, providing a scalable framework to modernize and expand current benchmark libraries with legible, high-level specifications.

6 Conclusion and Future Work

Conclusion. AIGER circuits are a fundamental input format in hardware verification, reactive synthesis, and hyperproperty analysis (Biere 2007; Langedijk, Jumelet, and Zuidema 2025). Yet, no tool currently offers a direct, compact translation from high-level temporal specifications supported by a formal proof of correctness. Most existing tools rely on multi-step pipelines that result in bloated, obfuscated circuits, hindering the adoption of AIGER in critical domains like AI-driven verification and autonomous systems.

This paper introduced AIGLE, a tool that generates minimal AIGER circuits from safety LTL[X] or full pLTL specifications. AIGLE provides a **provably correct** translation from LTL[X] to pLTL that ensures semantic correctness while delivering:

- significant size reductions (up to 96%) compared to the state-of-the-art (`py-aiger-pltl`),
- human-readable latch labels that provide a direct **semantic mapping** to the specification,
- efficient generation in milliseconds, significantly improving the performance of downstream synthesis and realizability solvers.

Recent studies underscore the urgent need for circuit **interpretability**—which we define as the combination of structural compactness and a clear correspondence between circuit components and temporal logic (Section 2). While downstream approaches (Nazari, Amini, and Raghothaman 2025; Nadeem et al. 2025) attempt to simplify already bloated circuits, AIGLE works *upstream*. By generating minimal, semantically labeled circuits from the start, it eliminates the need for complex post-processing and enables engineers to debug and synthesize with clarity from the first step. To our knowledge, AIGLE is the first tool providing direct LTL[X]/pLTL input, a formal proof of soundness, and high-performance execution suitable for research and industrial workflows.

Discussion and Future Work. The rise of AI-generated specifications via LLMs (Xu, Feng, and Miao 2024; Tang and Belle 2024) will flood verification pipelines with complex formulas, demanding tools that can handle the load without sacrificing clarity. AIGLE addresses this by scaling to formulas with 200+ operators in milliseconds while producing circuits suitable for real-time analysis.

Future work includes extending AIGLE to full LTL, integrating it with LLM-to-LTL pipelines for natural language synthesis, and developing visualization tools for latch-to-subformula mappings. AIGLE aims to modernize AIGER generation, offering the first compact, and practical translation framework ready for automated verification and synthesis ecosystems.

AI Declaration

No artificial intelligence-based tools or applications were used in the preparation of this manuscript. All content was produced entirely by the authors.

Acknowledgments

This work was funded in part by the DECO Project (PID2022-138072OB-I00), funded by MCIN/AEI/10.13039/501100011033/ FEDER, grant PIPF-2022/COM-24260, funded by the Madrid Regional Government and Grant Excelencia María de Maeztu CEX2024-001471-M funded by MICIU/AEI/10.13039/501100011033. Additionally, it was funded by the Luxembourg National Research Fund (FNR) PEARL program Grant agreement 16544475 and the RDI Law project "Innovations for 21st Century Assessment Authoring," financed by the Luxembourg Ministry of the Economy

References

Alviano, M., and Reiners, L. A. R. 2024. ASP Chef: Draw and Expand. In *Proc. of the 21st Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR'24)*, 720–730.

Arcuri, A., and Briand, L. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE'11)*, 1–10. ACM.

Artale, A.; Geatti, L.; Gigante, N.; Mazzullo, A.; and Montanari, A. 2023a. A Singly Exponential Transformation of LTL[X, F] into Pure Past LTL. In *Proc. of the Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR'23)*, volume 19, 65–74.

Artale, A.; Geatti, L.; Gigante, N.; Mazzullo, A.; and Montanari, A. 2023b. Complexity of Safety and coSafety Fragments of Linear Temporal Logic. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI'23)*, volume 37, 6236–6244.

Biere, A. 2007. The AIGER And-Inverter Graph (AIG) Format Version 20071012.

Bonassi, L.; Giacomo, G. D.; Favorito, M.; Fuggitti, F.; Gerevini, A. E.; and Scala, E. 2025. Planning for Temporally Extended Goals in Pure-Past Linear Temporal Logic. *Artificial Intelligence* 348:104409.

Brenguier, R.; Pérez, G. A.; Raskin, J.; and Sankur, O. 2014. AbsSynthe: abstract synthesis from succinct safety specifications. In *Proc. of the 3rd Workshop on Synthesis, (SYNT'14)*, volume 157 of *EPTCS*, 100–116.

Brenguier, R. Speculoos. https://github.com/romainbrenguier/Speculoos.

Brizzio, M.; Cordy, M.; Papadakis, M.; Sánchez, C.; Aguirre, N.; and Degiovanni, R. 2023. Automated Repair of Unrealisable LTL Specifications Guided by Model Counting. In *Proc. of of the Genetic and Evolutionary Computation Conference (GECCO'23)*, 1499–1507. ACM.

Brizzio, M.; Gorostiaga, F.; Sánchez, C.; and Degiovanni, R. 2025. Mode-Based Reactive Synthesis. In *NASA Formal Methods*, volume 15682 of *LNCS*, 116–137. Springer.

Cavada, R.; Cimatti, A.; Jochim, C. A.; Keighren, G.; Olivetti, E.; Pistore, M.; Roveri, M.; and Tchaltsev, A. 2005. NuSMV 2.4 User Manual. *CMU and ITC-irst* 11:22–45.

Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; and Tonetta, S. 2014. The nuXmv Symbolic Model Checker. In *Proc. of the 16th Int'l Conf. on Computer Aided Verification (CAV'14)*, volume 8559 of *LNCS*, 334–342. Springer.

Cimatti, A.; Geatti, L.; Gigante, N.; Montanari, A.; and Tonetta, S. 2020. Reactive Synthesis from Extended Bounded Response LTL Specifications. In *Proc. of the 20th Conf. on Formal Methods in Computer-Aided Design (FMCAD'20)*, 83–92. IEEE.

Clarke, E. M.; Grumberg, O.; and Hamaguchi, K. 1997. Another Look at LTL Model Checking. *Formal Methods in Systems Design* 1:47–71.

Coenen, N.; Dachselt, R.; Finkbeiner, B.; Frenkel, H.; Hahn, C.; Horak, T.; Metzger, N.; and Siber, J. 2022. Explaining Hyperproperty Violations. In *Proc. of the 34th Int'l Conf. on Computer Aided Verification (CAV'22)*, volume 13371 of *LNCS*, 407–429. Springer.

Dong, Y.; Xu, Y.; Deng, W.; Chen, Y.; Zhang, X.; Li, J.; Zhang, C.; and Pu, G. 2025. Diagnosing Performance Differences in Model Checkers via Runtime-Guided Problem Generation. In *Proc. of th 40th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'25)*, 129–140. IEEE Press.

Fang, W.; Wang, J.; Lu, Y.; Liu, S.; Wu, Y.; Ma, Y.; and Xie, Z. 2025. A Survey of Circuit Foundation Model: Foundation AI Models for VLSI Circuit Design and EDA. *arXiv preprint arXiv:2504.03711*.

Finkbeiner, B. 2016. Synthesis of Reactive Systems. In *Dependable Software Systems Engineering*. IOS Press. 72–98.

Geh, R. L.; Gonçalves, J.; Silveira, I. C.; Mauá, D. D.; and Cozman, F. G. 2024. dPASP: A Probabilistic Logic Programming Environment For Neurosymbolic Learning and Reasoning. In *Proc. of the 21st Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR'24)*, volume 21, 731–742.

Giacomo, G. D.; Stasio, A. D.; Fuggitti, F.; and Rubin, S. 2020. Pure-Past Linear Temporal and Dynamic Logic on Finite Traces. In *Proc. of the 29th Int'l Joint Conf. on Artificial Intelligence (IJCAI-20)*, 4959–4965. International Joint Conferences on Artificial Intelligence Organization. Survey track.

Gupta, A.; Komp, J.; Rajput, A. S.; Shankaranarayanan, K.; Trivedi, A.; and Varshney, N. 2024. Integrating Explanations in Learning LTL Specifications from Demonstrations. *arXiv preprint arXiv:2404.02872*.

Hahn, C.; Schmitt, F.; Kreber, J. U.; Rabe, M. N.; and Finkbeiner, B. 2020. Teaching Temporal Logics to Neural Networks. *arXiv preprint arXiv:2003.04218*.

Horak, T.; Coenen, N.; Metzger, N.; Hahn, C.; Flemisch, T.; Méndez, J.; Dimov, D.; Finkbeiner, B.; and Dachselt, R. 2021. Visual Analysis of Hyperproperties for Understanding Model Checking Results. *IEEE Transactions on Visualization and Computer Graphics* 28(1):357–367.

Hardware Model Checking Competition. https://hwmcc.github.io/2024/.

Ivliev, A.; Gerlach, L.; Meusel, S.; Steinberg, J.; and Krötzsch, M. 2024. Nemo: Your Friendly and Versatile Rule Reasoning Toolkit. In *Proc. of the 21st. Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR'24)*, volume 21, 743–754.

Jacobs, S.; Klein, F.; and Schirmer, S. SyFCo. https://github.com/reactive-systems/syfco.

Jacobs, S.; Klein, F.; and Schirmer, S. 2016. A High-Level LTL Synthesis Format: TLSF v1.1. *EPTCS* 229:112–132.

Jacobs, S. 2014. Extended AIGER Format for Synthesis. *arXiv preprint arXiv:1405.5793*.

Kretínský, J.; Meggendorfer, T.; and Sickert, S. 2018. Owl: A Library for ω-Words, Automata, and LTL. In *Proc. of the 16th Int'l Symp. on Automated Technology for Verification and Analysis (ATVA'18)*, volume 11138 of *LNCS*, 543–550. Springer.

Kupferman, O., and Rosenberg, A. 2010. The Blow-Up in Translating LTL to Deterministic Automata. In *Proc. of the Int'l Workshop on Model Checking and Artificial Intelligence (MoChArt'10)*, volume 6572 of *LNAI*, 85–94. Springer.

Langedijk, A.; Jumelet, J.; and Zuidema, W. 2025. Propositional Logic for Probing Generalization in Neural Networks. *arXiv preprint arXiv:2506.08978*.

Li, X.; Vasile, C.-I.; and Belta, C. 2017. Reinforcement Learning With Temporal Logic Rewards. In *Proc. of the 2017 IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems (IROS'17)*, 3834–3839. IEEE Press.

Lichtenstein, O.; Pnueli, A.; and Zuck, L. 1985. The glory of the past. In *Workshop on Logic of Programs*, 196–218. Springer.

Liu, J.; Zhai, J.; Zhao, M.; Lin, Z.; Yu, B.; and Shi, C. 2024. PolarGate: Breaking the Functionality Representation Bottleneck of And-Inverter Graph Neural Network. In *Proc. of the 43rd IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD'24)*, 1–9.

Ma, Z.; Wen, C.; Su, Z.; Liang, X.; Tian, C.; Qin, S.; and Yang, M. 2025. Bridging Natural Language and Formal Specification–Automated Translation of Software Requirements to LTL via Hierarchical Semantics Decomposition Using LLMs. In *Proc. of the 2025 40th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'25)*, 1208–1220.

Markey, N. 2003. Temporal Logic with Past is Exponentially More Succinct. *Bulletin-European Association for Theoretical Computer Science* 79:122–128.

Markey, N. 2004. Past is for Free: On the Complexity of Verifying Linear Temporal Properties with Past. *Acta Informatica* 40(6):431–458.

Meyer, P. J.; Sickert, S.; and Luttenberger, M. 2018. Strix: Explicit Reactive Synthesis Strikes Back! In *Proc. of the 30th Int'l Conf. on Computar Aided Verification (CAV'18). Part I*, volume 10981 of *LNTCS*, 578–586. Springer.

Nadeem, M.; Müller, L.; Jha, C. K.; and Drechsler, R. 2025. Advanced And-Inverter Graph Decomposition Technique for Reducing Circuit Complexity. *ACM Trans. Des. Autom. Electron. Syst.* 31(1).

Nazari, A.; Amini, M.; and Raghothaman, M. 2025. "How Does My Circuit Work?": Local Explanations for the Behavior of Sequential Circuits. In *Proc. of the 25th Conf. on Formal Methods in Computer-Aided Design (FMCAD'25)*, 8–18. IEEE.

Pnueli, A. 1977. The temporal logic of programs. In *Proc. of the 18th Annual Symp. on Foundations of Computer Science (SFCS'77)*, 46–57. IEEE.

Rodríguez, A., and Sánchez, C. 2023. Boolean Abstractions for Realizabilty Modulo Theories. In *Proc. of the 35th Int'l Conf. on Computer Aided Verification (CAV'23)*, volume 13966 of *LNCS*, 305–328. Springer, Cham.

Rose, E. Parsimonious. https://github.com/erikrose/parsimonious.

Schmitt, F.; Hahn, C.; Rabe, M. N.; and Finkbeiner, B. 2021. Neural Circuit Synthesis from Specification Patterns. *Advances in Neural Information Processing Systems* 34:15408–15420.

Sun, W.; Guo, S.; Wang, S.; Ma, Q.; and Li, H. 2025. Modeling Relational Logic Circuits for And-Inverter Graph Convolutional Network. *arXiv preprint arXiv:2508.11991*.

The Reactive Synthesis Competition. www.syntcomp.org.

Tang, W., and Belle, V. 2024. LTLBench: Towards Benchmarks for Evaluating Temporal Reasoning in Large Language Models. *arXiv preprint arXiv:2407.05434*.

Vazquez-Chanlatte, M., and Rabe, M. py-aiger. https://github.com/mvcisback/py-aiger.

Vazquez-Chanlatte, M., and Rabe, M. py-aiger-pltl. https://github.com/mvcisback/py-aiger-past-ltl.

Wu, J.; Fowze, F.; and Forte, D. 2022. EXERT: EXhaustive IntEgRiTy Analysis for Information Flow Security. In *2022 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 1–6. IEEE.

Xiang, Z.; Bienvenu, M.; Cima, G.; Gutiérrez-Basulto, V.; and Ibáñez-García, Y. 2024. ASPEN: ASP-Based System for Collective Entity Resolution. *arXiv preprint arXiv:2408.06961*.

Xu, Y.; Feng, J.; and Miao, W. 2024. Learning from Failures: Translation of Natural Language Requirements into Linear Temporal Logic with Large Language Models. In *Proc. of the IEEE 24th Int'l Conf. on Software Quality, Reliability and Security (QRS'24)*, 204–215. IEEE.