On-the-fly $\mathsf{LTL_f}$ Synthesis under Partial Observability

Nadav Alon[1], **Supratik Chakraborty**[2], **Alexandre Duret-Lutz**[3], **Dror Fried**[1], **Lucas M. Tabajara**[4], **Moshe Y. Vardi**[5], **Shufang Zhu**[6]

[1]The Open University of Israel, Israel
[2]IIT Bombay, India
[3]LRE, EPITA, Le Kremlin-Bicêtre, France
[4]Runtime Verification Inc., USA
[5]Rice University, USA
[6]University of Liverpool, UK

Abstract

$\mathsf{LTL_f}$ synthesis under partial observability requires reasoning about unobservable environment variables, which is typically handled by constructing a belief-state DFA via subset construction that universally quantifies these variables. Existing approaches perform this construction as a separate step prior to game solving, often generating belief states that are unnecessary in practice. We propose an on-the-fly approach to $\mathsf{LTL_f}$ synthesis under partial observability based on observable progression. Our method incrementally builds the belief-state DFA by progressing the specification with respect to observable variables only, universally quantifying unobservable variables on the fly. We prove the correctness of the construction and show that it naturally enables on-the-fly game solving, leading to a fully on-the-fly synthesis framework. Our implementation leverages DFAs represented using Multi-Terminal Binary Decision Diagrams: a compact representation that has proven highly effective for $\mathsf{LTL_f}$ synthesis under full observability. Experimental results demonstrate that our approach significantly outperforms existing methods and further highlight the practical benefits of integrating on-the-fly game solving with belief-state construction.

1 Introduction

A key challenge in Artificial Intelligence (AI) is enabling intelligent agents to autonomously deliberate their courses of action in order to achieve desired tasks (Reiter 2001; Ghallab, Nau, and Traverso 2016). Reasoning about actions, planning, and sequential decision making are closely connected to Formal Methods for strategic reasoning, such as reactive synthesis (Pnueli and Rosner 1989; Finkbeiner 2016; Ehlers et al. 2017). In reactive synthesis, an agent operates in an adversarial environment and must construct a strategy that guarantees task achievement under all possible environment behaviors. Within this setting, Linear Temporal Logic on finite traces ($\mathsf{LTL_f}$) synthesis has become a natural and expressive framework for specifying and automatically synthesizing correct-by-construction strategies over finite executions (De Giacomo and Vardi 2015). Moreover, $\mathsf{LTL_f}$ synthesis is closely related to (strong) planning for temporally extended goals in fully observable nondeterministic domains (Cimatti et al. 2003; Bacchus and Kabanza 2000; Calvanese, De Giacomo, and Vardi 2002; Baier, Fritz, and McIlraith 2007; Gerevini et al. 2009; De Giacomo and Rubin 2018; Camacho, Bienvenu, and McIlraith 2019).

In many realistic settings, however, agents do not have complete information about the environment in which they operate. They must therefore make decisions based on partial observations and reason about uncertainty while interacting with the environment (Rintanen 2004). As such, partial observability has been widely studied in planning and synthesis, including contingent and belief-based planning as well as reactive synthesis from logic specifications (Bonet and Geffner 2000; Hoffmann and Brafman 2005; Maliah, Komarnitski, and Shani 2022; Kupferman and Vardi 1997; De Giacomo and Vardi 2016; Tabajara and Vardi 2020).

In this paper, we focus on $\mathsf{LTL_f}$ synthesis under partial observability as described by De Giacomo and Vardi (2016). In their setting, a specification is given, in the form of an $\mathsf{LTL_f}$ formula over observable environment variables, unobservable environment variables, and system variables, which are also observable. The challenge is to construct a controller that for every history of observable environment assignments, directs how to assign the system variables, inducing a finite trace such that, no matter what the assignments for the unobservable variables are, the overall sequence meets the given specification.

De Giacomo and Vardi (2016) provide a general solution to $\mathsf{LTL_f}$ synthesis under partial observability. Given an $\mathsf{LTL_f}$ specification, their (belief-state-based) approach first constructs a deterministic finite automaton (DFA) for the formula. To handle partial observability, the DFA is lifted to a belief-state automaton via a subset construction that universally quantifies over unobservable environment variables. The synthesis problem then reduces to solving a reachability game on the resulting belief-state DFA. An MSO-based approach was suggested by Tabajara and Vardi (2020), which translates the $\mathsf{LTL_f}$ specification into a monadic second-order logic (MSO) formula with the unobservable variables universally quantified explicitly – a second-order operation that exceeds the expressive power of FOL. The approach then constructs a belief-state DFA from the MSO formula. In both approaches, synthesis ultimately requires solving a reachability game on a fully built belief-state DFA.

Through a symbolic implementation and experimental

evaluation, Tabajara and Vardi (2020) show that the MSO-based approach overall outperforms the other approaches in practice. A key reason for this is a semi-symbolic representation of DFAs (Henriksen et al. 1995), where transitions are compactly represented using Multi-Terminal Binary Decision Diagrams (MTBDDs). This representation, called MTDFA in the sequel, allows extensive sharing of common transition structures across states, resulting in significantly more effective automaton construction.

The effectiveness of MTDFA-based representations is further confirmed by recent advances in $\mathsf{LTL_f}$ synthesis (under full observability). Many state-of-the-art $\mathsf{LTL_f}$ synthesizers rely on such representations (Zhu et al. 2017; Bansal et al. 2020; De Giacomo and Favorito 2021; Zhu and Favorito 2025; Duret-Lutz et al. 2025). Using MTDFA alone, however, has its limitations. The top-ranked $\mathsf{LTL_f}$ synthesizer in the 2025 reactive synthesis competition[1], `ltlfsynt`, combines the MTDFA-based representation with *on-the-fly* techniques, incrementally constructing the MTDFA and integrating this construction with game solving (Duret-Lutz et al. 2025). This demonstrates that a tight integration of MTDFA-based on-the-fly automaton construction and on-the-fly game solving can substantially improve performance in practice.

This naturally raises the question of whether the same combination of techniques can be brought to $\mathsf{LTL_f}$ synthesis under partial observability. While MTDFAs provide a powerful technique for compact automaton representation and on-the-fly construction, partial observability introduces an additional challenge: universal quantification over unobservable environment variables. In all existing approaches to $\mathsf{LTL_f}$ synthesis under partial observability, such as that of Tabajara and Vardi (2020), this quantification is handled conceptually as a separate step, rather than being operationally integrated with automaton construction and game solving.

In this paper, we address this challenge by bringing together the key techniques that have proven successful in $\mathsf{LTL_f}$ synthesis under full observability, with complete operational integration of MTDFAs and universal quantification. We first propose an on-the-fly construction of belief-state DFAs from $\mathsf{LTL_f}$ specifications, which we call *observable progression*. This technique incrementally builds an automaton for the $\mathsf{LTL_f}$ specification via formula progression, while at the same time universally quantifying over unobservable environment variables, therefore directly resulting in a belief-state DFA. This construction enables integrating game solving directly during belief-state DFA construction. Thus, belief states are generated incrementally and only when needed, providing an on-the-fly approach to $\mathsf{LTL_f}$ synthesis under partial observability, which still preserves the 2EXPTIME worst-case complexity, as for full-observability. Furthermore, to harness the efficiency of MTDFA, we implement our tool using the MTDFA representation, in which belief states are obtained naturally as MTBDD terminals. In particular, with careful variable ordering, our MTDFA-based belief-state construction does not incur any additional blowup. This construction allows us to further integrate on-the-fly game solving on the same MTDFA representation. To summarize, our approach enables the automaton construction, the universal quantification, and the game solving, all to be performed on the fly and directly on MTBDDs.

Our implementation resulted in a modified version of `ltlfsynt` that we call `ltlfsynt-po`, which can now handle $\mathsf{LTL_f}$ synthesis under partial observability. Empirical evaluation showed that `ltlfsynt-po` dramatically outperforms the tools from Tabajara and Vardi (2020), which so far has represented the state of the art for partial-observability $\mathsf{LTL_f}$ synthesis. In addition, we show how each of the phases, belief-state DFA via observable progression, and on-the-fly game solving, contributes to the performance of `ltlfsynt-po`. Detailed analysis demonstrates the practical benefits of enabling on-the-fly game solving.

2 Preliminaries

2.1 Words over Assignments

Let Σ be a finite alphabet. We view a word over Σ as a sequence of letters indexed by positions: an *infinite word* is a function $\sigma : \mathbb{N} \to \Sigma$, whereas a *finite word* of length n is a function $\sigma : \{0, 1, \ldots, n-1\} \to \Sigma$. The collection of all infinite words is denoted Σ^ω. For each $n \in \mathbb{N}$, the set of length-n words is Σ^n, and we let $\Sigma^\star$ (resp. Σ^+) range over all finite words of length at least 0 (resp. strictly positive). For a finite word σ, its length is written $|\sigma|$. If $\sigma \in \Sigma^n$ and $0 \leq i < n$, we write $\sigma(..i)$ for the prefix of σ ending at position i. Note that $\sigma(..i)$ is of length $i+1$. Similarly, we denote $\sigma(i..)$ for the suffix starting at position i; the same notation applies to $\sigma \in \Sigma^\omega$ in the natural way.

Let $\mathcal{P}$ be a finite set of Boolean variables (equivalently, *atomic propositions*). An *assignment* for $\mathcal{P}$ is a mapping of variables to values in $\mathbb{B} = \{\bot, \top\}$, represented as $w : \mathcal{P} \to \mathbb{B}$. We use $\mathbb{B}^{\mathcal{P}}$ to denote the set of all such assignments. When $\mathcal{P}_1$ and $\mathcal{P}_2$ are disjoint variable sets, any pair of assignments $w_1 \in \mathbb{B}^{\mathcal{P}_1}$ and $w_2 \in \mathbb{B}^{\mathcal{P}_2}$ can be combined into a single assignment over $\mathcal{P}_1 \cup \mathcal{P}_2$, written $w_1 \sqcup w_2$, by agreeing with w_1 on $\mathcal{P}_1$ and with w_2 on $\mathcal{P}_2$. Formally, we have $(w_1 \sqcup w_2)(v) = w_1(v)$ for $v \in \mathcal{P}_1$ and $(w_1 \sqcup w_2)(v) = w_2(v)$ for $v \in \mathcal{P}_2$.

We use these objects to encode synchronous, discrete-time Boolean signal models. Assigning one variable in $\mathcal{P}$ to each signal, the global behavior over time is represented by a finite word $\sigma \in (\mathbb{B}^{\mathcal{P}})^+$ whose letters are assignments of $\mathcal{P}$ at successive time steps. The operator $\sqcup$ is lifted pointwise to such words: for disjoint $\mathcal{P}_1, \mathcal{P}_2$ and words $\sigma_1 \in (\mathbb{B}^{\mathcal{P}_1})^n$, $\sigma_2 \in (\mathbb{B}^{\mathcal{P}_2})^n$ of the same length n, we define $\sigma_1 \sqcup \sigma_2 \in (\mathbb{B}^{\mathcal{P}_1 \cup \mathcal{P}_2})^n$ by $(\sigma_1 \sqcup \sigma_2)(i) = \sigma_1(i) \sqcup \sigma_2(i)$ for every i such that $0 \leq i < n$.

2.2 Linear Temporal Logic over Nonempty Finite Words.

We use classical $\mathsf{LTL_f}$ semantics over nonempty finite words (De Giacomo and Vardi 2013).

Definition 1 ($\mathsf{LTL_f}$ formulas)**.** *An $\mathsf{LTL_f}$ formula φ is built from a set $\mathcal{P}$ of atomic propositions, using the following grammar, where $p \in \mathcal{P}$, and $\odot \in \{\wedge, \vee, \to, \leftrightarrow, \ldots\}$ is any*

[1] See https://www.syntcomp.org/syntcomp-2025-results/.

Boolean operator: $\varphi ::= tt \mid ff \mid p \mid \neg\varphi \mid \varphi \odot \varphi \mid \mathsf{X}\varphi \mid \mathsf{X}^!\varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \mid \varphi \,\mathsf{U}\, \varphi \mid \varphi \,\mathsf{R}\, \varphi$.

Symbols tt and ff represent the true *and* false $\mathsf{LTL_f}$ *formulas. Other than the usual Boolean operators, we have the temporal operators* X *(weak next),* $\mathsf{X}^!$ *(strong next),* F *(finally),* G *(globally),* U *(until), and* R *(release). Let* $\mathsf{LTL_f}(\mathcal{P})$ *denote the set of all formulas produced by the above grammar. For* $\varphi \in \mathsf{LTL_f}(\mathcal{P})$*, we use* $\mathsf{sf}(\varphi)$ *to denote the set of all sub-formulas of* φ*. A* maximal temporal subformula *of* φ *is a subformula whose primary operator is temporal and that is not strictly contained within any other temporal subformula of* φ*.*

The semantics of $\mathsf{LTL_f}$ *defines when a formula* $\varphi \in \mathsf{LTL_f}(\mathcal{P})$ *is satisfied by a word* $\sigma \in (\mathbb{B}^{\mathcal{P}})^n$ *at position* $0 \leq i < n$*. We denote this by* $\sigma, i \models \varphi$*, and define it as follows.*

$$\sigma, i \models tt \iff i < n \qquad \sigma, i \models p \iff p \in \sigma(i)$$
$$\sigma, i \models ff \iff i = n \qquad \sigma, i \models \neg\varphi \iff \neg(\sigma, i \models \varphi)$$
$$\sigma, i \models \varphi_1 \odot \varphi_2 \iff (\sigma, i \models \varphi_1) \odot (\sigma, i \models \varphi_2)$$
$$\sigma, i \models \mathsf{X}\varphi \iff (i+1 = n) \vee (\sigma, i+1 \models \varphi)$$
$$\sigma, i \models \mathsf{X}^!\varphi \iff (i+1 < n) \wedge (\sigma, i+1 \models \varphi)$$
$$\sigma, i \models \mathsf{F}\varphi \iff \exists j \in [i, n),\, \sigma, j \models \varphi$$
$$\sigma, i \models \mathsf{G}\varphi \iff \forall j \in [i, n),\, \sigma, j \models \varphi$$
$$\sigma, i \models \varphi_1 \,\mathsf{U}\, \varphi_2 \iff \exists j{\in}[i, n),\, \big((\sigma, j \models \varphi_2) \wedge (\forall k{\in}[i, j),\, \sigma, k \models \varphi_1)\big)$$
$$\sigma, i \models \varphi_1 \,\mathsf{R}\, \varphi_2 \iff \forall j{\in}[i, n),\, \big((\sigma, j \models \varphi_2) \vee (\exists k{\in}[i, j),\, \sigma, k \models \varphi_1)\big)$$

The set of words that satisfy $\varphi \in \mathsf{LTL_f}(\mathcal{P})$ *is represented by* $\mathscr{L}(\varphi) = \{\sigma \in (\mathbb{B}^{\mathcal{P}})^+ \mid \sigma, 0 \models \varphi\}$*.*

Example 1. *Consider the following* $\mathsf{LTL_f}$ *formula over* $\mathcal{P} = \{u, i, o\}$*:* $\Psi = \big(\mathsf{GF}u \rightarrow \mathsf{F}(i \leftrightarrow o)\big) \wedge \big(\mathsf{GF}\neg u \rightarrow \mathsf{F}(i \vee o)\big)$*. Suppose u and i are inputs, and o is the output of a system. The above formula specifies that if u is set in the last step, then o must eventually have the same value as i. Otherwise, at least one of i or o must eventually be set, although it is not necessary for them to have the same value.*

Definition 2 (Propositional Equivalence (Esparza, Křetínský, and Sickert 2018))**.** *For* $\varphi \in \mathsf{LTL_f}(\mathcal{P})$*, let* φ_P *be the Boolean formula obtained from* φ *by replacing every maximal temporal subformula* ψ *by a Boolean variable* x_ψ*. Two formulas* $\varphi_1, \varphi_2 \in \mathsf{LTL_f}(\mathcal{P})$ *are* propositionally equivalent*, denoted* $\varphi_1 \sim \varphi_2$*, if* ${\varphi_1}_P$ *and* ${\varphi_2}_P$ *are semantically equivalent Boolean formulas.*

Note that if $\varphi_1 \sim \varphi_2$, then $\mathscr{L}(\varphi_1) = \mathscr{L}(\varphi_2)$, but the converse is not true in general. We use $[\varphi]_\sim \in \mathsf{LTL_f}(\mathcal{P})$ to denote some unique representative of the equivalence class of φ with respect to $\sim$.

2.3 $\mathsf{LTL_f}$ Synthesis and Realizability

Consider a setting where the desired behavior of a system interacting with an environment is specified by an $\mathsf{LTL_f}$ formula over variables representing inputs of the environment and outputs of the system. The synthesis problem asks us to design a *controller* (aka. *strategy/plan*) that observes the history of inputs and determines the outputs at every time step, such that the specification is satisfied for every input sequence produced by the environment. Not all system inputs may be observable or even usable by the controller. This can happen if some inputs are noisy or cannot be measured reliably enough. Therefore, it makes sense to partition the input variables into two disjoint sets $\mathcal{I}$ and $\mathcal{U}$, representing observable and unobservable inputs, respectively. Let $\mathcal{O}$ be the set of output variables. In general, the desired system behavior is specified by a formula $\varphi \in \mathsf{LTL_f}(\mathcal{I} \uplus \mathcal{U} \uplus \mathcal{O})$.

We use the *terminating transducer* interpretation of controllers, originally introduced by Bansal et al. (2023). As observed by Jacobs, Perez, and Schlehuber-Caissier (2026), this is a natural interpretation in the context of $\mathsf{LTL_f}$ synthesis, since we expect a controller to know when a task is completed, and therefore when to stop interacting with the environment. The controller itself may be represented as a deterministic Mealy or Moore machine with input alphabet $\mathbb{B}^{\mathcal{I}}$, output alphabet $\mathbb{B}^{\mathcal{O}}$, and a distinguished subset of states designated as *terminating states*. With Mealy semantics, the controller can access the past and also current inputs to determine its current output; with Moore semantics, it can access only the past inputs. The controller M induces a function $\rho_M : (\mathbb{B}^{\mathcal{I}})^* \rightarrow (\mathbb{B}^{\mathcal{O}})$ that maps a history of assignments of observable input variables to the current assignment of output variables. Given an input word $\sigma \in (\mathbb{B}^{\mathcal{I}})^n$ of length n, the controller produces a corresponding word of n output assignments, denoted $\sigma_{\rho_M} \in (\mathbb{B}^{\mathcal{O}})^n$. If σ leads the underlying machine to a terminating state, the system is assumed to stop interacting with the environment after reading σ.

Synthesis under full observability: To understand when a controller M *realizes* a specification φ, we first consider the case when all inputs of the environment are observable by the controller, i.e., $\mathcal{U} = \emptyset$.

Definition 3 ((Bansal et al. 2023; Jacobs, Perez, and Schlehuber-Caissier 2026))**.** *A controller* M *realizes an* $\mathsf{LTL_f}$ *specification* φ *if for every word* $\sigma \in (\mathbb{B}^{\mathcal{I}})^\omega$ *there exists a position* $k \geq 0$ *such that (a)* M *enters a terminating state after reading the input sequence* $\sigma(..k)$*, and (b)* $(\sigma \sqcup \sigma_{\rho_M})(..k) \in \mathscr{L}(\varphi)$*.*

If we view the interaction between the system and the environment as a two-player game, then Definition 3 implies the following. For every input sequence produced by the environment, the controller generates an output at each step based on the input history so far. It does so such that, after finitely many steps, the play reaches a terminating state of the transducer. When the resulting finite interaction sequence satisfies the $\mathsf{LTL_f}$ specification φ, the system player chooses to terminate the play. The first player is chosen according to the desired semantics (Mealy or Moore).

Synthesis under partial observability: If $\mathcal{U} \neq \emptyset$, the situation becomes more interesting. While the controller M generates its output sequence by observing only the inputs on $\mathcal{I}$, satisfaction of φ depends on the values on $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{U}$. This motivates the following definition of realizability under partial observation (De Giacomo and Vardi 2016).

Definition 4. *A controller* M *implementing the function* $\rho_M : (\mathbb{B}^{\mathcal{I}})^* \rightarrow \mathbb{B}^{\mathcal{O}}$ *realizes a specification* $\varphi \in \mathsf{LTL_f}(\mathcal{I} \uplus$

$\mathcal{U} \uplus \mathcal{O}$) *if for any word* $\sigma \in (\mathbb{B}^{\mathcal{I}})^\omega$ *there exists a position* $k \geq 0$ *such that (a)* M *enters a terminating state after reading the input sequence* $\sigma(..k)$*, and (b) for every* $\sigma_{\mathcal{U}} \in (\mathbb{B}^{\mathcal{U}})^k$*, the word* $\sigma_{\mathcal{U}} \sqcup (\sigma \sqcup \sigma_{\rho_M})(..k) \in \mathscr{L}(\varphi)$.

The alternation of quantifiers in the above definition deserves careful attention. Intuitively, once the controller decides to terminate the play (by entering a terminating state), it must be possible to satisfy φ by augmenting $(\sigma \sqcup \sigma_{\rho_M})(..k)$ with any arbitrary sequence of valuations of $\mathcal{U}$ of length $k+1$. Note that we do not pick a separate k for each word in $(\mathbb{B}^{\mathcal{U}})^\omega$. Instead, once k is chosen, the same choice works in condition (b) of Definition 4 for all words in $(\mathbb{B}^{\mathcal{U}})^k$. This is why, despite the lack of visibility of $\mathcal{U}$, the outputs generated by the controller allow the system to win the game, irrespective of how the environment assigns values to $\mathcal{U}$.

We say a specification $\varphi \in \mathsf{LTL_f}(\mathcal{I} \uplus \mathcal{U} \uplus \mathcal{O})$ is *realizable under full (resp. partial) observability* if there exists a controller M that realizes φ in the sense of Definition 3 (resp. Definition 4). The synthesis problem for $\mathsf{LTL_f}$ under full (resp. partial) observability is the task of algorithmically generating a controller M that realizes φ. The computational complexity of both problems is known to be 2EXPTIME-complete (De Giacomo and Vardi 2015; De Giacomo and Vardi 2016).

Referring to Example 1, if both inputs i and u are observable, the specification is realizable in one step by a Mealy machine controller; however, with only i observable, two steps are needed. We discuss this in detail in Section 4.

Some important notions used in later sections: The notions of formula progression (Xiao et al. 2021; De Giacomo et al. 2022; Xiao et al. 2025; Duret-Lutz et al. 2025), belief-state automata construction (De Giacomo and Vardi 2016), and on-the-fly interleaved construction of automata and game-solving (Xiao et al. 2021; De Giacomo et al. 2022; Xiao et al. 2024; Favorito 2023; Li et al. 2025; Duret-Lutz et al. 2025) have been used in earlier work in the context of $\mathsf{LTL_f}$ synthesis under full observability. We introduce non-trivial adaptations of these techniques to work in the presence of partial observability. Details of these techniques and our adaptations are discussed in Section 3.

Multi-Terminal Binary Decision Diagrams (MTBDDs) (Minato 1996; Fujita, McGeer, and Yang 1997; Klarlund and Møller 2001), also called Algebraic Decision Diagrams (ADDs) (Bahar et al. 1993; Somenzi 2015), are graph-based symbolic representations of functions $f : \mathbb{B}^{\mathcal{P}} \to \mathcal{S}$, where $\mathcal{P}$ is a set of Boolean propositions and $\mathcal{S}$ is a set of labels. We use MTBDDs to symbolically represent the next-state transition functions of deterministic finite automata (DFA) arising from $\mathsf{LTL_f}$ formulas. MTBDD-based representations of DFA, also called MTDFA, have been used by Zhu et al. (2017) and Duret-Lutz et al. (2025) for $\mathsf{LTL_f}$ synthesis under full observability. More details on MTDFAs are given in Section 4.

3 On-the-fly Synthesis Framework

This section presents an on-the-fly approach to $\mathsf{LTL_f}$ synthesis under partial observability. We first briefly recall formula progression as a standard technique for constructing DFAs for $\mathsf{LTL_f}$ specifications on the fly. We then extend this technique to partial observability by introducing *observable progression*, which incrementally constructs a belief-state DFA, performing universal quantification over unobservable variables at each step. Finally, we show how this construction enables on-the-fly game solving, by interleaving belief-state DFA construction with game exploration.

3.1 $\mathsf{LTL_f}$-to-DFA via Formula Progression

On-the-fly DFA construction for an $\mathsf{LTL_f}$ formula is based on the idea of formula progression (Xiao et al. 2021; De Giacomo et al. 2022; Xiao et al. 2025). Our formula progression procedure, denoted by $\mathsf{fp}(\cdot,\cdot)$, is adapted from existing progression-based translations to better align with state-of-the-art MTDFA-based implementation of formula progression (Duret-Lutz et al. 2025). Specifically, our adaptation requires $\mathsf{fp}(\cdot,\cdot)$ to return a (formula, flag) pair, rather than a formula alone. This is essential for introducing *observable progression* in the next section, which leverages the MTDFA representation to construct belief-state DFAs on the fly.

The procedure $\mathsf{fp}(\varphi, w)$ takes an $\mathsf{LTL_f}$ formula φ and an assignment $w \in \mathbb{B}^{\mathcal{P}}$, and returns a pair $(\varphi', b) \in \mathsf{LTL_f}(\mathcal{P}) \times \mathbb{B}$, where φ' is an $\mathsf{LTL_f}$ formula representing the remaining to be satisfied after reading w, and b is a Boolean value indicating whether φ is satisfied after reading w, assuming the trace terminates at that point. Note that we use the following lifting of Boolean connectives to pairs: for $\odot \in \{\wedge, \vee, \to, \leftrightarrow, \oplus\}$, we have $(\varphi_1, b_1) \odot (\varphi_2, b_2) = ([\varphi_1 \odot \varphi_2]_\sim, b_1 \odot b_2)$, and $\neg(\varphi, b) = ([\neg\varphi]_\sim, \neg b)$.

$$\mathsf{fp}(tt, w) = (tt, \top) \quad \text{and} \quad \mathsf{fp}(f\!f, w) = (f\!f, \bot);$$

$$\mathsf{fp}(p, w) = \begin{cases} (tt, \top) & \text{if } p \in w, \\ (f\!f, \bot) & \text{if } p \notin w; \end{cases}$$

$$\mathsf{fp}(\neg\varphi, w) = \neg\mathsf{fp}(\varphi, w);$$

$$\mathsf{fp}(\varphi_1 \odot \varphi_2, w) = \mathsf{fp}(\varphi_1, w) \odot \mathsf{fp}(\varphi_2, w);$$

$$\mathsf{fp}(\mathsf{X}\varphi, w) = (\varphi, \top) \quad \text{and} \quad \mathsf{fp}(\mathsf{X}^!\varphi, w) = (\varphi, \bot);$$

$$\mathsf{fp}(\mathsf{F}\varphi, w) = \mathsf{fp}(\varphi, w) \vee (\mathsf{F}\varphi, \bot);$$

$$\mathsf{fp}(\mathsf{G}\varphi, w) = \mathsf{fp}(\varphi, w) \wedge (\mathsf{G}\varphi, \top);$$

$$\mathsf{fp}(\varphi_1 \,\mathsf{U}\, \varphi_2, w) = \mathsf{fp}(\varphi_2, w) \vee \big(\mathsf{fp}(\varphi_1, w) \wedge (\varphi_1 \,\mathsf{U}\, \varphi_2, \bot)\big);$$

$$\mathsf{fp}(\varphi_1 \,\mathsf{R}\, \varphi_2, w) = \mathsf{fp}(\varphi_2, w) \wedge \big(\mathsf{fp}(\varphi_1, w) \vee (\varphi_1 \,\mathsf{R}\, \varphi_2, \top)\big).$$

In order to build a DFA of an $\mathsf{LTL_f}$ formula $\varphi \in \mathsf{LTL_f}(\mathcal{P})$ via formula progression, we consider $[\varphi]_\sim$ as the initial state of the DFA. Then, for each assignment $w \in \mathbb{B}^{\mathcal{P}}$, we compute $\mathsf{fp}(\varphi, w)$ and obtain a progressed formula, which is treated as the successor state of φ with respect to the input w. By repeating this construction for newly generated formulas, the DFA is built on the fly, generating only the states that are reachable from φ, denoted by $\mathsf{Reach}(\varphi)$, where $\mathsf{Reach}(\varphi) = \{[\varphi]_\sim\} \cup \{[\varphi']_\sim \mid (\varphi', b) = \mathsf{fp}(\varphi, \sigma)$ for every $\sigma \in (\mathbb{B}^{\mathcal{P}})^+\}$. Note that the function $\mathsf{fp}(\cdot,\cdot)$ is naturally generalized to finite non-empty words $\sigma \in (\mathbb{B}^{\mathcal{P}})^+$: $\mathsf{fp}(\varphi, \sigma) = (\varphi_{|\sigma|-1}, b_{|\sigma|-1})$ with $(\varphi_i, b_i) = \mathsf{fp}(\varphi_{i-1}, \sigma(i))$ for $0 \leq i < |\sigma|$, and φ_{-1} refers to the original formula φ. Due to propositional equivalence, this con-

struction produces a doubly exponential DFA, i.e. $2^{2^{O(|\mathsf{sf}(\varphi)|)}}$ states, in the worst case.

3.2 Belief-State DFA Construction

Note that in $\mathsf{LTL_f}$ synthesis under partial observability, a subset of variables is unobservable. Therefore, a straightforward way to handle the unobservable variables $\mathcal{U}$ is projecting them away from the transitions of the DFA over alphabet $\mathcal{I} \uplus \mathcal{O} \uplus \mathcal{U}$. This projection introduces an NFA over alphabet $\mathcal{I} \uplus \mathcal{O}$ that must be determinized via subset construction to obtain a belief-state DFA (De Giacomo and Vardi 2016). In this section, we show how this belief-state DFA construction can be performed on the fly by interleaving formula progression with subset construction.

In the remainder of the section, for simplicity, we only distinguish observable and unobservable variables. Thus, we consider $\mathcal{P} = \mathcal{P}_o \uplus \mathcal{P}_u$, where $\mathcal{P}_o = \mathcal{I} \uplus \mathcal{O}$ is the set of observable (environment and system) variables and $\mathcal{P}_u = \mathcal{U}$ is the set of unobservable (environment) variables.

Intuitively, formula progression decomposes an $\mathsf{LTL_f}$ formula φ into a requirement about the *current* assignment, which can be checked immediately, and a requirement about the *future* that must hold on the yet unavailable suffix of the trace. Under partial observability, however, the current assignment is only partially known, due to the unobservable variables $\mathcal{P}_u$. As a result, progressing an $\mathsf{LTL_f}$ formula with respect to an observable assignment $w_o \in \mathbb{B}^{\mathcal{P}_o}$ no longer leads to a single future requirement, but to a set of possible future requirements corresponding to different assignments of the unobservable variables.

Accordingly, given an $\mathsf{LTL_f}$ formula $\varphi \in \mathsf{LTL_f}(\mathcal{P})$, a belief state is a finite set $s = \{\varphi_1, \ldots, \varphi_k\} \subseteq \mathsf{Reach}(\varphi)$. In particular, we represent such a belief-state by the formula $\psi_s = \bigwedge_{\varphi_i \in s} \varphi_i$, which naturally captures the universal quantification over unobservable variables induced by subset construction. It is worth noting that the implementation of belief states by Tabajara and Vardi (2020) did introduce an additional exponential blowup beyond standard $\mathsf{LTL_f}$-to-DFA construction, resulting in a triply exponential worst-case complexity. However, this additional cost is not inherent to our belief-state approach. Although belief states are defined as finite sets of formulas, they can be represented as conjunctions of formulas, and identified through propositional equivalence, rather than treated as distinct syntactic sets. We return to this point later and formally show that the resulting belief-state space remains doubly exponential in the size of the original formula. For simplicity, we identify belief states s with their conjunctive formula representations ψ_s, or ψ when it's clear from the context.

Observable Progression. Let ψ_s be a belief state, and let $w_o \in \mathbb{B}^{\mathcal{P}_o}$ be an observable assignment. Intuitively, observable progression takes the current belief state, progresses it with respect to the observable assignment w_o, and produces the corresponding successor belief state. To this end, observable progression considers all full assignments $w \in \mathbb{B}^{\mathcal{P}}$ that agree with w_o on the observable variables, i.e., $w_o = w|_{\mathcal{P}_o}$. Formula progression is applied to ψ_s under each such assignment. Since the successor belief state must account for all such possible assignments, we combine the resulting progressions conjunctively. Note that formula progression returns a pair in $\mathsf{LTL_f}(\mathcal{P}) \times \mathbb{B}$, so to define observable progression, we lift conjunction to such pairs as follows: for a non-empty finite set $X \subseteq \mathsf{LTL_f}(\mathcal{P}) \times \mathbb{B}$, we define $\bigwedge X = \Big(\bigwedge_{(\varphi,b)\in X} \varphi,\ \bigwedge_{(\varphi,b)\in X} b\Big)$.

Definition 5. *The observable progression of ψ_s under w_o is defined as*

$$\mathsf{fpObs}(\psi_s, w_o) = \bigwedge \big\{ \mathsf{fp}(\psi_s, w) \mid w \in \mathbb{B}^{\mathcal{P}},\ w|_{\mathcal{P}_o} = w_o \big\}.$$

The following two lemmas show that observable progression is a suitable generalization of formula progression to the setting of partial observability. In particular, observable progression preserves the semantics of $\mathsf{LTL_f}$ under unobservable variables as well as the propositional behavior of $\mathsf{LTL_f}$ formulas.

Lemma 1. *Let $\mathcal{P} = \mathcal{P}_o \uplus \mathcal{P}_u$, let ψ_s be a belief state, and ψ'_s the observably progressed formula over $w_o \in \mathbb{B}^{\mathcal{P}_o}$, i.e., $(\psi'_s, _) = \mathsf{fpObs}(\psi_s, w_o)$. Let $\sigma \in (\mathbb{B}^{\mathcal{P}})^+$ be a full trace and let $i < |\sigma| - 1$ be a position such that $\sigma(i)|_{\mathcal{P}_o} = w_o$. Then*

$$\sigma, i+1 \models \psi'_s \quad \textit{iff} \quad \forall w_u \in \mathbb{B}^{\mathcal{P}_u} : \sigma^{w_u}, i \models \psi_s,$$

where σ^{w_u} is obtained from σ by changing only position i: $\sigma^{w_u}(i) = w_o \sqcup w_u$ and $\sigma^{w_u}(j) = \sigma(j)$ for all $j \neq i$.

Proof. By definition of observable progression,

$$\begin{aligned} \mathsf{fpObs}(\psi_s, w_o) &= \bigwedge \big\{ \mathsf{fp}(\psi_s, w) \ \big|\ w \in \mathbb{B}^{\mathcal{P}},\ w|_{\mathcal{P}_o} = w_o \big\} \\ &= \bigwedge \big\{ \mathsf{fp}(\psi_s,\, w_o \sqcup w_u) \ \big|\ w_u \in \mathbb{B}^{\mathcal{P}_u} \big\}. \end{aligned}$$

For each $w_u \in \mathbb{B}^{\mathcal{P}_u}$, let (ψ_{w_u}, b_{w_u}) be the result of progressing ψ_s under the assignment $w_o \sqcup w_u$, that is, $(\psi_{w_u}, b_{w_u}) = \mathsf{fp}(\psi_s,\, w_o \sqcup w_u)$. By construction, the formula ψ' returned by $\mathsf{fpObs}(\psi_s, w_o)$ is the conjunction of all formulas ψ_{w_u} over $w_u \in \mathbb{B}^{\mathcal{P}_u}$. Therefore, we need to show

$$\sigma, i+1 \models \psi'_s \quad \text{iff} \quad \forall w_u \in \mathbb{B}^{\mathcal{P}_u} : \sigma, i+1 \models \psi_{w_u}. \tag{1}$$

Now let $w_u \in \mathbb{B}^{\mathcal{P}_u}$ be arbitrary. By definition of the trace σ^{w_u}, its value at position i is $w_o \sqcup w_u$. By the correctness of formula progression, we have

$$\sigma^{w_u}, i \models \psi_s \quad \text{iff} \quad \sigma^{w_u}, i+1 \models \psi_{w_u}. \tag{2}$$

Moreover, the traces σ^{w_u} and σ share the same suffix starting at $i+1$. Hence, the satisfaction of ψ_{w_u} at position $i+1$ is the same on both traces. That is,

$$\sigma^{w_u}, i+1 \models \psi_{w_u} \quad \text{iff} \quad \sigma, i+1 \models \psi_{w_u}. \tag{3}$$

Combining (2) & (3), we obtain that for every $w_u \in \mathbb{B}^{\mathcal{P}_u}$,

$$\sigma^{w_u}, i \models \psi_s \quad \text{iff} \quad \sigma, i+1 \models \psi_{w_u}.$$

Taking universal quantification over all $w_u \in \mathbb{B}^{\mathcal{P}_u}$ and using (1), we conclude that

$$\sigma, i+1 \models \psi'_s \quad \text{iff} \quad \forall w_u \in \mathbb{B}^{\mathcal{P}_u} : \sigma^{w_u}, i \models \psi_s. \quad \square$$

Lemma 2. *Let φ and ψ be two $\mathsf{LTL_f}$ formulas over $\mathcal{P}$ such that $\varphi \sim \psi$, and let φ' and ψ' be their respective observably progressed formulas over $w_o \in \mathbb{B}^{\mathcal{P}_o}$, i.e., $(\varphi', _) = \mathsf{fpObs}(\varphi, w_o)$ and $(\psi', _) = \mathsf{fpObs}(\psi, w_o)$. Then*

$$\varphi' \sim \psi'.$$

Proof. By definition of observable progression,

$$\mathsf{fpObs}(\varphi, w_o) = \bigwedge \{ \mathsf{fp}(\varphi, w) \mid w \in \mathbb{B}^{\mathcal{P}},\ w|_{\mathcal{P}_o} = w_o \},$$

and

$$\mathsf{fpObs}(\psi, w_o) = \bigwedge \{ \mathsf{fp}(\psi, w) \mid w \in \mathbb{B}^{\mathcal{P}},\ w|_{\mathcal{P}_o} = w_o \}.$$

Since $\varphi \sim \psi$ and formula progression preserves propositional equivalence, for every $w \in \mathbb{B}^{\mathcal{P}}$ we have $\mathsf{fp}(\varphi, w) \sim \mathsf{fp}(\psi, w)$. For each $w \in \mathbb{B}^{\mathcal{P}}$, write $(\varphi_w, _) = \mathsf{fp}(\varphi, w)$ and $(\psi_w, _) = \mathsf{fp}(\psi, w)$, and we have that $\varphi_w \sim \psi_w$. Therefore, since $\varphi_w \sim \psi_w$ for every $w \in \mathbb{B}^{\mathcal{P}}$ with $w|_{\mathcal{P}_o} = w_o$, and propositional equivalence is preserved under conjunction, it follows that

$$\bigwedge_{w|_{\mathcal{P}_o} = w_o} \varphi_w \sim \bigwedge_{w|_{\mathcal{P}_o} = w_o} \psi_w.$$

Hence, $\varphi' \sim \psi'$, concluding the proof. □

Let $\varphi \in \mathsf{LTL_f}(\mathcal{P})$ be an $\mathsf{LTL_f}$ formula such that $\mathcal{P} = \mathcal{P}_o \uplus \mathcal{P}_u$. We construct a belief-state DFA for φ by using observable progression to obtain the state space and transition function. Intuitively, each state of the automaton represents a belief state reachable from the initial specification φ with some sequence of observable assignments $\sigma_o \in (\mathbb{B}^{\mathcal{P}_o})^+$. Progressing on an observable assignment $w_o \in \mathbb{B}^{\mathcal{P}_o}$ transits from the current belief state to its successor by applying observable progression, which accounts for all full assignments $w \in \mathbb{B}^{\mathcal{P}}$ consistent with w_o. Acceptance is checked by the Boolean value returned by observable progression. We generalize $\mathsf{LTL_f}$ observable progression from single instance to finite traces by defining $\mathsf{fpObs}(\varphi, \sigma_o \cdot w_o) = \mathsf{fpObs}(\mathsf{fpObs}(\varphi, \sigma_o), w_o)$, where $w_o \in \mathbb{B}^{\mathcal{P}_o}$ and $\sigma_o \in (\mathbb{B}^{\mathcal{P}_o})^+$ and formalize the belief-state DFA construction as follows.

Definition 6 ($\mathsf{LTL_f}$ to Belief-State DFA)**.** *Given an $\mathsf{LTL_f}$ formula $\varphi \in \mathsf{LTL_f}(\mathcal{P})$, where $\mathcal{P} = \mathcal{P}_o \uplus \mathcal{P}_u$, let $\mathcal{A}^B_\varphi = \langle \mathcal{S}, \mathbb{B}^{\mathcal{P}_o}, \iota^B, \delta^B, \mathcal{T}^B \rangle$ be a belief-state DFA by setting $\mathcal{S} = \mathsf{Reach}^B(\varphi)$, where $\mathsf{Reach}^B(\varphi) = \{[\varphi]_\sim\} \cup \{[\psi']_\sim \mid (\psi', _) = \mathsf{fpObs}(\varphi, \sigma_o), \sigma_o \in (\mathbb{B}^{\mathcal{P}_o})^+\}$, $\iota^B = [\varphi]_\sim$, $\delta^B : \mathcal{S} \times \mathbb{B}^{\mathcal{P}_o} \to \mathcal{S}$ is such that $\delta^B(s, w_o) = s'$, where $(s', _) = \mathsf{fpObs}(s, w_o)$ for $s \in \mathcal{S}$ and $\mathcal{T}^B = \{(s, w_o) \in \mathcal{S} \times \mathbb{B}^{\mathcal{P}_o} \mid (_, b) = \mathsf{fpObs}(s, w_o)$ and $b = \top\}$.*

The following theorem shows the correctness of the construction in Definition 6.

Theorem 1. *Let $\varphi \in \mathsf{LTL_f}(\mathcal{P})$ with $\mathcal{P} = \mathcal{P}_o \uplus \mathcal{P}_u$, and let $\mathcal{A}^B_\varphi$ be the belief-state DFA constructed in Definition 6. Then for every observable word $\sigma_o \in (\mathbb{B}^{\mathcal{P}_o})^+$,*

$$\sigma_o \in \mathscr{L}(\mathcal{A}^B_\varphi) \quad \textit{iff}$$
$$\forall \sigma \in (\mathbb{B}^{\mathcal{P}})^+ \textit{ such that } \sigma|_{\mathcal{P}_o} = \sigma_o \textit{ we have that } \sigma \models \varphi.$$

Proof. Let $\sigma_o = w_o^0 \cdots w_o^{n-1} \in (\mathbb{B}^{\mathcal{P}_o})^+$. Consider the (unique) run of $\mathcal{A}^B_\varphi$ on σ_o:

$$s_0 \xrightarrow{w_o^0} s_1 \xrightarrow{w_o^1} \cdots \xrightarrow{w_o^{n-1}} s_n,$$

where $s_0 = [\varphi]_\sim$ and $\delta^B(s_t, w_o^t) = s_{t+1}$ for $0 \le t \le n-1$. By Definition 6, this means that

$$(s_{t+1}, _) = \mathsf{fpObs}(s_t, w_o^t). \tag{1}$$

For $0 \le t < n$, let $\tau \in (\mathbb{B}^{\mathcal{P}})^{n-t}$ be such that $\tau|_{\mathcal{P}_o} = w_o^t \cdots w_o^{n-1}$. We need to show that the following holds:

$$\tau \models s_t \iff$$
$$\forall (w_u^0 \cdots w_u^{t-1}) \in (\mathbb{B}^{\mathcal{P}_u})^t . (\sigma_o(..t-1) \sqcup (w_u^0 \cdots w_u^{t-1})) \cdot \tau \models \varphi.$$

For $t = 0$, the equivalence holds since $s_0 = [\varphi]_\sim$.

Assume the equivalence holds for some $t < n$. Let $w_o = w_o^t = \tau(0)|_{\mathcal{P}_o}$. By (1), we have $(s_{t+1}, _) = \mathsf{fpObs}(s_t, w_o)$. Applying Lemma 1, we have that

$$\tau(1..) \models s_{t+1} \quad \text{iff} \quad \forall w_u \in \mathbb{B}^{\mathcal{P}_u} : \tau^{w_u} \models s_t,$$

where $\tau^{w_u}(1..) = \tau(1..)$ and $\tau^{w_u}(0) = w_o \sqcup w_u$.

By induction hypothesis, we have that $\tau(1..) \models s_{t+1}$ iff $\forall (w_u^0 \cdots w_u^{t-1}) \in (\mathbb{B}^{\mathcal{P}_u})^t$ and $\forall w_u \in (\mathbb{B}^{\mathcal{P}_u})$

$$(\sigma_o(..t-1) \sqcup (w_u^0 \cdots w_u^{t-1})) \cdot (w_o \sqcup w_u) \cdot \tau(1..) \models \varphi$$

which is equivalent to $\forall (w_u^0 \cdots w_u^t) \in (\mathbb{B}^{\mathcal{P}_u})^{t+1}$

$$(\sigma_o(..t) \sqcup (w_u^0 \cdots w_u^t)) \cdot \tau(1..) \models \varphi.$$

Thus, the equivalence holds for $t + 1$.

Finally, by Definition 6, the run on σ_o is accepting iff the last transition (s_{n-1}, w_o^{n-1}) is accepting, i.e., iff for $(_, b) = \mathsf{fpObs}(s_{n-1}, w_o^{n-1})$ we have $b = \top$. Unfolding the definition of $\mathsf{fpObs}(\cdot)$ and the Boolean value of $\mathsf{fp}(\cdot)$, this holds exactly when φ is satisfied for all full traces consistent with σ_o. Therefore, $\sigma_o \in \mathscr{L}(\mathcal{A}^B_\varphi)$ iff every full trace $\sigma \in (\mathbb{B}^{\mathcal{P}})^+$ with $\sigma|_{\mathcal{P}_o} = \sigma_o$ satisfies φ. □

We now consider the size of the belief-state space induced by observable progression. Although belief states are defined as sets of formulas obtained from formula progression, they are always identified by propositional equivalence through their conjunctive representation. Intuitively, while belief states may be considered as sets, they are represented as single Boolean formulas built from subformulas of φ. Indeed, there are at most doubly exponentially many propositionally non-equivalent Boolean formulas of this form, thereby avoiding a triply exponential worst-case blowup of Tabajara and Vardi (2020)'s construction. Thanks to propositional equivalence, this new belief-state construction maintains a doubly exponential worst-case bound, as formalized by the following theorem.

Theorem 2. *Let $\varphi \in \mathsf{LTL_f}(\mathcal{P})$ and let $\mathcal{A}^B_\varphi$ be the belief-state DFA constructed in Definition 6. Then the state set $\mathcal{S}$ of $\mathcal{A}^B_\varphi$ is finite and satisfies $|\mathcal{S}| = |\mathsf{Reach}^B(\varphi)| = O(2^{2^n})$, where $n = |\mathsf{sf}(\varphi)|$.*

Proof. By construction, every belief state is represented by a formula obtained as a conjunction of formulas produced by formula progression starting from φ. Formula progression introduces only Boolean combinations of subformulas of φ. Since $\mathsf{sf}(\varphi)$ is finite with $|\mathsf{sf}(\varphi)| = n$, there are at most 2^{2^n} propositional equivalence classes of such Boolean formulas. Therefore, the set of reachable belief states, considering propositional equivalence (Lemma 2), is finite and bounded by $O(2^{2^n})$. □

A straightforward way of using our belief-state DFA construction technique would be solving a reachability game on a fully built belief-state DFA, as done by Tabajara and Vardi (2020). However, this cannot avoid the worst-case doubly exponential blowup. In the next section, we show how to integrate on-the-fly game solving to the belief-state DFA construction, enabling a unified on-the-fly approach to $\mathsf{LTL_f}$ synthesis under partial observability.

3.3 On-the-fly Synthesis

For $\mathsf{LTL_f}$ synthesis under full observability, on-the-fly synthesis techniques have proven to be highly effective (Xiao et al. 2021; De Giacomo et al. 2022; Xiao et al. 2024; Favorito 2023; Li et al. 2025; Duret-Lutz et al. 2025). In this setting, the DFA for the $\mathsf{LTL_f}$ specification is constructed incrementally via formula progression and explored only as needed during game solving, avoiding the upfront construction of the full automaton.

Our belief-state construction enables the same on-the-fly technique to be applied to $\mathsf{LTL_f}$ synthesis under partial observability. Since belief states and transitions are generated on demand via observable progression, the belief-state DFA does not need to be built upfront. Instead, belief-state construction and game solving can be tightly interleaved, exploiting only those belief states that are actually visited during the reachability game.

4 Implementation

The Spot library (Duret-Lutz et al. 2022) contains a tool for $\mathsf{LTL_f}$-synthesis called `ltlfsynt`[2] (Duret-Lutz et al. 2025). Our implementation is an extension of `ltlfsynt` that adds the option `--unobservable-ins`, which is publicly available since version 2.15 of Spot. We first recall how that tool does synthesis with *full observability* before discussing our changes.

In `ltlfsynt`, the $\mathsf{LTL_f}$ specification is converted into a DFA with transition-based acceptance, represented using MTBDDs: they call that representation an MTDFA. The states $\mathcal{Q} \subseteq \mathsf{LTL_f}(\mathcal{P})$ of an MTDFA are identified with $\mathsf{LTL_f}$ formulas denoting the language recognized from these states. The set of outgoing transitions of a state are represented by an MTBDD with internal nodes labeled by $\mathcal{P}$ and terminals labeled by $\mathsf{LTL_f}(\mathcal{P}) \times \mathbb{B}$. In a terminal (α, b), the destination state is α, while b indicates whether the automaton can stop upon reaching this terminal (in other words, b is the acceptance of the transition leading to α).

[2]https://spot.lre.epita.fr/ltlfsynt.html

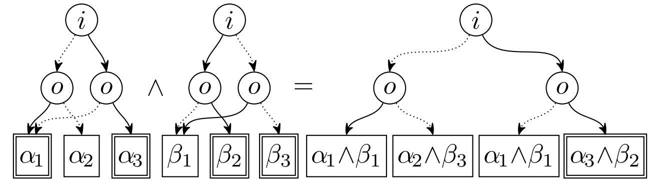

Figure 1: Conjunction of two MTBDDs with $\mathsf{LTL_f}$ terminals. As typical in BDD-based representations, a node (x) with dotted edge to ℓ and solid edge to h should be read as "if x then h else ℓ".

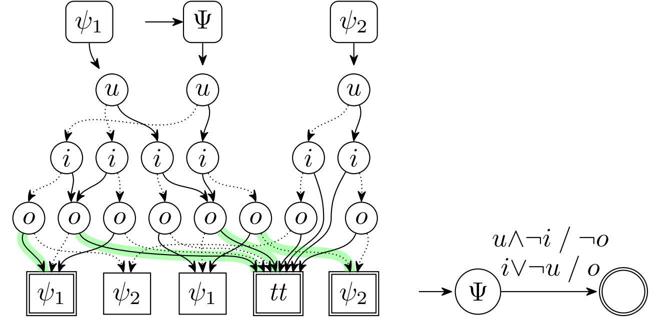

Figure 2: (left) An MTDFA for the formula $\Psi = \psi_1 \wedge \psi_2$ where $\psi_1 = (\mathsf{G}\mathsf{F}u) \rightarrow \mathsf{F}(i \leftrightarrow o)$ and $\psi_2 = (\mathsf{G}\mathsf{F}\neg u) \rightarrow \mathsf{F}(i \vee o)$. A root $(\alpha) \rightarrow m$ indicates that $\mathsf{tr}(\alpha) = m$. If this automaton is interpreted as a game where the environment plays $\mathcal{I} = \{u, i\}$, and the controller plays $\mathcal{O} = \{o\}$, then the controller has a strategy to reach an accepting terminal if it takes the highlighted edges. (right) The corresponding controller as a terminating Mealy machine.

Concretely, outgoing transitions are computed using the function $\mathsf{tr} : \mathsf{LTL_f}(\mathcal{P}) \rightarrow \mathsf{MTBDD}(\mathcal{P}, \mathsf{LTL_f}(\mathcal{P}) \times \mathbb{B})$ defined inductively as follows, where we represent a terminal labeled by $(\alpha, \bot)$ (resp. $(\alpha, \top)$) as $\boxed{\alpha}$ (resp. $\boxed{\boxed{\alpha}}$).

$$\mathsf{tr}(\mathit{ff}) = \boxed{\mathit{ff}} \qquad \mathsf{tr}(\mathsf{X}^{!}\alpha) = \boxed{\alpha}$$

$$\mathsf{tr}(\mathit{tt}) = \boxed{\boxed{\mathit{tt}}} \qquad \mathsf{tr}(\mathsf{X}\alpha) = \boxed{\boxed{\alpha}}$$

$$\mathsf{tr}(p) = (p) \text{ with dotted edge to } \boxed{\mathit{ff}} \text{ and solid edge to } \boxed{\boxed{\mathit{tt}}} \quad \text{for } p \in \mathcal{P} \qquad \mathsf{tr}(\neg\alpha) = \neg\mathsf{tr}(\alpha)$$

$$\mathsf{tr}(\alpha \odot \beta) = \mathsf{tr}(\alpha) \odot \mathsf{tr}(\beta) \text{ for any } \odot \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$$

$$\mathsf{tr}(\alpha \mathbin{\mathsf{U}} \beta) = \mathsf{tr}(\beta) \vee (\mathsf{tr}(\alpha) \wedge \boxed{\alpha \mathbin{\mathsf{U}} \beta}) \qquad \mathsf{tr}(\mathsf{F}\alpha) = \mathsf{tr}(\alpha) \vee \boxed{\mathsf{F}\alpha}$$

$$\mathsf{tr}(\alpha \mathbin{\mathsf{R}} \beta) = \mathsf{tr}(\beta) \wedge (\mathsf{tr}(\alpha) \vee \boxed{\boxed{\alpha \mathbin{\mathsf{R}} \beta}}) \qquad \mathsf{tr}(\mathsf{G}\alpha) = \mathsf{tr}(\alpha) \wedge \boxed{\boxed{\mathsf{G}\alpha}}$$

Boolean operators that appear to the right of the equal sign are applied to MTBDDs using algorithms similar to those of BDDs, with the difference that $\mathsf{LTL_f}$-labeled terminals are combined to form new terminals: $(\alpha, a) \odot (\beta, b) = ([\alpha \odot \beta]_\sim, a \odot b)$, as done with formula progressions (Section 3.1). Figure 1 shows an example of a conjunction of two MTBDDs, that we shall reuse later.

The different paths in the MTBDD computed by $\mathsf{tr}(\varphi)$ represent all possible formula progressions for φ, but the above efficiently constructs all progressions at once, without having to compute $\mathsf{fp}(\varphi, w)$ for each $w \in \mathbb{B}^{\mathcal{P}}$ independently. By computing such an MTBDD for all $\mathsf{LTL_f}$ terminals reach-

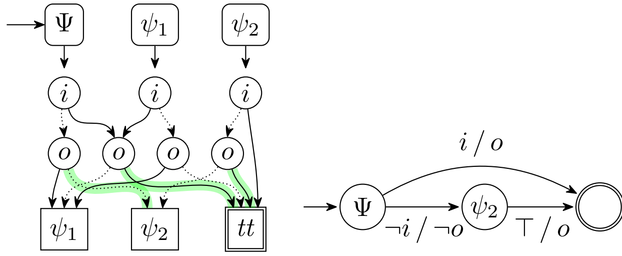

Figure 3: (left) Belief-state MTDFA for Ψ, assuming that $\mathcal{U} = \{u\}$ is unobservable. If this automaton is interpreted as a game where the controller plays $\mathcal{O} = \{o\}$, the highlighted edges represent one possible winning strategy. (right) The corresponding controller as a terminating Mealy machine.

able from φ, one can obtain an MTDFA representation as shown in Figure 2 for the formula Ψ of Example 1.

Assuming an appropriate ordering of the MTBDD variables $\mathcal{P}$ (e.g., input variables before output variables for Mealy semantics), the constructed MTDFA can be interpreted directly as a two-player game to solve the corresponding $\mathsf{LTL_f}$ synthesis problem under full observability ($\mathcal{I} = \{i, u\}$ and $\mathcal{O} = \{o\}$). In this construction, accepting terminals (such as $\boxed{tt}$ in Fig. 2) are the targets of the reachability game, so their successors do not need to be computed. Finally, the fact that $\mathsf{tr}(\cdot)$ computes all successors of one state enables an on-the-fly construction of the MTDFA while solving the game (Duret-Lutz et al. 2025, Algorithm 2). For instance in Figure 2, the highlighted winning strategy can already be found after computing $\mathsf{tr}(\Psi)$, so the algorithm can stop without exploring the discovered successors.

MTDFA-based Observable Progression. The partial-observability extension introduces support for partial observability in the above architecture.

To do so, we use an MTBDD operation to universally quantify the unobservable variables $\mathcal{U} \subseteq \mathcal{P}$ in the MTBDD computed by $\mathsf{tr}(\varphi)$ during on-the-fly exploration of the MTDFA. If $\mathsf{tr}(\varphi)$ was a symbolic representation of the set of progressions of the form $\mathsf{fp}(\varphi, w)$ for $w \in \mathbb{B}^{\mathcal{P}}$, then $\forall\mathcal{U}.\mathsf{tr}(\varphi)$ is the symbolic representation of the set of all observable progressions $\mathsf{fpObs}(\varphi, w_o)$ for $w_o \in \mathbb{B}^{\mathcal{P}_o}$.

The MTBDD $\forall\mathcal{U}.\mathsf{tr}(\varphi)$ is computed bottom-up using an algorithm similar to those of BDDs: an internal MTBDD node labeled with a variable in $\mathcal{U}$ is replaced by the conjunction of its children. For instance, the MTBDD computed for $\mathsf{tr}(\Psi)$ in Figure 2 has its top-level node labeled by $u \in \mathcal{U}$, so it should be replaced by the conjunction of its two children. This is exactly the operation depicted in Figure 1, with $\alpha_1 = \alpha_2 = \psi_1$, $\beta_1 = \beta_3 = \psi_2$, and $\alpha_3 = \beta_2 = tt$.

Doing this quantification for every state reached during the translation gives the belief-state MTDFA of Figure 3. To make this quantification efficient, we always put all unobservable variables $\mathcal{U}$ at the bottom of the variable order: thus all nodes that should be removed appear in subtrees of the MTBDDs, and their quantification replaces those subtrees by terminals labeled by the conjunction of their leaves.

Theorem 3. *For $\varphi \in \mathsf{LTL_f}(\mathcal{P})$, with $\mathcal{P} = \mathcal{P}_o \uplus \mathcal{U}$, let $\mathcal{A}^B_\varphi = \langle \mathcal{S}, \mathcal{P}_o, \iota^B, \Delta^B \rangle$ be the belief-state MTDFA obtained by setting $\iota^B = [\varphi]_\sim$, $\Delta^B(s) = \forall\mathcal{U}.\mathsf{tr}(s)$, and letting $\mathcal{S}$ be the smallest subset of $\mathsf{LTL_f}(\mathcal{P})$ such that (i) $\iota^B \in \mathcal{S}$, and (ii) for every $s \in \mathcal{S}$ and every terminal labeled (α, b) in $\Delta^B(s)$, $\alpha \in \mathcal{S}$. Then $\mathcal{S}$ is finite and $\mathcal{A}^B_\varphi$ is a symbolic representation of the belief-state DFA defined in Definition 6.*

Proof. Finiteness follows directly from Theorem 2: $\mathsf{tr}(s)$ contains only Boolean combinations of subformulas of φ, and the quantification $\forall\mathcal{U}$ only combines leaves by conjunction. Hence all states are still Boolean combinations of subformulas of φ, up to propositional equivalence, and there are finitely many of them.

Correctness follows from the definition of MTBDD universal quantification. For every $s \in \mathcal{S}$ and $w_o \in \mathbb{B}^{\mathcal{P}_o}$, evaluating $\forall\mathcal{U}.\mathsf{tr}(s)$ under w_o yields the conjunction of the progressions obtained from all completions of w_o with valuations of the unobservable variables: $(\forall\mathcal{U}.\mathsf{tr}(s))(w_o) = \bigwedge\big\{ \mathsf{fp}(\varphi, w) \;\big|\; w \in \mathbb{B}^{\mathcal{P}},\ w|_{\mathcal{P}_o} = w_o \big\}$. By definition, this is exactly $\mathsf{fpObs}(s, w_o)$. Therefore, Δ^B induces the same successors and accepting transitions as the belief-state DFA of Definition 6. ☐

On-the-fly synthesis is enabled by solving the induced reachability game during the exploration of the belief-state MTDFA, without constructing the full automaton upfront. On the example of Figure 3, an on-the-fly construction that explores ψ_2 before ψ_1 would solve the game before exploring ψ_1. The game-solving procedure was not changed; it was only applied to the belief states.

5 Empirical Evaluation

In this section, we evaluate our on-the-fly synthesis approach by comparing our implementation against state-of-the-art tools for $\mathsf{LTL_f}$ synthesis under partial observability. As mentioned in Section 4, our implementation is available in `ltlfsynt`, starting from Spot 2.15. In the experimental plots and tables, we denote that partial-observability configuration by `ltlfsynt-po`. In addition, to evaluate the practical benefits of integrating the on-the-fly game solving (see Sec 3.3), we implemented a variant of `ltlfsynt-po`, named `ltlfsynt-po-naive`, in which on-the-fly game solving is disabled. In this configuration, a belief-state MTDFA is first constructed in full, and only afterward a reachability game is solved on the resulting automaton. All of our experiments were run on a machine with an Intel Xeon Gold 6130 CPU with 2.10GHz, 64 cores and 188.5 GB of RAM. We limited each test to 90 seconds.

5.1 Baseline Tools

The tools used as baselines in our evaluation are those introduced by Tabajara and Vardi (2020), which implement different approaches. All tools were built on the symbolic synthesis framework *Syft* for $\mathsf{LTL_f}$ under full observability (Zhu et al. 2017). The first approach is belief-state based, implemented in *Syft-bsc*. This approach is similar to belief constructions commonly used in planning (Bonet and Geffner 2000; Hoffmann and Brafman 2005), which first builds a

deterministic finite automaton (DFA) for the $\mathsf{LTL_f}$ specification and then applies a second subset construction to turn it into a belief-state DFA, with the unobservable variables omitted from the transition function. The second approach is projection based, and implemented in *Syft-proj*. It instead starts from a nondeterministic finite automaton (NFA) and applies a sequence of manipulations to obtain an equivalent belief-state DFA. The third approach is MSO based, implemented in *Syft-mso*. It translates the $\mathsf{LTL_f}$ specification into a monadic second-order logic (MSO) formula with the unobservable variables universally quantified, and then constructs a DFA directly for this MSO formula, which is equivalent to the belief-state DFA obtained by the other two approaches. Since *Syft-proj* has been shown to perform worse than *Syft-bsc* and *Syft-mso* in their experimental evaluation, we exclude it from our comparison. Refer to Tabajara and Vardi (2020) for further details on these approaches.

5.2 Benchmarks

We ran our experiments on several benchmark sets, which we divide into two categories. The first category is the *TV-Benchmarks*, introduced by Tabajara and Vardi (2020) and designed with partial-observability in mind. That is, every instance in this benchmark already contains a set of unobservable environment variables. This category includes *Coin game* (8 instances), *Traveling Target* (9 instances), and *Private Peek*, which consists of 16 sub-categories with 100 instances each (1600 instances in total).

The second category is *SYNTCOMP-fin Benchmarks*, taken from the annual reactive synthesis competition (SYNTCOMP) (Perez 2025). These benchmarks do not originally contain any unobservable variables. As such, we randomly selected half of the environment variables and made them unobservable. To get a better chance that the unobservable variables affect the overall result, we selected benchmarks from designated two-player games, in which such variables are highly likely to affect the outcome. In total, we have 150 instances: *Single Counter* (20 instances), *Double Counter* (20 instances), *Chomp Game* (22 instances), and *Nim Game* (88 instances). We ran each instance 10 times, and in each run randomly selected a fresh set of 50% environment variables considered unobservable, without repeating any previously chosen set, to reduce the impact of randomness and obtain more robust and representative performance measurements.

Correctness. The implementation of `ltlfsynt-po` and `ltlfsynt-po-naive` was verified by comparing the results (realizable/unrealizable) returned by them with those from *Syft-bsc* and *Syft-mso*. The results were consistent among all tools in all solved cases.

5.3 Evaluation Results

Due to space limitations, we only present overall comparison results here. A detailed comparison on individual benchmark families can be found in the appendix.

Figure 4 is a cactus plot for *TV-Benchmarks*, comparing the end-to-end runtime. All benchmark instances are sorted by runtime for each tool. The colors correspond to `ltlfsynt-po` (green), `ltlfsynt-po-naive` (yellow), *Syft-bsc* (blue) and *Syft-mso* (red). Recall that in this category the partial observability is native to the specifications. This figure shows a significant performance improvement for `ltlfsynt-po` over *Syft-bsc* and *Syft-mso*, with the speedup ranging from 27.65x to 126.6x in average runtime across the three different benchmark families. The speedup is measured by taking the total running time of both tools (`ltlfsynt-po` and the best-performing between *Syft-bsc* and *Syft-mso*) for each individual benchmark family and computing their ratio.

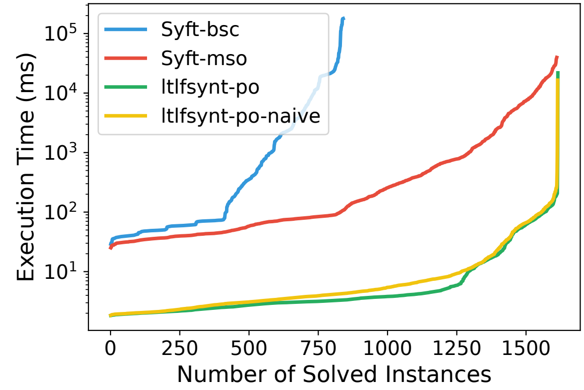

Figure 4: Cactus plot for *TV-Benchmarks*

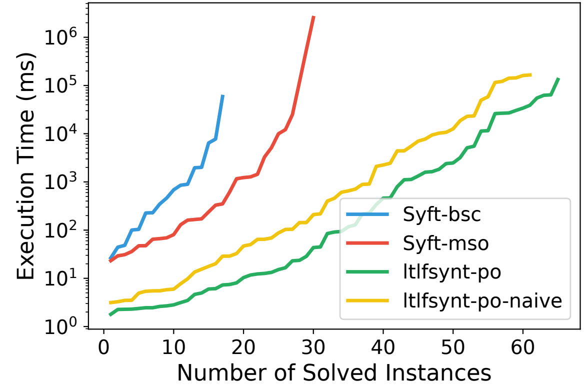

Figure 5: Cactus plot for *SYNTCOMP-fin Benchmarks*

Similarly, Figure 5 is a cactus plot for the *SYNTCOMP-fin Benchmarks*, in which the unobservable variables are randomly selected, following the same palette of colors as in Figure 4. Here too, the plot shows a significant performance improvement for `ltlfsynt-po` over *Syft-bsc* and *Syft-mso*, with the speedup ranging from 4.81x to 7,892.82x in average runtime across the different benchmark families.

Overall, `ltlfsynt-po` dramatically outperforms current state-of-the-art tools. These results demonstrate the strength of both our theoretical on-the-fly approaches as described in Section 3, and our practical MTDFA-based implementation, as described in Section 4.

We conducted a more detailed comparison to better under-

Table 1: Combined average speedups of `ltlfsynt-po` compared to `ltlfsynt-po-naive` across all benchmarks.

Source	Family	Speedup
TV-Benchmarks	Coin-Game	16.54×
	Moving-Target	1.22×
	Private-Peek	1.17×
SYNTCOMP-fin Bench.	Chomp	4.62×
	Counter	4.46×
	Counters-Double	5.11×
	Nim	4.61×

stand the contribution of the on-the-fly game solving component in `ltlfsynt-po`. In general, `ltlfsynt-po` consistently outperforms `ltlfsynt-po-naive` across all benchmarks instances.

Table 1 summarizes the speedup on both benchmark categories. Specifically, for the *TV-Benchmarks*, we can see that `ltlfsynt-po` achieves a speedup ranging from 1.17x to 16.54x over `ltlfsynt-po-naive` across the individual benchmark families. For the *SYNTCOMP-fin Benchmarks*, the speedups range from 4.46x to 5.11x. These results indicate that the major improvement in performance is due to our belief-state MTDFA construction techniques. At the same time, they also show that the integration of on-the-fly game solving plays an important role. By solving the reachability game already during the construction of the belief-state MTDFA, `ltlfsynt-po` can avoid exploring parts of the automaton that are not needed, which further improves the overall performance.

6 Conclusion and Future Work

In this paper, we presented an on-the-fly approach to $\mathsf{LTL_f}$ synthesis under partial observability based on observable progression. This approach incrementally constructs the belief-state DFA, universally quantifying unobservable environment variables during construction, enabling a tight integration of belief-state DFA construction and game solving. The resulting implementation, leveraging MTDFA representation, shows significant improvements compared to existing approaches. More generally, our results show that MTDFA-based on-the-fly techniques are not limited to the fully-observable setting. Note that our belief-state DFA construction through observable progression essentially builds an automaton incrementally that is equivalent to a formula of the form $\forall\mathcal{U}.\varphi$, with $\varphi \in \mathsf{LTL_f}(\mathcal{P})$ and $\mathcal{U} \subseteq \mathcal{P}$. This technique can be adapted to support DFA construction for $\exists\mathcal{U}.\varphi$ by changing universal quantification to existential quantification on the MTBDDs, and more generally for quantified $\mathsf{LTL_f}$ formulas ($\mathsf{QLTL_f}$). Thus our approach can be extended to settings that apply synthesis to $\mathsf{QLTL_f}$ formulas, such as in (Hagemeier, De Giacomo, and Vardi 2025).

Acknowledgments

We thank the reviewers for their insightful comments. SZ and SC are partly supported by the Royal Society International Exchanges grant IES\R2\252189. NA is supported by the Open University of Israel MSc Excellence Scholarship and by The Open University of Israel Research Fund 47372.

AI Declaration

Generative AI was used for language polishing and experimental script development. All scientific results, proofs, artifacts, and conclusions were independently verified by the authors, who assume full responsibility for the final content of the paper.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1–2):123–191.

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1993. Algebraic decision diagrams and their applications. In *ICCAD*, 188–191.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *ICAPS*, 26–33.

Bansal, S.; Li, Y.; Tabajara, L. M.; and Vardi, M. Y. 2020. Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In *AAAI*, 9766–9774.

Bansal, S.; Li, Y.; Tabajara, L. M.; Vardi, M. Y.; and Wells, A. M. 2023. Model checking strategies from synthesis over finite traces. In *ATVA*, 227–247.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *AIPS*, 52–61.

Calvanese, D.; De Giacomo, G.; and Vardi, M. Y. 2002. Reasoning about actions and planning in LTL action theories. In *KR*, 593–602.

Camacho, A.; Bienvenu, M.; and McIlraith, S. A. 2019. Towards a unified view of AI planning and reactive synthesis. In *ICAPS*, 58–67.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1–2):35–84.

De Giacomo, G., and Favorito, M. 2021. Compositional approach to translate LTL_f/LDL_f into deterministic finite automata. In *ICAPS*, 122–130.

De Giacomo, G., and Rubin, S. 2018. Automata-theoretic foundations of FOND planning for LTL_f/LDL_f goals. In *IJCAI*, 4729–4735.

De Giacomo, G., and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, 854–860.

De Giacomo, G., and Vardi, M. Y. 2015. Synthesis for LTL and LDL on finite traces. In *IJCAI*, 1558–1564.

De Giacomo, G., and Vardi, M. Y. 2016. $\mathrm{LTL_f}$ and $\mathrm{LDL_f}$ Synthesis under Partial Observability. In *IJCAI*, 1044–1050.

De Giacomo, G.; Favorito, M.; Li, J.; Vardi, M. Y.; Xiao, S.; and Zhu, S. 2022. LTL_f synthesis as AND-OR graph search: Knowledge compilation at work. In *IJCAI*, 2591–2598.

Duret-Lutz, A.; Renault, E.; Colange, M.; Renkin, F.; Aisse, A. G.; Schlehuber-Caissier, P.; Medioni, T.; Martin, A.; Dubois, J.; Gillard, C.; and Lauko, H. 2022. From Spot 2.0 to Spot 2.10: What's new? In *CAV*, 174–187.

Duret-Lutz, A.; Zhu, S.; Piterman, N.; De Giacomo, G.; and Vardi, M. Y. 2025. Engineering an LTL_f synthesis tool. In *CIAA*, 129–147.

Ehlers, R.; Lafortune, S.; Tripakis, S.; and Vardi, M. Y. 2017. Supervisory control and reactive synthesis: a comparative introduction. *Discrete Event Dynamic Systems* 27(2):209–260.

Esparza, J.; Křetínský, J.; and Sickert, S. 2018. One theorem to rule them all: A unified translation of LTL into ω-automata. In *LICS*, 384–393.

Favorito, M. 2023. Forward LTL_f synthesis: DPLL at work. In *IPS-RCRA-SPIRIT@AI*IA*.

Finkbeiner, B. 2016. Synthesis of reactive systems. In *Dependable Software Systems Engineering*, volume 45. IOS Press. 72–98.

Fujita, M.; McGeer, P. C.; and Yang, J. C. 1997. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design* 10(2/3):149–169.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence* 173(5–6):619–668.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge.

Hagemeier, C.; De Giacomo, G.; and Vardi, M. Y. 2025. LTL_f synthesis under unreliable input. In *AAAI*, 14958–14966.

Henriksen, J. G.; Jensen, J.; Jørgensen, M.; Klarlund, N.; Paige, R.; Rauhe, T.; and Sandholm, A. 1995. Mona: Monadic second-order logic in practice. In *TACAS*, 89–110.

Hoffmann, J., and Brafman, R. I. 2005. Contingent planning via heuristic forward search with implicit belief states. In *ICAPS*, 71–80.

Jacobs, S.; Perez, G. A.; and Schlehuber-Caissier, P. 2026. The temporal logic synthesis format TLSF v1.2. arXiV. (Updated in 2026 to use terminating semantics.).

Klarlund, N., and Møller, A. 2001. MONA version 1.4, user manual. Technical report, BRICS.

Kupferman, O., and Vardi, M. 1997. Synthesis with incomplete informatio. In *ICTL*, 1044–1050.

Li, Y.; Xiao, S.; Zhu, S.; Li, J.; and Pu, G. 2025. A compositional framework for on-the-fly LTL_f synthesis. In *ECAI*, 1711–1718.

Maliah, S.; Komarnitski, R.; and Shani, G. 2022. Computing contingent plan graphs using online planning. *ACM Trans. Auton. Adapt. Syst.* 16(1):1:1–1:30.

Minato, S.-i. 1996. *Representation of Multi-Valued Functions*. Boston, MA: Springer US. 39–47.

Perez, G. A. 2025. SyntComp's GitHub repository of TLSF-fin benchmarks. https://github.com/SYNTCOMP/benchmarks/tree/master/tlsf-fin.

Pnueli, A., and Rosner, R. 1989. On the synthesis of a reactive module. In *POPL*, 179–190.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Rintanen, J. 2004. Complexity of planning with partial observability. In *ICAPS*, 345–354.

Somenzi, F. 2015. CUDD: CU Decision Diagram package release 3.0.0.

Tabajara, L. M., and Vardi, M. Y. 2020. LTL_f synthesis under partial observability: From theory to practice. In *GandALF*, 1–17. Benchmarks and tools can be found at https://github.com/lucasmt/ltlf-po-benchmarks (commit `2c866cd`) and https://github.com/lucasmt/Syft/tree/double-negation-dfa (commit `7d797f0`).

Xiao, S.; Li, J.; Zhu, S.; Shi, Y.; Pu, G.; and Vardi, M. 2021. On-the-fly synthesis for LTL over finite traces. In *AAAI*, 6530–6537.

Xiao, S.; Li, Y.; Huang, X.; Xu, Y.; Li, J.; Pu, G.; Strichman, O.; and Vardi, M. Y. 2024. Model-guided synthesis for LTL over finite traces. In *VMCAI*, 186–207.

Xiao, S.; Li, Y.; Zhu, S.; Sun, J.; Li, J.; Pu, G.; and Vardi, M. Y. 2025. An on-the-fly synthesis framework for LTL over finite traces. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Zhu, S., and Favorito, M. 2025. LydiaSyft: A compositional symbolic synthesis framework for LTL_f specifications. In *TACAS*, 295–302.

Zhu, S.; Tabajara, L. M.; Li, J.; Pu, G.; and Vardi, M. Y. 2017. Symbolic LTL_f synthesis. In *IJCAI*, 1362–1369.

A Supplementary Evaluation

A.1 Full Benchmark Results

We provide detailed results of our experiments. We first discuss the *TV-Benchmarks*, in which the partial observability is originated in the specification. Then we discuss the *SYNTCOMP-fin Benchmarks*, in which the unobservable variables were randomly selected.

TV-Benchmarks Figures 6, 7, and 8 show the performance of `ltlfsynt-po`, `ltlfsynt-po-naive`, *Syft-bsc* and *Syft-mso* on the *TV-Benchmarks*.

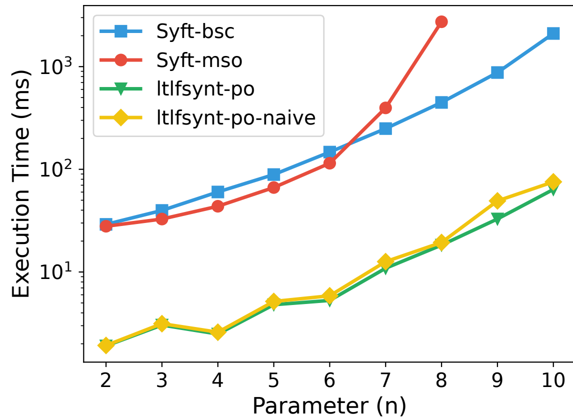

Figure 6: Moving Target Performance

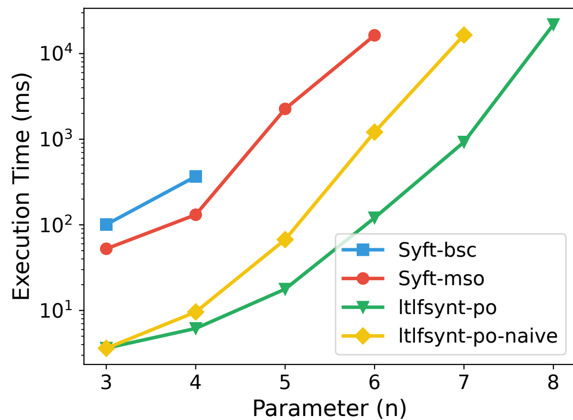

Figure 7: Coin Game Performance

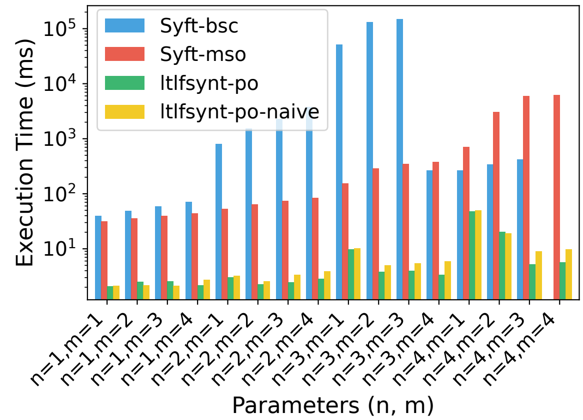

Figure 8: Private Peek Performance

Table 2 shows the average speedup in each benchmark family. The speedup is measured by taking the total running time of both tools (`ltlfsynt-po` and the best-performing one between *Syft-bsc* and *Syft-mso*) for each individual benchmark family and computing their ratio.

Benchmark Family	Speedup
Coin-Game	126.60×
Moving-Target	27.65×
Private-Peek	126.00×

Table 2: Average speedups of `ltlfsynt-po` compared to the best-performing Syft implementation (*Syft-bsc* or *Syft-mso*) across *TV-Benchmarks*.

SYNTCOMP-fin Benchmarks Similarly, Figures 9, 10, 11 and 12 show the performance of `ltlfsynt-po`, `ltlfsynt-po-naive`, *Syft-bsc* and *Syft-mso* on the *SYNTCOMP-fin Benchmarks*.

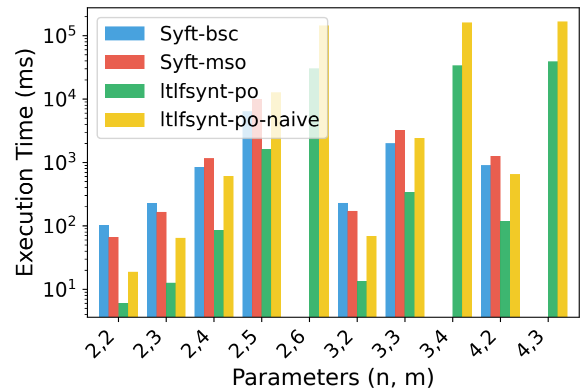

Figure 9: Chomp Performance

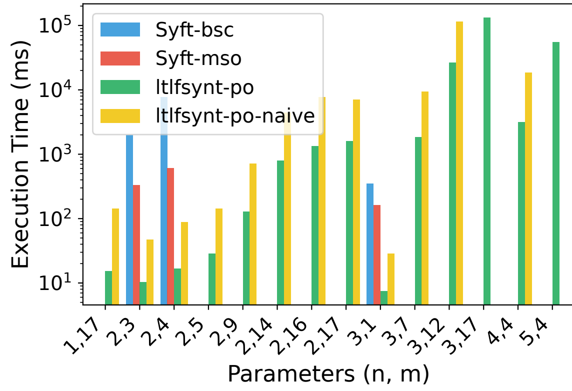

Figure 10: Nim Performance

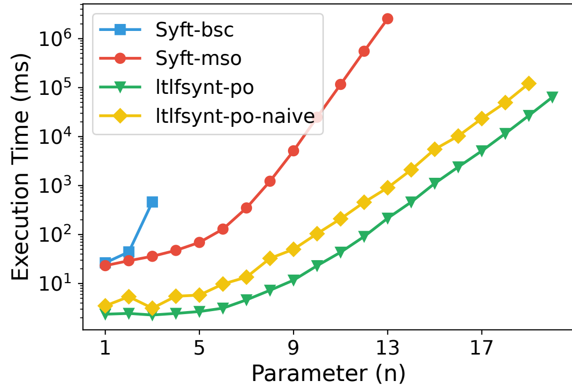

Figure 11: Counter Performance

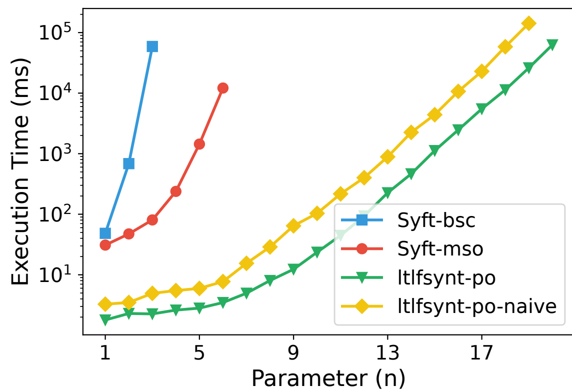

Figure 12: Double Counters Performance

Table 3 shows a speedup ranging from 4.81x to 7892.82x. It should be noted that both `ltlfsynt-po` and `ltlfsynt-po-naive` solve a lot of benchmarks that the state-of-the-art tools could not solve due to either time constraints or memory constraints.

Benchmark Family	Speedup
Chomp	4.81×
Counter	7,892.82×
Counters-Double	915.84×
Nim	32.10×

Table 3: Average speedups of `ltlfsynt-po` compared to the best-performing Syft implementation across *SYNTCOMP-fin Benchmarks* (50% variables hidden).

B Reproducibility

We describe how to reproduce the results that appear in this paper.

B.1 Benchmarks

TV-Benchmarks can be found online at https://github.com/lucasmt/ltlf-po-benchmarks

SYNTCOMP-fin Benchmarks can be found online at https://github.com/SYNTCOMP/benchmarks/tree/master/tlsf-fin

B.2 Our Tool

The implementation is publicly available in Spot starting from version 2.15 (available at https://spot.lre.epita.fr/install.html). After installing Spot, the tool can be invoked as follows:

ltlfsynt-po
```
./ltlfsynt <formula>
--ins=<inputs>
--outs=<outputs>
--unobservable-ins=<unobservables>
--semantics=<mealy or moore>
```

ltlfsynt-po-naive
```
./ltlfsynt <formula>
--ins=<inputs>
--outs=<outputs>
--unobservable-ins=<unobservables>
--semantics=<mealy or moore>
--translation=restricted
```

B.3 Baseline Tools

The version of Syft with support for partial observability can be found online at https://github.com/lucasmt/Syft/tree/double-negation-dfa. To install, follow the instructions in https://github.com/lucasmt/Syft/blob/double-negation-dfa/INSTALL. You might need to change the `CUDD_ROOT` variable in https://github.com/lucasmt/Syft/blob/double-negation-dfa/CMakeModules/Findcudd.cmake

to point to your local CUDD installation. Afterwards, add the `build/bin` folder to your path in order to have access to the `ltlf2fol` and `Syft` binaries.

Syft-bsc To run *Syft-bsc* on a benchmark `bm` composed of $\mathsf{LTL_f}$ file `bm.ltlf` and variable partition file `bm.part`, follow these steps:

1. Run `ltlf2fol` to convert the $\mathsf{LTL_f}$ file into a MONA input file `bm.mona`:
   ```
   ltlf2fol NNF bm.ltlf > bm.mona
   ```
2. Run MONA to generate a DFA file:
   ```
   mona -u -xw bm.mona > bm.dfa
   ```
3. Run Syft on the resulting DFA, with the system as the starting player (0) and partial observability (implemented via symbolic belief-states construction):
   ```
   Syft bm.dfa bm.part 0 partial dfa
   ```

Syft-mso To run *Syft-mso* on a benchmark `bm` composed of $\mathsf{LTL_f}$ file `bm.ltlf` and variable partition file `bm.part`, follow these steps:

1. Run `ltlf2fol` to convert the $\mathsf{LTL_f}$ file into a MONA input file `bm.mona`:
   ```
   ltlf2fol NNF bm.ltlf > bm.mona
   ```
2. Quantify the unobservable variables universally in the MONA file (see the script in https://github.com/lucasmt/ltlf-po-benchmarks/blob/master/quantify.py for an example), producing a new file `bm.mona.quant`.
3. Remove the unobservable variables from the variable partition file, producing a new file `bm.part.quant`.
4. Run MONA to generate a DFA file from the universally-quantified MSO formula:
   ```
   mona -u -xw bm.mona.quant >
   bm.dfa.quant
   ```
5. Run Syft on the resulting DFA, with the system as the starting player (0) and full observability (since partial observability has already been handled by universal quantification):
   ```
   Syft bm.dfa.quant bm.part.quant 0 full
   dfa
   ```

Note that for the benchmarks in https://github.com/lucasmt/ltlf-po-benchmarks, the `.mona`, `.part`, `.mona.quant` and `.part.quant` files have already been generated, and therefore only the last two steps are necessary in each case.