Inductive First-Order Formula Synthesis by ASP: A Case Study in Invariant Inference[*]

Ziyi Yang George Pîrlea Ilya Sergey

National University of Singapore, Singapore

yangziyi@u.nus.edu gpirlea@u.nus.edu ilya@nus.edu.sg

We present a framework for synthesising formulas in first-order logic (FOL) from examples, which unifies and advances state-of-the-art approaches for inference of transition system invariants. To do so, we study and categorise the existing methodologies, encoding techniques in their formula synthesis via answer set programming (ASP). Based on the derived categorisation, we propose *orthogonal slices*, a new technique for formula enumeration that partitions the search space into manageable chunks, enabling two approaches for incremental candidate pruning. Using a combination of existing techniques for first-order (FO) invariant synthesis and the orthogonal slices implemented in our framework FORCE, we significantly accelerate a state-of-the-art algorithm for distributed system invariant inference. We also show that our approach facilitates *composition* of different invariant inference frameworks, allowing for novel optimisations.

1 Introduction

First-Order Logic (FOL) has been used with great success as a foundational tool for modelling and verifying complex systems. Its applications span various domains, ranging from hardware design [5] to software verification [21]. These successes are largely attributed to the development of advanced frameworks that allow for automated verification and synthesis, often supported by high-performance provers such as Z3 [19] and cvc5 [1].

To achieve *fully automated* verification of a complex system in FOL, it is often necessary to *synthesise* formulas capturing the invariants (*i.e.*, the properties always hold) of the system being verified. Despite the undecidable nature of this task, many recent efforts have made substantial progress to infer inductive invariants of complex distributed systems (*e.g.*, Lamport's Paxos consensus protocol [16]) by synthesising FO formulas from examples: sampled traces of a protocol or counter-examples to induction. The resulting approaches are implemented by a plethora of distinct frameworks [27, 8], and a systematic study of their inter-connections is still missing. This raises important questions: are the existing synthesis methods fundamentally non-overlapping? Could techniques developed for one approach be adapted to benefit others? Addressing these questions would not only deepen our understanding of the underlying methodologies but also enable the development of superior tools for formula synthesis, potentially improving scalability of automated verification tools across various domains.

In this paper, we present a unified framework for synthesising bounded first-order formulas from examples—first-order structures that the formulas satisfy. Our framework is designed to encode and *combine* diverse synthesis techniques, enabling seamless integration with different high-level algorithms for invariant inference. To achieve this, we have conducted a detailed study of *nine* recent approaches for invariant inference of distributed systems (DS), each of which offered a different take on inductive

[*] We thank Andrew Cropper, Roland Kaminski, and Jianan Yao for providing help on Popper, Clingo, and DuoAI, respectively. We are also grateful to Roland Yap for his feedback on a draft of this paper. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001 and by Amazon Research Award "Scaling Automated Verification of Distributed Protocols with Specification Transformation and Synthesis".

Submitted to:
41st International Conference on Logic Programming (ICLP 2025)

synthesis of FO formulas. We summarise our study in two main observations *w.r.t.* inductive synthesis of FO formulas, hinting an opportunity for improvement in the state of the art:

O1 Existing techniques fall within a small number of distinct classes, in terms of how they treat their inputs and results, *e.g.*, how examples are used and how formulas are constructed. We give a uniform categorisation of these approaches.

O2 Whilst a wide variety of synthesis techniques exists, the vast majority fall into one of two categories, exploiting the properties of the first-order theories they employ: "redundancy elimination" and "incremental pruning".

To make a unified framework that captures **O1**, we use Answer Set Programming (ASP) [17], to encode the enumeration-based FO formula synthesis (as constraint solving) and the customisations and techniques of the synthesis (as knowledges representation). To improve on the existing techniques, we exploit **O2** by proposing the idea of *orthogonal slices* (also implemented by ASP) of the FO search space: a new approach to efficiently prune candidate formulas during the inductive synthesis. The key idea of orthogonal slices is to partition the search space into ordered slices, where the synthesis of former slices can be used to prune the latter slices using *either satisfied or unsatisfied* formulas. The practical benefits of the unified framework, FORCE (**F**irst-**O**rder synthesiser via o**R**thogonal sli**CE**s), are demonstrated by improving two state-of-the-art DS invariant synthesisers, DuoAI [27] and Flyvy [8], *without any conceptual modifications to their high-level algorithms*. Our results show that our framework is sufficiently expressive and extensible to encode and compose existing formula synthesis techniques, advancing the state of the art in DS invariant synthesis.

2 Overview

We start with a primer on inductive synthesis of FO formulas—a common subroutine in existing invariant inference frameworks for distributed systems (DS). Then we summarise nine notable existing approaches for DS invariant inference (with the earliest dated 2019), concluding with a brief description of our ASP-based framework to capture various aspects of the synthesis and its particular instance, orthogonal slices.

2.1 Problem Definition

First-Order Language. In this work, we focus on system properties that are expressible in a first-order, many-sorted logic with equality, following the common textbook definitions. A *signature* $\Sigma = \langle C, R, F, S \rangle$ consists of: a set of *constant symbols* C, a set of *relation symbols* (predicates) R, a set of *function symbols* F, and a set of *sorts* S for the variables, constants, and function symbols. In the rest of this paper we assume all signatures to be *finite*, *i.e.*, the sets C, R, F, and S are finite.

Logic *terms* are defined recursively. A term is either a *constant* $c \in C$, a *variable* x, or a *function symbol* $f \in F$ applied to other terms (*e.g.*, $f(x_1, x_2)$). Logic *atoms* are the basic formulas formed by applying relation symbols from R or equality to terms of appropriate sorts; and a *literal* is an atom or its negation (*i.e.*, $\neg p(x)$). *Formulas* are constructed by closing literals under *logical connectives* (*i.e.*, conjunction $\wedge$ and disjunction $\vee$) and *quantification* (universal $\forall$ and existential $\exists$). It is well-known that any FO formula can be transformed into an equivalent formula in *prenex normal form*, where all quantifiers are placed at the beginning. In this structure, the *prefix* includes the quantifiers, and the *matrix* consists of the remaining Boolean components. For example, the formula $\forall x : s_1\, \exists y : s_2.\ p(x,y) \vee (\neg q(x) \wedge r(y))$ is in prenex normal form, with prefix $\forall x : s_1\, \exists y : s_2$ and matrix $p(x,y) \vee (\neg q(x) \wedge r(y))$.

Definition 1 (First-Order Synthesis Problem) *Given a set of formulas Ω_0 over signature Σ and a set of FO structures $\sigma = \{M_1, \ldots, M_k\}$, where each M_i is a model over Σ, find a set of formulas $\Phi = \{\phi_1, \ldots, \phi_n\} \subseteq \Omega_0$ s.t. $\forall \phi \in \Phi$:*

1. $\forall M \in \sigma . M \models \phi$, *(satisfies all input FO structures)*,
2. $FreeVars(\phi) = \emptyset$, *(closed formula)*,
3. $\exists M \notin \sigma . M \not\models \phi$, *(non tautology)*,
4. $\forall \phi' \in \Phi . \phi \neq \phi' \wedge \phi \not\models \phi'$, *(no formula entails another)*.

In other words, the problem is to find in search space Ω_0 a conjuncted set of well-formed formulas Φ that satisfy all the given first-order structures in σ. Such a conjunction describes the "most precise" formula that satisfies all the given structures, because there is no satisfied formula which is entailed by any ϕ_i. The problem can be seen as an instance of the general specification synthesis problem [22] in the setting of positive-only learning [26]. An example of the problem is illustrated in App. B.

It is worth noting that we make several assumptions and simplifications to the problem definition above: (1) a disjunction of all input FO structures is always a valid (though overfit) solution, but in practice, meaningful formulas are to be found in a size-restricted search space; (2) search spaces with function symbols and constants can be easily handled by introducing new literals [28, §4], so we will avoid discussing them in detail; (3) we assume users want to synthesise prenex DNF formulas, as DS invariants are commonly expressed in this form; our techniques can be extended to other FO formulas.

2.2 A Glance on Existing Invariant Inference Algorithms

Existing invariant inference methods use different formula enumeration techniques and employ a variety of optimisations to effectively reduce the search space. We summarise nine representative existing approaches: I4 [18], FOL-IC3 [13], IC3PO [11], SWISS [12], DistAI [28], P-FOL-IC3 [14], DuoAI [27], Scimitar [25], and Flyvy [8] (in the order of their publication dates).

The categories are based on five detailed sub-aspects. The first two sub-aspects are "system-level", capturing the high-level design of a synthesis framework, while the remaining three are "algorithm-level": they define the pruning techniques used by the synthesis. Here we give a brief description of the sub-aspects (an interested reader can find the detailed study on each aspect and examples in App. C), with the concrete techniques followed by the tools using them.

- **Inference mode:** how the inference procedure uses FO formula synthesis, *e.g.*, one-shot synthesis by generalisation among traces [18, 28, 27], multi-shot synthesis by counter-example guidance [25, 8], and combined approaches [13, 11, 12, 14].
- **Language:** the syntactic restrictions imposed on the FO formulas, *e.g.*, effectively propositional logic (*a.k.a.* EPR [23, 6]) and its extensions (all), k-pseudo-DNF [14, 8], sub-template [28, 27], and other syntactic customisations [13, 12, 25, 8].
- **Redundancy:** pruning redundant formulas in the search space, *e.g.*, equivalence [18, 12, 28, 27, 8], tautology and contradiction [28, 27], and decomposition [27, 8].
- **Incrementality:** pruning the search space incrementally (from intermediate synthesis results), *e.g.*, entailment [12, 28, 27], co-implication [27], and implication graph [27].
- **Others:** other techniques used in the synthesis process, *e.g.*, symmetry v. quantification [11], inductive proof graph [25], and separation checking [13, 14].

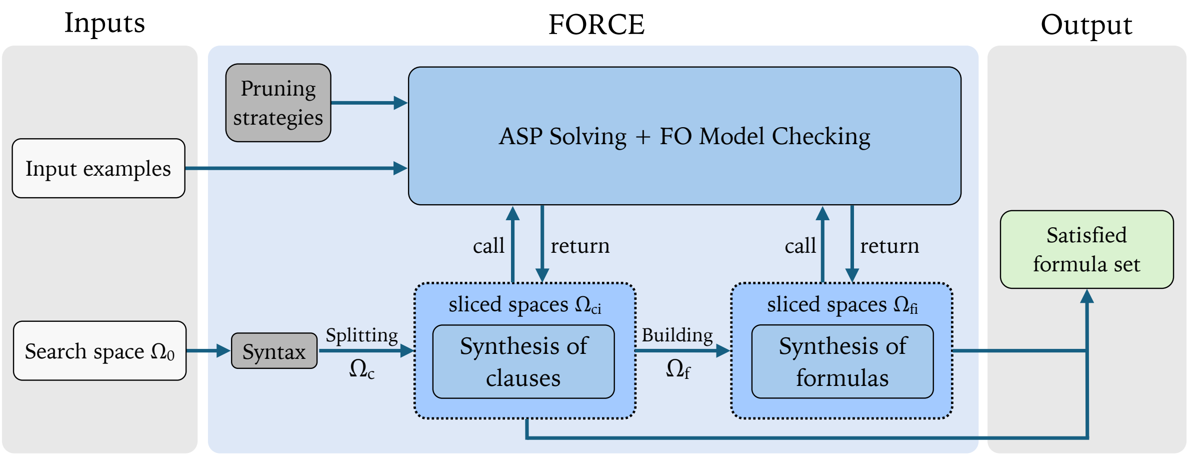

Fig. 1: The workflow of FORCE.

This taxonomy suggests a possibility of a unified framework to capture the existing techniques for FO synthesis and propose new ones. In particular, we find that the **Language** and **Redundancy** aspects together form a *static* search space, while the **Incrementality** aspect defines the *dynamic* pruning of the search space during the synthesis. These aspects are applicable to any FO synthesis problem. As for the remaining ones, **Inference mode** is about how the synthesis is used which is out of the scope of this work, and **Others** is about the specific techniques used in existing synthesis process which we will discuss in detail in Sec. 6.

2.3 FORCE: ASP-based Synthesis + Orthogonal Slices

Before the technical details, we give a high-level overview of our ASP-based synthesiser FORCE, whose workflow is shown in Fig. 1, by combining pruning strategies with the new *orthogonal slices* technique. The framework is customised by the grey parts: (1) the FO language (DNF by default) and (2) a set of pruning rules (predefined but extensible). Given the initial search space Ω_0 and a set of FO structures as input, FORCE starts by splitting the search space of clauses Ω_c from Ω_0. For further slices of Ω_c (called Ω_{c_1}, Ω_{c_2}, ...), the algorithm generates the formulas in Ω_{c_i} (by ASP solving), tests them on the examples (by model checking), and prunes the later slices ($\Omega_{c_j},\ j > i$) based on the results. Then using the output of Ω_c, the algorithm further builds the search space of formulas *other than clauses* Ω_f into $\Omega_{f_1}, \Omega_{f_2}, \ldots$, reusing the same process as in Ω_c to generate-test-prune. Finally, FORCE outputs both the clauses and the non-clause formulas that satisfy the inputs. It is called *orthogonal* because (1) slicing Ω_0 into Ω_c and Ω_f and (2) slicing of Ω_c and Ω_f are using different pruning strategies and work together.

3 Static Search Spaces of First-Order Formulas in ASP

We assume the reader is familiar with the basics of ASP, such as rules, choices, aggregates, and refer to the literature for more details [9]. This section demonstrates how ASP is suitable for encoding the static search space of FO formulas.

Our high-level approach follows the "generate-and-test" workflow, similar to other invariant synthesis systems, but differs in how the search space and pruning techniques are encoded. ASP is a paradigm suitable for such exhaustive enumeration with restrictions (search space customisations and pruning techniques in our case). As shown in Fig. 2, existing imperative approaches (demonstrated on the left) require manual integration of pruning steps (line 3-6), which can lead to brittle and hard-to-maintain code due

```
tmp ← init()
for all $x \in \Omega_0$ do
    if pruning1(x, tmp) then
        continue
    else if pruning2(x, tmp) then
        update2(tmp, x)
    else
        if Satisfied(x) then
            tmp.update(x)
return tmp.result
```

Algorithm 1 Extending Def. 1 with Customised Pruning Rules

```
1: function FOSYN($\Omega_0$, $\sigma$, prunings)
2:     solver ← init($\Omega_0$, prunings)
3:     while solver.solve() do
4:         x ← solver.get_current()
5:         if Satisfied(x) then
6:             yield x
```

Fig. 2: The synthesis loop (imperative programming vs. ASP solving).

to dependencies on intermediate results and evaluation order. In contrast, our ASP-based approach on the right modularises the search space and pruning strategies directly into the solver (line 2). This ASP-based "generate-and-test" loop improves extensibility and maintainability, by allowing easier integration of new pruning rules (as new domain knowledge) and automatic handling of dependencies. In the remainder of this section, we first demonstrate how to encode a basic search for FO formulas and then show the search is customised with different knowledge of pruning.

3.1 Encoding the Enumeration

Let us show how to encode FO formula enumeration in ASP. As an illustration, we use DuoAI's [27] configuration of FO search space for synthesising invariants of the `lockserv` distributed protocol.

```
var: node: n1, n2; lock: l1
relations: lock_msg:         node, lock;  grant_msg:  node, lock;
           unlock_msg:       node, lock;  holds_lock: node, lock;
           server_holds_lock: lock
max-literal: 4 max-or: 3 max-and: 3 max-exists: 1
```

The first two lines, "var" and "relations", together specify the variables (in `node` and `lock` sorts) and atoms (made out of five relations and valid variables) that can be used in the formulas. Note that the order of variables in formulas' prefix is usually fixed in EPR formulas, *i.e.*, the variable `l1` cannot appear before any node variable (`n1`, `n2`) in the prefix. The search space of formulas in DNF is then constrained by the problem-specific customisations: the maximum number of literals in the formula, the maximum number of disjunctions, the maximum number of cubes (*i.e.*, conjunctions of literals), and the maximum number of existential quantifiers. They together define the Ω_0 in Def. 1.

Thanks to the simplicity of FO formulas' prenex normal form, the enumeration can be easily encoded in ASP. The enumeration of formulas requires generating their two parts: prefixes and matrices, within the restrictions of the search space. The ASP encoding of the basic search space is illustrated as follows:

```
var(node, n1). var(node, n2). var(lock, l1).            % variables
0{exists(Var): var(Var, _)}1.                           % prefix bounded by max-exists
rel(lock_msg, (node, lock)). ...                        % relations
vars((node, lock), (n1, l1)). ...                       % variable tuples
atom(Pred, Args) :- rel(Pred, Types), vars(Types, Args). % atoms
pos(0..1). cube(1..3).                                  % sign of literal, max-or
0{lit_in_C(P,A,Pos,C):atom(P, A), pos(Pos)}3 :- cube(C). % matrix bounded by max-and
:- #count{P,A,Pos,C: lit_in_C(P,A,Pos,C)} >= 4.         % max-literal
```

In the program above, the prefix and matrix generation of formulas are achieved by the choice construct `0{...}n` on `exists()` and `lit_in_C()` predicates, which are restricted by the parameters in the configuration. The last line then eliminates the answer sets where more than four literals are in the corresponding formula. As an example, the answer set $\{\texttt{lit\_in\_C(lock\_msg,(n1,l1),0,1)}, \texttt{lit\_in\_C(grant\_msg,(n2,l1),1,1)}, \texttt{lit\_in\_C(unlock\_msg,(n1,l1),0,2)}, \texttt{exists(l1)}\}$ corresponds to $\forall\texttt{n1 n2},\exists\texttt{l1}.\ (\neg\texttt{lock\_msg(n1,l1)} \land \texttt{grant\_msg(n2,l1)}\ ) \lor \neg\texttt{unlock\_msg(n1,l1)}$.

3.2 Encoding Pruning by Redundancy

A particular encoding of a search space can output many answer sets whose corresponding formulas not necessarily satisfy the examples or even basic well-formedness constraints. For instance, the formula $\forall n_1.\texttt{lock\_msg}(n_2,l_1)$ should be ignored because n_2 is not in the prefix. As another example, $\textit{prefix}.\texttt{lock\_msg}(n_1,l_1) \lor \texttt{lock\_msg}(n_2,l_1)$ is equivalent to $\textit{prefix}.\texttt{lock\_msg}(n_2,l_1) \lor \texttt{lock\_msg}(n_1,l_1)$ but can be featured twice as two different answer sets. Our next step is, therefore, to encode those **Redundancy** techniques from Sec. 2.2 on the search space to only output well-formed formulas.

Let us illustrate the *symmetry-based normalisation* technique allowing to exploit equivalence using ASP; the remaining encoding of redundancy elimination can be found in our implementation. The normalisations are done by building a partial order on the formulas. That is, we can make sure that (1) the number of literals in cubes (from left to right in a DNF) is non-decreasing, (2) the minimal (in alphabetic order) literal in a cube i is less than the minimal literal in cube $j > i$ if their number of literals are the same, and (3) if two variables of one sort $v_i < v_j$, then the minimum predicate where v_i appears should be less or equal to the minimum predicate where v_j appears. The following ASP program encodes these pruning rules by constraining the orders in the formula.

```
lit_no_in(No, C) :- lit_in_C(P,A,Pos,C), lit_no(P,A,Pos,No).
num_lit(C, N) :- cube(C), #count{P,A,Pos: lit_in_C(P,A,Pos,C)} = N.
:- num_lit(C1,N1), num_lit(C2,N2), C1 < C2, N1 > N2.
min_lit(C, Min) :- cube(C), #min{No:lit_no_in(No,C)} = Min.
:- num_lit(C1,N), num_lit(C2,N), min_lit(C1,Min1), min_lit(C2,Min2), C1<C2, Min1>Min2.
min_lit_var(V, Min) :- used_var(V), #min{P:lit_in_C(P,A,_,_), var_in(V,A)} = Min.
:- min_lit_var(V1,Min1), min_lit_var(V2,Min2), V1 < V2, Min1 > Min2.
```

To summarise this section, we have shown that for the synthesis of FO formulas, the existing techniques can be easily encoded in ASP, which works as the backbone of our synthesiser. The expressive power of ASP allows us to encode existing techniques shown in Sec. 2.2 concisely and make them work together efficiently, being further optimised with our new pruning technique, described in the next section.

4 Dynamic Search Spaces via Orthogonal Slices

Armed with the ASP-based framework formula enumeration in the previous section, which allows one to combine pruning techniques, we propose *orthogonal slices* to handle the dynamic search space, which (1) generalises the state-of-the-art incremental pruning technique *implication graph* (IG), and (2) introduces a novel complementary pruning to resolve the bottleneck of IG. Both are achieved by slicing the FO search space into smaller ordered parts, and easily implemented by ASP's incremental solving.

4.1 Implication Graphs

Amongst the proposed pruning techniques for distributed system invariant synthesis in Sec. 2.2, *implication graph* of FO formulas is a prominent one. An IG is a directed graph where each node represents a formula, and an edge from A to B indicates that A implies B. The pruning is processed by removing formulas that are implied by already satisfied formulas from the search space, dynamically, to accelerate the synthesis process. That said, we can call it "pruning by satisfied formulas". The IG-based tool DuoAI [27] can infer inductive invariants for many complex protocols where no other existing tools can.

$$\forall x.P(x) \vee R(x) \tag{1}$$
$$\forall x.P(x) \vee S(x) \tag{2}$$
$$\forall x.P(x) \vee (R(x) \wedge S(x)) \tag{3}$$

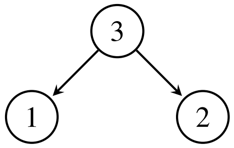

Fig. 3: Left: Formulas example. Right: Implication graph of the formulas.

Example 1 (An Illustration of the Implication Graph Shown in Fig. 3) *Formula* (3) *implies both formula* (1) *and formula* (2). *If formula* (3) *is satisfied, then formulas* (1) *and* (2) *are automatically satisfied and can be pruned from the search space.*

4.2 From the Formula to the Search Space

The first step of the orthogonal slices is to *abstract* the entailment relation from formulas to the level of the search spaces, then synthesise formulas following the partial order on the search spaces to achieve incremental pruning. We represent a FO formula search space by means of imposing syntactic constraints of the candidates, with the sets of possible values as arguments. As such, the search space is the Cartesian product of those sets, where each element describes a sub search space.

Definition 2 (Template of First-Order Formulas and Its Slicing) *A template T of first-order formulas is defined by a tuple of parameter sets $T = (P_1, \dots, P_n)$, where each P_i represents a set of possible values for a parameter. The search space $\Omega(T)$ defined by T is the Cartesian product of these parameter sets. A valid slicing of T (called $\mathtt{SL}(T)$) is defined as a partitioning of T into sliced-templates $\{T_1, \dots, T_j\}$, where each sliced-template $T_i = (P_{i1}, \dots, P_{in})$ corresponds to a subset of the parameter sets. The search space $\Omega(T)$ is then partitioned into subsets $\{\Omega(T_1), \dots, \Omega(T_j)\}$, where:*

$$\Omega(T) = \prod_{i=1}^{n} P_i. \qquad \bigcup_{i=1}^{j} \Omega(T_i) = \Omega(T), \quad \textit{and} \quad \Omega_m \cap \Omega_n = \emptyset \quad \textit{for all } m \neq n.$$

This ensures that the entire search space is covered, without overlap between slices, and each slice Ω_i corresponds to a sliced-template T_i.

Example 2 (A Template of `lockserv` in Sec. 3.1) • *The number of existential quantifiers ne,*

- *The number of variables used in each sort in a tuple $tv = (n_{v1}, \dots, n_{vi}, \dots)$,*
- *The number of literals in each cube sorted in a tuple $tl = (n_{l1}, \dots, n_{li}, \dots)$.*

And the parameters of the template (after the redundancy pruning in Sec. 3.2) are:

- $P_{ne} = \{0, 1\}$
- $P_{tv} = \{(0,1), (1,0), (1,1), (2,0), (2,1)\}$
- $P_{tl} = \{(1), (2), (3), (1,1), (1,2), (1,3), (2,2), (1,1,2)\}$

Now we describe the slicing for “pruning by satisfied formulas”, which is defined by the partial order of the sliced-templates (and their parameters).

Definition 3 (Partial Order for Parameters of Templates) *Given two parameters P_{im} and P_{in} as subsets of P_i in a template $T = (P_1, \dots, P_i, \dots, P_n)$, we say that P_{im} is less or equal than P_{in} (denoted as $P_{im} \preceq P_{in}$; $P_{im} \prec P_{in}$ in case $P_{im} \neq P_{in}$)* w.r.t. $\mathtt{SL}(T)$ *if*

$$\forall T_1 = (P'_1, \dots, P_{im}, \dots, P'_n),\ T_2 = (P'_1, \dots, P_{in}, \dots, P'_n) \in \mathtt{SL}(T),$$
$$\forall \phi \in \Omega(T_1),\ \exists \phi' \in \Omega(T_2) \textit{ such that } \phi' \models \phi.$$

Definition 4 (Partial Order for Sliced-templates) *Given two sliced-templates $T_i = (P_{i1}, \ldots, P_{in})$ and $T_j = (P_{j1}, \ldots, P_{jn})$ sliced from* $\mathtt{SL}(T)$*, we say that T_i is less or equal than T_j (denoted as $T_i \preceq T_j$) if $\forall k \in [1,n], P_{ik} \preceq P_{jk}$, where the equality holds if and only if $P_{ik} = P_{jk}$ for all $k \in [1,n]$.*

With two definitions above, the whole search space of FO formulas is sliced (and ordered) by the partial order; we call this procedure SPLITTEM. Note that possibly the search space's parameters are not as regular as in Ex. 2, but the worst case of the partial order is exactly the implication graph: the set of possible formula candidates is the only parameter, and the partial order is defined by FO entailment. This says, the parameterisation of FO search space by syntactic constraints is "always" possible.

Example 3 (Partial Order of the Templates in Ex. 2) *The partial order of the templates' parameters of* `lockserv` *is defined as follows:*

- *For n_e, it follows the integer order: $\{0\} \prec \{1\}$.*
- *For t_v, the set $\{(n_{v1}, n_{v2}, \ldots, n_{vi}, \ldots)\} \prec \{(n_{v1}, n_{v2}, \ldots, n_{vi}+1, \ldots)\}$*
- *For t_l, the set $\{(n_{l1}, n_{l2}, \ldots, n_{li}, \ldots)\} \prec \{(n_{l1}, n_{l2}, \ldots, n_{li}-1, \ldots)\}$ and $\{(n_{l1}, n_{l2}, \ldots, n_{li}, \ldots)\} \prec \{(n_{l1}+1, n_{l2}, \ldots, n_{li}, 1)\}$*

The partial order on template parameters is defined consistently with formula entailment relations: by swapping $\forall$ into $\exists$, a formula becomes more general; by replacing one variable with a fresh one (together with the formula decomposition, proven in [28, §4]), a formula becomes more general; by deleting a literal from a cube or adding a new cube, a formula becomes more general. Therefore, by obtaining the partial order of a slicing of the search space, the "pruning by satisfied formulas" is naturally achieved.

The ASP encoding of "pruning by satisfied formulas" is done by multi-shot solving [10], which is standard to achieve incremental solving in ASP. The sketch of its encoding is as follows:

```
#program inv(prefix,matrix).
:- output(Prefix, Matrix), pre_weaken(Prefix, prefix), mat_weaken(Matrix, matrix).
```

which says that the formula entailed by the `inv` (*i.e.*, a satisfied formula) should not be generated. The entailment checking is implemented by variable substitution together with prefix and matrix weakening (detailed in our implementation).

4.3 Slicing DNF Modulo Clauses

The problem of implication graph (or "pruning by satisfied formulas" in general) is its scalability: the search starts from the root to leaves of the graph (top to bottom in Fig. 3), but the number of root nodes is still exponential in the search space's size. To see the issue, let us take the formula *prefix*. $(lit_{11} \wedge lit_{12} \wedge lit_{13}) \vee (lit_{21} \wedge lit_{22})$ as an example: it can be a root node used to prune the formula *prefix*. $lit_{1i} \vee lit_{2j}$ because of the entailment. However, checking all root formulas has complexity of $O(n^5)$ (n is the number of literals), but this effort prunes only formulas using $O(n^2)$ space.

Intuitively, if "pruning by satisfied formulas" is costly when using the result of a larger search space to prune the smaller one, its dual version–"pruning by unsatisfied formulas", should solve it. This idea, however, is not immediately applicable for two reasons: (1) the two pruning strategies have different directions (general-to-specific v. specific-to-general), which means an algorithm needs to deal with both, and (2) for complex problems, the majority of formulas are unsatisfied (they do not cover all examples), which means reducing the search space too many times incurs a large performance overhead.

Our solution to the first issue is simply another slicing, which is orthogonal to the slicing of templates: first synthesising clauses (disjunctions of literals) by slicing them from the whole search space (named

SPLITDNF), and then pruning the search of DNF synthesis based on the unsatisfied clauses. To further address the second issue, we avoid the high-cost of "pruning by unsatisfied clauses" by *constructing* the DNF search space from the satisfied clauses. The definition of DNF search space construction given below describes the pruning of this slicing.

Definition 5 (Possibly Satisfied DNF Modulo Clauses) *Given a set of satisfied (*w.r.t. *input examples) clauses Φ_c, a formula in DNF of the form prefix.* $(lit_{11} \wedge \ldots \wedge lit_{1k_1}) \vee \ldots \vee (lit_{m1} \wedge \ldots \wedge lit_{mk_m})$ *is possibly satisfied only if* $\forall l_i \in \bigcup_{i=1}^{m}\{lit_{i1}, \ldots, lit_{ik_i}\}$, *prefix.* $l_1 \wedge \ldots \wedge l_m$ *is satisfied* w.r.t. Φ_c.

The function constructing DNFs' candidates by Def. 5 is denoted as BUILDFROMCLAUSES. In plain words, a formula in DNF is added into the search space only when all clauses it entails satisfy all input FO structures. The reason we resolve the bottleneck of "pruning by satisfied formulas" is evident: the search space of clauses, which is exponential in the number of cubes, is much smaller than the search space of DNF (exponential in the number of literals) for most cases. Moreover, since this search space construction is also naturally encoded by ASP, all existing techniques to formulas in Sec. 2.2 are directly applied to further refine this search space built from clauses.

4.4 The Synthesis Algorithm

Algorithm 2 The Core Algorithm of FORCE

```
1:  function FORCE($\Omega_0$, $\sigma$, $prunings = pre\_def$)
2:      $\Phi_c$, $\Phi_f \leftarrow \emptyset$
3:      $\Omega_c \leftarrow$ SPLITDNF($\Omega_0$)
4:      for $\Omega_{ci} \in$ SPLITTEM($\Omega_c$) do
5:          $\Phi_i \leftarrow$ FOSYN($\Omega_{ci}$, $\sigma$, $prunings$)
6:          $\Phi_c \leftarrow \Phi_c \cup \Phi_i$
7:          $prunings \leftarrow prunings.update(\Phi_i)$
8:      $\Omega_f \leftarrow$ BUILDFROMCLAUSES($\Phi_c$, $\Omega_0$)
9:      for $\Omega_{fi} \in$ SPLITTEM($\Omega_f$) do
10:         $\Phi_i \leftarrow$ FOSYN($\Omega_{fi}$, $\sigma$, $prunings$)
11:         $\Phi_f \leftarrow \Phi_f \cup \Phi_i$
12:         $prunings \leftarrow prunings.update(\Phi_i)$
13:     $\Phi_c \leftarrow$ FILTERIMPLIED($\Phi_c$, $\Phi_f$)
14:     return $\Phi_c \cup \Phi_f$
```

Given the formula synthesis algorithm in Fig. 1 for an input search space, and the two approaches to slice the search space, the procedure for synthesising formulas is given by Algorithm 2: first synthesise clauses by its sliced template (lines 4 to 7), then synthesise other formulas by the sliced template of DNF modulo clauses (lines 9 to 12), finally normalising the satisfied formulas and output (lines 13 and 14). The input *pruning* rules are customisable, but we pre-defined existing redundant and incremental rules in Sec. 2.2. The soundness of the algorithm *w.r.t.* orthogonal slices (*i.e.*, it does not over-prune) is guaranteed by the fact that if a formula A is pruned, A is either unsatisfied, having being pruned by unsatisfied clauses in DNF modulo clauses, or satisfied but more general than a satisfied formula which prunes A in a sliced template.

5 Experimental Evaluation

5.1 Implementation and Settings

Our implementation of FORCE combines the use of ASP (250 lines for the enumeration plus 80 lines for pruning, which is extremely concise to encode the different existing and new pruning techniques), and C++ to call APIs of Clingo [9] (400 lines for the synthesiser call functions, and 200 lines to integrate with other systems). The benchmarks are run on a MacBook Pro with 8-core M1 Pro CPU, 16GB RAM. The current version of FORCE is available at `https://zenodo.org/records/15654427`.

We ran FORCE on a set of benchmarks from the work on DuoAI [27], the most efficient state-of-the-art DS invariant inference tool. Briefly speaking, DuoAI's algorithm is an enumeration-based inductive

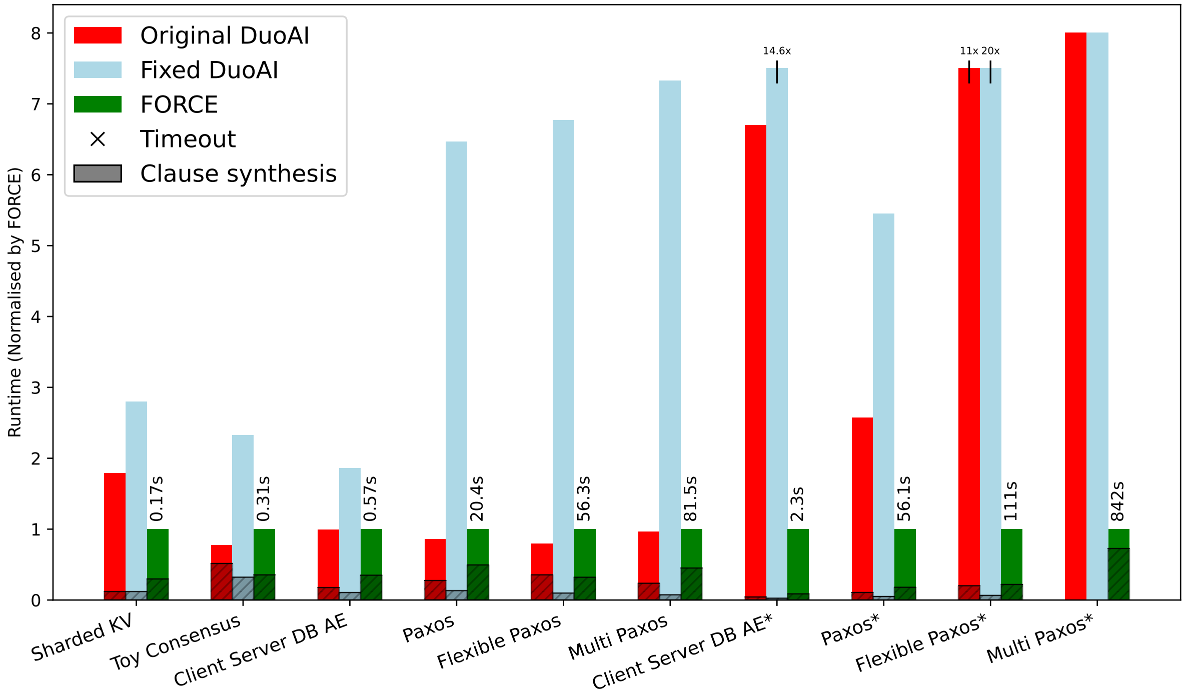

Fig. 4: Invariant synthesis. Results are normalised *w.r.t.* FORCE run times.

synthesis using sampled traces of the protocols as examples, followed by an optimised Houdini [7]. Since the formula enumeration component has non-negligible run time (detailed later), reducing the overhead of the enumeration part can significantly improve the overall performance.

Note that (at least) two unsound optimisations were made in DuoAI: the first is an *over-pruning*, found and rectified by [8]: DuoAI identifies formulas that can be decomposed into smaller formulas, then not testing the original formula to reduce the search space; however, a subset of those should not be decomposable, which leads to unsoundness by missing possibly satisfied formulas. The second unsound optimisation is the use of *restricted quantifiers* of formulas; specifically, the default setting of DuoAI only allows arbitrary quantifiers over the last three variables in a template (*e.g.*, for $Q_1X_1.Q_2X_2.Q_3X_3.Q_4X_4$, Q_1 can only be $\forall$), which results in an incomplete search in general, but practically works for their benchmarks. We treat the first issue as a bug, and the second observation as domain-specific knowledge that encodes an extra constraint on the search space.

5.2 Results and Analysis

Our performance statistics are shown in Fig. 4. We selected six complex distributed protocols from the DuoAI suite for evaluation on synthesising inductive invariants. FORCE was given the same input as DuoAI: traces as input examples and a search space configuration which contains the inductive invariant. For each protocol, we benchmark with two configurations: one with the restricted quantifier limitation from DuoAI and one without it. The unrestricted quantifier setting is indicated by a $*$ in the figure when enabled. The reason for showing only 10 data points, rather than 12, is that the difference of quantifier restriction did not affect the small search space for the first two smaller protocols. We fixed the over-pruning bug in DuoAI (referring to the result as “fixed DuoAI”) with our best effort for a fair comparison. That said, we also provide results of the “original” DuoAI version as a reference point.

The results show that FORCE significantly outperforms fixed DuoAI in all benchmarks, and also beats the original (unsound) DuoAI in most cases. The difference in performance is larger without the quantifier restriction. As a reference, the original DuoAI's runtime to synthesise invariants for Paxos, FlexiblePaxos and MultiPaxos are 60.4s, 78.7s and 1,549s, respectively; this means the enumeration in DuoAI is the main bottleneck for complex protocols, so the improvement is effective for overall runtime.

The improvement is mainly sourced from the effective pruning by DNF modulo clauses (Sec. 4.3): as the *clause synthesis* part in the bars shows, FORCE takes (slightly) longer time on the clause synthesis than DuoAI, and results in a much shorter time for the remaining synthesis by reducing the search space. The huge difference for cases without quantifier restriction is also explained for the same reason: taking Paxos as an example, the number of satisfied clauses increased from 134 to 141 when disabling the restriction in FORCE, which means the increased search space for DNF is not much (with 36s difference of time); but for DuoAI, the enumeration explores the whole new search space (thus 76s difference in total). Note that abstracting the formula into the search space as in Sec. 4.2 allows parallelism of the formula synthesis in FORCE, which also contributes to the efficiency, but it is specific to (1) the implementation of parallelism, and (2) the multi-threaded solving in Clingo, which is not essential to discuss here. The overall extra time when disabling the parallelism in our system varies from 50% to 100%, which shows potential improvement with a better implementation of parallelism.

Another interesting comparison *w.r.t.* to the unsoundness in DuoAI is shown between the red and blue bars: the fixed version takes a longer time to explore the (now larger) search space, but the clause synthesis time is lower, because additional discovered satisfied formulas prune the clauses entailed (which also implies the unsoundness of original DuoAI). By analogy with our orthogonal slices, they are using results from "larger slices" to prune the "smaller slices", which results in inefficiency, as expected (see Sec. 4.3). From a high-level aspect, this comparison illustrates the common trade-off between completeness and efficiency in synthesis tasks, but FORCE achieves complete search with even better efficiency comparing to the original DuoAI by spending minor extra time (in the green bars) to synthesise clauses.

5.3 On Composability of FORCE

More than the performance benefits of the orthogonal slices, the "solver-aided" feature of FORCE is also promising: it works as a general framework to combine different approaches by providing a common interface for FO synthesis. To illustrate this, we built the bridges for both DuoAI and Flyvy [8] (about 200 lines of code for each), and used the clause output (*i.e.*, part of Fig. 2 before line 8) to optimise Flyvy.

The high-level task of Flyvy is to output the *strongest* inductive invariant (that contains a set of FOL formulas) given a distributed system protocol and a bounded FO language. Without describing the details of Flyvy's algorithm, it is clear that if Flyvy is obtaining formulas, those formulas can be bounded by DNF modulo clauses. We use the traces obtained from DuoAI together with the bounded language to synthesise satisfied clauses and output them as an ASP program by FORCE, then restricting the formulas in Flyvy by calling the ASP program to filter out the unnecessary formulas. The results show that, for two most complex protocols can be synthesised in Flyvy, Paxos and FlexiblePaxos, the sizes of the output invariants are reduced by about 13% (165/1260) and 7% (61/816) respectively on average (detailed in App. D), and they remain strongest in the sense that unnecessary formulas are removed during the intermediate steps of Flyvy. We believe it is a promising direction to explore different combinations of existing approaches with the power of FORCE.

6 Related Work and Discussion

Inductive Logic Programming. Our work is closely related to the field of Inductive Logic Programming (ILP, [20, 3]), which focuses on learning *logic programs* inductively. Concretely, the state-of-the-art

ILP system Popper [4] shares a similar workflow with FORCE: in a "generate-test-prune" loop, both systems use ASP to generate the candidate and prune the search. There are two main differences between FORCE and Popper: (1) the preferred candidate (among all correct solutions) in ILP is the most general program because both positive and negative examples are given; while in FORCE, the output is the conjunction of most specific formulas that satisfies all positive examples, because "logical true" is always a valid general formula in absence of negative examples. (2) the properties of FO formulas expose more domain knowledge to be encoded by ASP for pruning compared to Horn clauses.

Other Approaches for Invariant Inference. Among the techniques shown in Sec. 2.3, techniques from three tools (in the **Other** category) have not been encoded into FORCE yet. The first two have similarities in how they improve the inference process: FOL-IC3 [13] is an inference framework building on a first-order logic *separability* solver; IC3PO [11] is an invariant inference tool based on the relation between the *symmetry* of satisfied FO structures and the necessary *quantification* in satisfying formulas in FOL. Both of them are based on PDR/IC3 [2], where the generalisation of FO structures is performed based on a small set of examples (usually less than 10, in contrast with DuoAI, SWISS, and FORCE, where the generalisation is among thousands of examples). From a technical perspective, their techniques can be encoded into FORCE, since both are SAT-based. However, the scalability of their techniques make them not generally applicable to large sets of examples, which we will discuss later.

Scimitar [25], unlike most of the related works where a "global" inductive invariant is inferred, is a tool that infers "local" invariants for each transition state in the distributed system protocol. It builds an *inductive proof graph* that abstracts the protocols, and locally synthesises the invariants for each node in the graph, where the local synthesis problem is an instance of our FO synthesis.

Discussion on Inductive Generalisation. The least general generalisation [24] is known to be a foundational result in the field of bottom-up inductive logic programming, where the given programs (bottoms) are generalised to the general program (top). In contrast, FORCE is a top-down approach by "generate-and-test". Notably, in the domain of distributed systems invariant inference, the authors of the SWISS tool [12] described their "failed attempt" to use constraint solving for bottom-up generalisation (without a detailed explanation). This is coincident with our observation that the bottom-up generalisation (like FOL-IC3 and IC3PO) is not scalable to large sets of examples compared to the top-down approach. We provide our own explanation: given a whole search space, any example can be regarded as a constraint to prune the search; while the cost for the constraints can be considered linear in the number of examples, the benefit of pruning decreases as the number of constraints increases. This brings a possible further work to combine the top-down and bottom-up approaches by using subsets of examples for utilising the bottom-up generalisation.

Outside the domain discussed above, [15] is a notable work from automata theory that studies the learnability (*w.r.t.* generalisation) of formulas in finite variable logic. It does not focus on the concrete algorithm in practice, but show potential to extend FORCE to logics more general than FOL.

7 Conclusion

In this work, we proposed a unified ASP-based framework FORCE for synthesising formulas in first-order logic from examples. To do so, we used ASP as a framework for implementing inductive formula synthesis offering constraint solving to encode the search and rule-based knowledge to prune the search space. Using our ASP-based encoding, we proposed *orthogonal slices*—a novel technique that significantly accelerates formula synthesis. Finally, we have shown that declaratively capturing the essence of different approaches for formula synthesis in ASP enables a more efficient and composable solution.

References

[1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli & Yoni Zohar (2022): *cvc5: A Versatile and Industrial-Strength SMT Solver*. In: *TACAS*, *LNCS* 13243, Springer, pp. 415–442, doi:10.1007/978-3-030-99524-9_24.

[2] Aaron R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In: *VMCAI*, *LNCS* 6538, Springer, pp. 70–87, doi:10.1007/978-3-642-18275-4_7.

[3] Andrew Cropper & Sebastijan Dumancic (2022): *Inductive Logic Programming At 30: A New Introduction*. *J. Artif. Intell. Res.* 74, pp. 765–850, doi:10.1613/JAIR.1.13507.

[4] Andrew Cropper & Rolf Morel (2021): *Learning programs by learning from failures*. *Machine Learning* 110(4), pp. 801–856.

[5] Niklas Eén, Alan Mishchenko & Robert K. Brayton (2011): *Efficient implementation of property directed reachability*. In: *FMCAD*, FMCAD Inc., pp. 125–134.

[6] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin & Andrei Voronkov (2010): *Encoding industrial hardware verification problems into effectively propositional logic*. In: *FMCAD*, IEEE, pp. 137–144.

[7] Cormac Flanagan & K. Rustan M. Leino (2001): *Houdini, an Annotation Assistant for ESC/Java*. In: *FME*, *LNCS* 2021, Springer, pp. 500–517, doi:10.1007/3-540-45251-6_29.

[8] Eden Frenkel, Tej Chajed, Oded Padon & Sharon Shoham (2024): *Efficient Implementation of an Abstract Domain of Quantified First-Order Formulas*. *CoRR* abs/2405.10308, doi:10.48550/ARXIV.2405.10308. arXiv:2405.10308.

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2012): *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, doi:10.2200/S00457ED1V01Y201211AIM019.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2019): *Multi-shot ASP solving with clingo*. *Theory Pract. Log. Program.* 19(1), pp. 27–82, doi:10.1017/S1471068418000054.

[11] Aman Goel & Karem A. Sakallah (2021): *On Symmetry and Quantification: A New Approach to Verify Distributed Protocols*. In: *NASA Formal Methods*, *LNCS* 12673, Springer, doi:10.1007/978-3-030-76384-8_9.

[12] Travis Hance, Marijn Heule, Ruben Martins & Bryan Parno (2021): *Finding Invariants of Distributed Systems: It's a Small (Enough) World After All*. In: *NSDI*, USENIX Association, pp. 115–131. Available at `https://www.usenix.org/conference/nsdi21/presentation/hance`.

[13] Jason R. Koenig, Oded Padon, Neil Immerman & Alex Aiken (2020): *First-order quantified separators*. In: *PLDI*, ACM, pp. 703–717, doi:10.1145/3385412.3386018.

[14] Jason R. Koenig, Oded Padon, Sharon Shoham & Alex Aiken (2022): *Inferring Invariants with Quantifier Alternations: Taming the Search Space Explosion*. In: *TACAS*, *LNCS* 13243, Springer, pp. 338–356, doi:10.1007/978-3-030-99524-9_18.

[15] Paul Krogmeier & P. Madhusudan (2022): *Learning formulas in finite variable logics*. *Proc. ACM Program. Lang.* 6(POPL), pp. 1–28, doi:10.1145/3498671.

[16] Leslie Lamport (1998): *The Part-Time Parliament*. *ACM Trans. Comput. Syst.* 16(2), pp. 133–169, doi:10.1145/279227.279229. Available at `https://doi.org/10.1145/279227.279229`.

[17] Vladimir Lifschitz (2002): *Answer set programming and plan generation*. *Artificial Intelligence* 138(1-2), pp. 39–54.

[18] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci & Karem A. Sakallah (2019): *I4: incremental inference of inductive invariants for verification of distributed protocols*. In: *SOSP*, ACM, pp. 370–384, doi:10.1145/3341301.3359651.

[19] Leonardo Mendonça de Moura & Nikolaj S. Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[20] Stephen H. Muggleton & Luc De Raedt (1994): *Inductive Logic Programming: Theory and Methods*. *J. Log. Program.* 19/20, pp. 629–679, doi:10.1016/0743-1066(94)90035-3.

[21] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv & Sharon Shoham (2016): *Ivy: Safety Verification by Interactive Generalization*. In: *PLDI*, ACM, pp. 614–630, doi:10.1145/2908080.2908118.

[22] Kanghee Park, Loris D'Antoni & Thomas W. Reps (2023): *Synthesizing Specifications*. *Proc. ACM Program. Lang.* 7(OOPSLA2), pp. 1787–1816, doi:10.1145/3622861.

[23] Ruzica Piskac, Leonardo Mendonça de Moura & Nikolaj S. Bjørner (2010): *Deciding Effectively Propositional Logic Using DPLL and Substitution Sets*. *J. Autom. Reason.* 44(4), pp. 401–424, doi:10.1007/S10817-009-9161-6.

[24] Gordon D Plotkin (1970): *A note on inductive generalization*. *Machine intelligence* 5(1), pp. 153–163.

[25] William Schultz, Edward Ashton, Heidi Howard & Stavros Tripakis (2024): *Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing*. *CoRR* abs/2404.18048, doi:10.48550/ARXIV.2404.18048. arXiv:2404.18048.

[26] Ziyi Yang & Ilya Sergey (2025): *Inductive Synthesis of Inductive Heap Predicates*. *Proc. ACM Program. Lang.* 9(OOPSLA1), pp. 169–195, doi:10.1145/3720420.

[27] Jianan Yao, Runzhou Tao, Ronghui Gu & Jason Nieh (2022): *DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols*. In: *OSDI*, USENIX Association, pp. 485–501. Available at https://www.usenix.org/conference/osdi22/presentation/yao.

[28] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana & Gabriel Ryan (2021): *DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols*. In: *OSDI*, USENIX Association, pp. 405–421. Available at https://www.usenix.org/conference/osdi21/presentation/yao.

A Conceptual Connection between Synthesis and ASP

Looking back to the whole synthesis process, both static and dynamic search spaces we discussed are essentially achieved by different set operations, which is the algorithmic reason for us to use ASP. In the table below, we summarise the connection between the synthesis problem and ASP by unifying them as set operations, which hopefully helps the readers from either synthesis or ASP background to understand "why and what synthesis task is suitable for ASP".

Synthesis	**ASP**	**Connection**
Parameterised search space	choice constructs	set Cartesian product
Slicing	external statement	set partition
Static pruning	integrity constraint	set comprehension
Dynamic pruning	multi-shot solving	set union, difference

B An Example of Def. 1 in Sec. 2.1

This section provides an example of the FO synthesis problem.

Example 4 (A tiny example) *Considering a FO signature $\Sigma = \langle C, R, F, S \rangle$ with C and F empty, $R = \{p, q, r\}$, and $S = \{X\}$. The synthesis problem is defined with the inputs:*

- *the set of FO formulas is restricted to the form of* $\forall X : lit_1$ *or* $\forall X : lit_1 \vee lit_2$*, which is a disjunction of one or two literals with one universal quantifier. Without the pruning,* Ω_0 *contains* $(2*3)+(2*3)^2 = 42$ *formulas in total.*
- *the set of FO structures is* $\sigma = \{M_1, M_2\}$*, where* M_1 *and* M_2 *are two models over* Σ *sharing the universe* $\{x_0, x_1, x_2\} \in X$ *and the interpretations of p, q, and r are:*
  1. M_1*: [*$p^{M_1} = \{x_0,\ x_1\}$*,* $q^{M_1} = \{x_1,\ x_2\}$*,* $r^{M_1} = \emptyset$*]*
  2. M_2*: [*$p^{M_2} = \{x_0,\ x_1\}$*,* $q^{M_2} = \{x_2\}$*,* $r^{M_2} = \{x_1\}$*]*

And the output of the synthesis problem based on Def. 1 is a set of formulas

$$\Phi = \{\forall X.p(X) \vee q(X), \forall X.p(X) \vee \neg r(X)\}.$$

As a simple case of the sliced-template, our algorithm will first find satisfied formula in $\forall X : lit_1$ (and fails), then checking the satisfied formula in $\forall X : lit_1 \vee lit_2$ (with the two formulas successfully found). The inputs of FORCE, the bounded FO search space and the real traces of distributed system protocols, are much more complex. An interested reader can refer to `configs/` and `traces/` folders in `https://github.com/verse-lab/FORCE` for the real instances.

C Detailed Taxonomy of Invariant Synthesis Techniques

In this section, we provide our detailed study on the synthesis techniques used in existing invariant inference tools outlined in Sec. 2.2.

C.1 System-Level Aspects

Inference mode. This aspect determines the way the overall inference procedure uses FO formula synthesis. In particular, *one-shot* synthesis generates all satisfied formulas in the search space given fixed input examples (*e.g.*, sampled traces of a distributed protocol), while *multi-shot* synthesis generates a set of candidate formulas incrementally, based on examples obtained incrementally (*e.g.*, counter-examples of current invariants). Combined approaches use system traces or counter examples to guide the multi-shot synthesis, calling the synthesis procedure multiple times.

Historically, the majority of multi-shot synthesis algorithms can be seen as extensions of IC3 [2], while one-shot synthesis can be considered as extensions of Houdini [7]. Crucially, both modes share the underlying FO synthesis problem similar to the definition of our synthesis problem, which means our FORCE framework can be used to improve most existing approaches.

Language restriction. To make the synthesis problem tractable, it is frequently defined for a subclass of first-order logic. The most common one is the Effectively Propositional Logic (EPR) fragment, a subset of FOL in which formulas can be transformed into equivalent propositional formulas, allowing for provably decidable verification.

While EPR provides a theoretical decidability guarantee, in practice, synthesis approaches often impose additional other syntactic constraints. For example, the k-pseudo-DNF proposed in P-FOL-IC3 [14] is a syntax restriction for practical efficiency: it is based on the observation that the invariant formulas written in such form are smaller than standard DNF, reducing the search space that needs to be explored. Specifically, a k-pseudo-DNF formula has the matrix in the form of $c_1 \rightarrow (c_2 \vee \ldots \vee c_k)$, where

c_i is a conjunction of literals. This is essentially a heuristic of the search, which makes sense because implications are commonly used to express invariants.

DistAI [28] proposed sub-templates for efficiency, exploiting the following property of first-order logic with equality:

$$\forall X_1, X_2 : T.mat(X_1, X_2) \equiv \forall X : T.mat(X, X) \wedge \forall X_1 \neq X_2 : T.mat(X_1, X_2) \tag{0}$$

With such a property, formula synthesis can avoid enumerating formulas on the LHS of Eq. 0 because they are equivalent to the conjunction of the two RHS formulas. Therefore, the only enumerations among two sub-templates on the right are required to synthesis the satisfied formula on the left, resulting in an accelerated enumeration.

However, upon close examination of existing invariant inference implementations, we discovered that syntactic constraints are not extensible in many tools. For example, extending k-pseudo-DNF with sub-templates would require modifying any algorithm that manipulates formulas in P-FOL-IC3.

C.2 Algorithm-Level Aspects

Since the search space of first-order formulas is large and thus intractable for a brute-force search, existing tools introduce algorithmic pruning techniques. We characterise pruning techniques according to the remaining three aspects in Sec. 2.2. Since in Sec. 6 and Sec. 4.1 we have detailed the last one of them and the implication graph, here we discuss the remaining ones.

Redundancy elimination. The most common pruning to apply is redundancy elimination, which is often used in synthesis. The idea is to simply eliminate the formulas that are equivalent to each other. For many tools, approximations such as "pruning by symmetry" (shown in Sec. 3) are used, which uses the symmetry of formulas under quantification to identify and prune away redundant ones. More than the syntactic-based equivalence, the redundancy can also be introduced by semantics (*e.g.*, tautology and contradiction), where many case-by-case rules are used in DuoAI. An example of formulas eliminated by this kind of redundancy is the one containing $p(X) \wedge \neg p(X)$ or $p(X) \vee \neg p(X)$ as sub-expressions. Another approach to redundancy is elimination *canonicalisation*, which is implemented by Flyvy [8]: it also uses symmetry breaking, but defines a partial order of formula sets (instead of the individual formulas) to eliminate redundancy.

A more complex but very effective pruning strategy of this category is proposed in DuoAI [27], where a large number of DNF formulas are shown to be redundant by their *decomposition*: $A \equiv B \wedge C$. Similar to Eq. 0, the formula A can be decomposed into a conjunction of smaller formulas, so the original formula can be pruned if all the smaller formulas are in the search space. However, the authors of Flyvy (*cf.* [8, Appendix D]) found that the decomposition in DuoAI is unsound in certain cases (*i.e.*, it leads to over-pruning), and proposed an amended version. We found it not easy to switch the DuoAI implementation to the amended version, as it uses the intermediate results of the original decomposition in the overall synthesis loop, which also makes it challenging to fairly compare the efficiency of the two methods.

Incremental pruning. Another inherent part of nearly all efficient approaches to FOL formula synthesis is incremental pruning. The idea is to test formulas against the set of input FO structures in a specific order, exploiting the entailment relation between formulas to eliminate the need to test some of them.

As introduced in Sec. 4.1, the *implication graph* is an effective technique of this aspect. More than this, DuoAI also identify another incremental pruning technique is based on *co-implication*, which

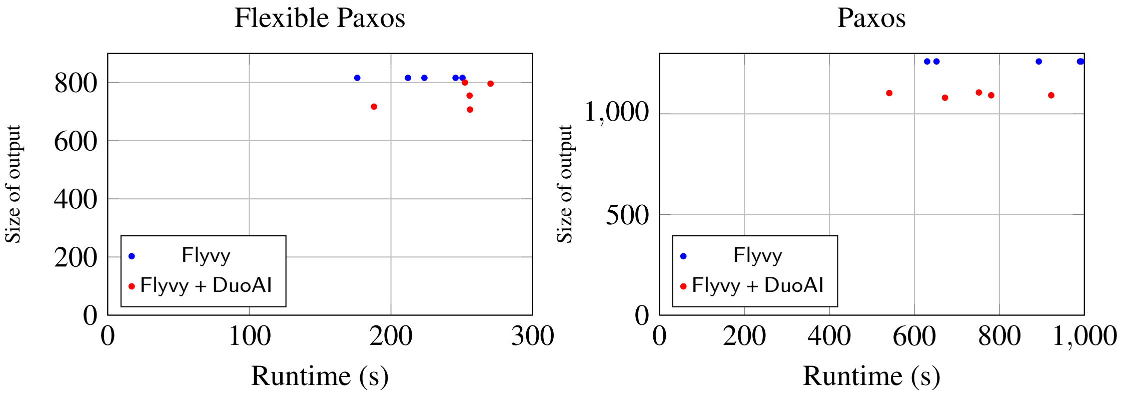

Fig. 5: Optimising Flyvy via DuoAI's output.

combines the intermediate results of FO model checking with the redundancy elimination to prune the search space. For example, if we find that formula $\forall x. P(x) \Rightarrow R(x)$ satisfies all examples, we do not need to the test any formulas of the form *prefix*. $(P(x) \land R(x) \land F) \lor G$, since they are co-implied by the original formula and *prefix*. $(P(x) \land F) \lor G$.

The two kinds of pruning discussed above are essential in accelerating the search of FO formulas. Their correctness is justified by the properties of FOL, such as entailment and equivalence. Importantly, this means that one can see the synthesis of FO formulas as enumeration *modulo* property-based pruning.

D Additional Statistics of Sec. 5.3

As said in Sec. 5.3, there are only two non-trivial protocols in Flyvy's benchmark that have quantifier alternation, for which we can produce comparable results. For both protocols, we execute the original Flyvy and the optimised Flyvy for 5 times with the sizes of the output (been discussed) and the runtime shown in Fig. 5. Performance-wise, we did not achieve notable improvement on the runtime of Flyvy, which is not unexpected, since the bottleneck of Flyvy is not the synthesis; instead, the SMT solving in Flyvy dominates the runtime and makes the runtime unstable.