LOUD: Synthesizing Strongest and Weakest Specifications

KANGHEE PARK*, University of California San Diego, USA
XUANYU PENG*, University of California San Diego, USA
LORIS D'ANTONI, University of California San Diego, USA

This paper tackles the problem of synthesizing specifications for nondeterministic programs. For such programs, useful specifications can capture demonic properties, which hold for *every* nondeterministic execution, but also angelic properties, which hold for *some* nondeterministic execution. We build on top of a recently proposed SPYRO framework in which given (*i*) a *quantifier-free* query Ψ posed about a set of function definitions (i.e., the behavior for which we want to generate a specification), and (*ii*) a language $\mathcal{L}$ in which each extracted property is to be expressed (we call properties in the language $\mathcal{L}$-properties), the goal is to synthesize a conjunction $\bigwedge_i \varphi_i$ of $\mathcal{L}$-properties such that each of the φ_i is a *strongest $\mathcal{L}$-consequence* for Ψ: φ_i is an over-approximation of Ψ and there is no other $\mathcal{L}$-property that over-approximates Ψ and is strictly more precise than φ_i. This framework does not apply to nondeterministic programs for two reasons: it does not support existential quantifiers in queries (which are necessary to expressing nondeterminism) and it can only compute $\mathcal{L}$-consequences, i.e., it is unsuitable for capturing both angelic and demonic properties.

This paper addresses these two limitations and presents a framework, LOUD, for synthesizing both *strongest $\mathcal{L}$-consequences* and *weakest $\mathcal{L}$-implicants* (i.e., under-approximations of the query Ψ) for queries that can involve *existential quantifiers*. We devise algorithms for handling the quantifiers appearing in LOUD queries and implement them in a solver, ASPIRE, for problems expressed in LOUD which can be used to describe and identify sources of bugs in both deterministic and nondeterministic programs, extract properties from concurrent programs, and synthesize winning strategies in two-player games.

CCS Concepts: • **Theory of computation → Program specifications**; **Abstraction**; • **Software and its engineering → Automated static analysis**; **Automatic programming**.

Additional Key Words and Phrases: Program Specifications, Program Synthesis

ACM Reference Format:
Kanghee Park, Xuanyu Peng, and Loris D'Antoni. 2025. LOUD: Synthesizing Strongest and Weakest Specifications. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 114 (April 2025), 28 pages. https://doi.org/10.1145/3720470

1 Introduction

Specifications allow us to understand what programs do, but are often hard to write and maintain. Writing specifications is especially hard for programs involving nondeterminism, a construct necessary to model many practical applications such as concurrency, random execution, and games. Part of what makes writing such specifications hard is that specifications for programs involving nondeterminism might capture two types of properties: *demonic properties*, which hold for *every* nondeterministic execution, and *angelic properties*, which hold for *some* nondeterministic execution.

Consider a program that nondeterministically shuffles the elements of a list. A possible valid demonic property could state that the output list is always a permutation of the input list. However,

*Kanghee Park and Xuanyu Peng contributed equally to this work.

Authors' Contact Information: Kanghee Park, University of California San Diego, USA, kap022@ucsd.edu; Xuanyu Peng, University of California San Diego, USA, xup002@ucsd.edu; Loris D'Antoni, University of California San Diego, USA, ldantoni@ucsd.edu.

ACM 2475-1421/2025/4-ART114
https://doi.org/10.1145/3720470

de Vries and Koutavas [10] argued that this specification alone is "incomplete"; a sorting function would also meet this specification, even though it would not behave correctly as a shuffle function. A better specification would include an angelic property that states that the shuffle function can in fact produce all permutations of the input list—i.e., for any permutation of the input list, there exists *some* nondeterministic execution that can generate it.

The goal of this paper is to devise a unified logical framework for synthesizing provably sound angelic and demonic specifications for nondeterministic programs. Most approaches that automatically generate specifications from code [2, 16, 17, 42] rely on a finite set of dynamically generated input executions and cannot guarantee soundness even when the programs are deterministic. To our knowledge, SPYRO [34] is the only framework for synthesizing specifications that are provably sound, but it is fundamentally limited to deterministic programs. In this paper, we redesign the SPYRO framework to support nondeterminism, and both angelic and demonic reasoning.

Limitations of SPYRO. In SPYRO, the problem of synthesizing a specification is phrased as follows: Given (*i*) a *quantifier-free* query Ψ posed about a set of function definitions, and (*ii*) a domain-specific language $\mathcal{L}$ in which each extracted property is to be expressed (we call properties in the language $\mathcal{L}$-properties), the goal is to synthesize a conjunction $\bigwedge_i \varphi_i$ of logically incomparable $\mathcal{L}$-properties such that each φ_i is an *over-approximation* of Ψ and is a strongest in $\mathcal{L}$—i.e., there is no other $\mathcal{L}$-property that over-approximates Ψ and that strictly implies φ_i. For example, for a query $\Psi := (l_{out} = \texttt{reverse}(l_{in}))$ describing a list-reverse function, and a language $\mathcal{L}$ of arithmetic formulas over variables and their lengths, the property $\varphi := \texttt{len}(l_{out}) \leq \texttt{len}(l_{in})$ is an $\mathcal{L}$-consequence of Ψ, but not a strongest one, because $\varphi_1 := \texttt{len}(l_{out}) = \texttt{len}(l_{in})$ is a stronger $\mathcal{L}$-consequence of Ψ.

Because the query Ψ and DSL $\mathcal{L}$ can be provided by a user, the SPYRO framework can be applied to many domains. By setting $\mathcal{L}$ to capture the syntax of restricted refinement types, SPYRO has been used for extracting refinement types from data-structure transformations [37], whereas by setting $\mathcal{L}$ to capture algebraic specification, SPYRO could synthesize interfaces for software modules [35].

SPYRO is very expressive, but it does not support nondeterministic programs for two reasons. First, SPYRO's synthesis algorithms are fundamentally limited to *quantifier-free queries*. Without existential quantifiers, SPYRO's queries cannot model nondeterministic programs, concurrent programs, or uncertainty. Second, SPYRO is *limited to synthesizing over-approximations* of the program behavior. While over-approximations can capture what must happen for *every* (nondeterministic) execution (the demonic properties), reasoning about the actual behaviors that *some* (nondeterministic) execution can exhibit (the angelic properties) requires under-approximated specifications.

The LOUD Framework. This paper addresses the two limitations of SPYRO and presents LOUD, a general framework for solving the following problem:

> Given an **existentially quantified query** Ψ posed about a set of function definitions and a language $\mathcal{L}$, find (*i*) a **strongest conjunction** of $\mathcal{L}$-consequences that is implied by Ψ, and (*ii*) a **weakest disjunction** of $\mathcal{L}$-implicants that implies Ψ.

The LOUD framework is our key contribution. While LOUD superficially looks like a small modification of SPYRO [34], the minimal extension of including existential quantifications in queries allows LOUD to elegantly capture many new complex scenarios into a single unified logical framework. Specifically, existentially quantified queries allow LOUD to reason about both angelic and demonic properties of programs involving nondeterminism. For example, consider the dining-philosophers problem where *n* philosophers are arranged in a circle, each concurrently (and nondeterministically) acquiring and releasing contended resources placed on either of their sides. We can model what combinations of actions and scheduling lead to a deadlock using an existentially quantified query

such as $\exists s.\, dl = \texttt{philo}(s, p_1, \ldots, p_n)$, where $p_i \in \{L, R\}$ indicates which resource the philosopher p_i tries to take first and dl denotes that a deadlock has happened; s is the nondeterministic sequence of order in which threads are scheduled (detailed description in Section 7.3).

Supporting both over- and under-approximate reasoning (i.e., computing both $\mathcal{L}$-consequences and $\mathcal{L}$-implicants) enables new applications. Let's say we are interested in understanding what philosophers' actions *may* lead to a deadlock for *some* possible schedule. When given an appropriate language $\mathcal{L}$, a possible under-approximation (i.e., $\mathcal{L}$-implicant) of the query $\exists s.\, dl = \texttt{philo}(s, p_1, \ldots, p_n)$ is $dl \wedge p_1 = \cdots = p_n$, which states that deadlock can happen when all the philosophers prefer the same direction. On the other hand, if we instead are interested in what philosophers' choices *must* prevent deadlocks for *any* possible schedule, we can resort to over-approximation. A possible over-approximation (i.e. $\mathcal{L}$-consequence) in the LOUD framework is $p_1 \neq p_2 \Rightarrow \neg dl$, which states that deadlock will not happen when processes p_1 and p_2 disagree on their fork choice. The above two example properties show that, for nondeterministic programs, consequences hold for *every possible* nondeterministic choice (the demonic perspective), whereas implicants hold for *at least one* nondeterministic choice (the angelic perspective).

Thanks to its generality, LOUD can also capture reasoning capabilities of Hoare logic [23] (e.g., computing weakest liberal precondition and strongest postcondition) and incorrectness logic [10, 31] (e.g., computing weakest possible precondition and weakest under-approximate postcondition).

New Synthesis Algorithms in LOUD. Existentially quantified queries and the ability to synthesize both over- and under-approximations make the LOUD framework more expressive than SPYRO, but also make synthesis more challenging, thus requiring new algorithmic insights.

Park et al. [34] presented a counterexample-guided synthesis (CEGIS) algorithm for solving problems in the SPYRO framework. The algorithm accumulates positive and negative examples of possible program behaviors with respect to the given query and synthesizes $\mathcal{L}$-properties consistent with them. A primitive called CHECKSOUNDNESS checks if a candidate property is indeed sound and, if not, it produces a new positive example that the property fails to accept. To ensure $\mathcal{L}$-consequences are strongest, a primitive CHECKPRECISION checks if the current $\mathcal{L}$-consequence is strongest; if it is not, CHECKPRECISION returns a new $\mathcal{L}$-property that accepts all positive examples, rejects all negative examples, and rejects one more negative example (which is also returned). By alternating calls to these primitives, the algorithm eventually finds a strongest $\mathcal{L}$-consequence.

Key contributions and innovations in how we algorithmically solve LOUD problems include (*i*) generalized CHECKSOUNDNESS and CHECKPRECISION primitives so that each operation has a dual form that can be used to synthesize both $\mathcal{L}$-consequences and $\mathcal{L}$-implicants, and (*ii*) how we implement these primitives in the presence of existential quantifiers. Specifically, proving that an $\mathcal{L}$-consequence is strongest and proving that an $\mathcal{L}$-implicant is sound require solving a constraint with quantifier alternations of the form $\exists e.\, \forall h.\, \neg\psi(e, h) \wedge \varphi(e)$. To perform this check, we integrate a counterexample-guided quantifier instantiation algorithm (CEGQI) that operates in tandem with the overall CEGIS algorithm. The CEGIS algorithm accumulates examples that approximate the behavior of the query, while the CEGQI algorithm accumulates instances of the quantified variable h that show if an example is positive or negative. To our knowledge, our algorithm is the first one to combine CEGIS and CEGQI to deal with multiple nested quantifiers.

We implement a tool, called ASPIRE, to solve the synthesis problems in the LOUD framework. ASPIRE can describe and identify sources of bugs in both deterministic and nondeterministic programs, extract properties from concurrent programs, and synthesize winning strategies in two-player games. Because ASPIRE is built on the top of the SKETCH program synthesizer [44], it is only sound for programs in which inputs, recursion, and loops are bounded. In the future, this limitation can be lifted by considering more general (though less efficient) program synthesizers [26].

```
Variables {
    int a, M, y;
    exist int x;
}
Query {
    y = modhash(a, M, x);
}
```

(a) Query Ψ

```
Language {
    D -> \/[AP,0..6];
    AP -> I {<=|<|==|!=} I
        | isPrime(M)
        | !isPrime(M)
    I -> 0 | a | y | M | -M
}
```

(b) DSL $\mathcal{O}$

```
C1: 0 <= y
C2: y < M
C3: (a == 0) => (y == 0)
C4: (a == M) => (y == 0)
C5: (a == -M) => (y == 0)
```

(c) $\mathcal{O}$-consequences

Fig. 1. (a) A query Ψ for identifying properties of modhash that hold for any choice of input x. Users declare variables and label existentially quantified ones with the keyword **exist**. (b) A DSL $\mathcal{O}$ for over-approximations. \/[AP, 0..6] is a shorthand for the disjunction of 0 to 6 atomic propositions. (c) $\mathcal{O}$-consequences by our tool ASPIRE when given the query Ψ and the DSL $\mathcal{O}$. We write $p \Rightarrow q$ instead of $\neg p \vee q$ for readability.

Contributions. Our work makes the following contributions:

- A unified logical framework, LOUD, for the problem of synthesizing strongest $\mathcal{L}$-consequences and weakest $\mathcal{L}$-implicants for existentially quantified queries (§3).
- Algorithms for solving LOUD problems using four simple well-defined primitives: SYNTHESIZE, CHECKIMPLICATION, CHECKSTRONGEST and CHECKWEAKEST (§4).
- An algorithm that combines CEGQI and CEGIS to efficiently implement the primitives CHECKIMPLICATION and CHECKSTRONGEST for existentially quantified queries (§5).
- A tool that implements our framework, called ASPIRE (§6).
- Multiple instantiations of LOUD, showing its capability across a wide range of applications, e.g., reasoning about nondeterministic/concurrent programs and synthesizing game strategies (§7).

§8 discusses related work. §9 concludes. In the extended paper [36], §A relates LOUD to program logics; §B contains further details about algorithms; §C contains proofs; and §D contains further evaluation details.

2 Motivating Examples

In this section, we illustrate how the LOUD framework can be used to synthesize useful over-approximated (§2.1) and under-approximated (§2.2) properties of programs.

Consider the parametric hash function shown in Figure 2, where x is an integer input and a and $M > 0$ are possible parameters—i.e., a and M are fixed in a specific implementation of modhash. Intuitively, modhash can be viewed as a family of hash functions where the variable x is the input.

```
int modhash (int a, M, x) {
    return a * x mod M; }
```

Fig. 2. modhash function

In LOUD, to reason about the behavior of a program, one provides a logical query they are interested in over- or under-approximating with properties in a given language. For example, one may provide the query Ψ in Figure 1a, which is equivalent to the existentially quantified formula $\exists x.\ y = ax \bmod M$. In this example, our goal is to identify which choices of parameters a and M will make modhash surjective onto $\mathbb{Z}_M = \{0, 1, \ldots, M-1\}$. Specifically, we seek properties that capture the relationship between the output variable y and the parameters a and M (colored in red). To do so, we treat the input x as a nondeterministic parameter (existentially quantified and thus colored in blue).

2.1 Over-Approximate Reasoning

We start with properties that are consequences (over-approximations) of the query in Figure 1a. That is, we want formulae $\varphi(y, a, M)$ such that $\forall y, a, M.\ (\exists x.\ y = ax \bmod M) \Rightarrow \varphi(y, a, M)$.

As argued by Park et al. [34], different applications—e.g., generating type judgments or generating algebraic specifications—require formulae to adhere to a specific syntactic fragment. Thus in our framework, users are in charge of providing a DSL $\mathcal{L}$ as an input to express properties they are interested in; we call properties expressible in this DSL $\mathcal{L}$-properties. We note that the DSL $\mathcal{L}$ might contain user-given functions for which an implementation should be provided. Furthermore, we say an $\mathcal{L}$-property φ is an $\mathcal{L}$-consequence if it is a consequence of the given query formula. One goal of our framework is to synthesize a set of incomparable *strongest* $\mathcal{L}$-consequences.

In our example, the user provides the DSL $\mathcal{O}$ shown in Figure 1b, which includes the constant `0`, all free variables and comparison operations between them, and also the user-defined function `isPrime` (together with its implementation) that is potentially related to the problem.

An incomparable set of strongest $\mathcal{O}$-consequences for the query Ψ is shown in Figure 1c. Since conjoining two consequences of Ψ results in a stronger consequence of Ψ, we interpret the set of properties as their conjunction and thus call the set an $\mathcal{O}$-conjunction. The properties in Figure 1c give us insights into the behavior of the function `modhash`—e.g., that setting the value of a to be equal to M or $-M$ is probably not a good idea as it would result in a function that always returns $y = 0$. For our discussion, we focus our attention on the first two properties, which imply that the output y falls within $0 \leq y < M$. A well-designed hash function with a set S as range should be surjective onto the set S, meaning that for every value v in S, there should be inputs that yield v as output. However, because the formulae in Figure 1c are over-approximations, we are not guaranteed that all the values in $0 \leq y < M$ are indeed possible outputs of `modhash`.

2.2 Under-Approximate Reasoning

Over-approximation alone cannot capture whether a specific program behavior *can* occur—i.e., is reachable. For a formula $\varphi(y, a, M)$ to define a reachability condition (i.e., a behavior that *must* happen) of y from some input x, the formula $\varphi(y, a, M)$ must be an implicant (under-approximation) of the query Ψ, which formally can be stated as follows: $\forall y, a, M.\, \varphi(y, a, M) \Rightarrow \exists x.\, y = ax \bmod M$

We say an $\mathcal{L}$-property φ is an $\mathcal{L}$-implicant if it is an implicant of the given query formula. Another goal of our framework is to synthesize a set of incomparable *weakest* $\mathcal{L}$-implicants.

In our example, the user provides the DSL $\mathcal{U}$ using the rules for `AP` and `I` as shown in Figure 1b, but replaces disjunction rules (nonterminal `D`) with the conjunction rules `C -> /\[AP, 0..6]`. Throughout the paper we will use $\mathcal{O}$ to denote the language from Figure 1b in examples involving over-approximations, and $\mathcal{U}$ for the language described here in examples involving under-approximation. For instance, $a = 0 \wedge y = 0$ is a $\mathcal{U}$-implicant for query Ψ, but not a weakest one, as it strictly implies a $\mathcal{U}$-implicant $y = 0$.

A mutually incomparable set of weakest $\mathcal{U}$-implicants for the query Ψ is shown in Figure 3. We interpret the set of properties as their disjunction and refer to the set as a $\mathcal{U}$-disjunction. Each formula provides a sufficient condition for reachability of the output y—that is, if y, a and M satisfy any formula in Figure 3, then there exists an input x such that $y = ax \bmod M$. Crucially, the last formula provides a sufficient condition for `modhash` to be surjective onto $\mathbb{Z}_M = \{0, 1, \ldots, M-1\}$—i.e., for a prime value of M and non-zero value of a selected from the range $-M < a < M$, all values of y in $\mathbb{Z}_M$ are attainable from some choice of input x.

```
I1: y == 0
I2: 0 <= a /\ a < M /\ a == y
I3: 0 <= y /\ y < M /\ -M < a
    /\ a < M /\ a != 0 /\ isPrime(M)
```

Fig. 3. Synthesized $\mathcal{U}$-implicants.

While our primary motivation is to model nondeterminism, the generality of our framework enables a variety of other applications. In particular, existential quantifiers can be used to model both forward and backward reasoning in the style of Hoare and incorrectness program logics for both deterministic and nondeterministic programs. For example, given a program and a predicate

over its output, one can synthesize preconditions on what input may produce (over-approximate) or must produce (under-approximate) an output that satisfies the given predicate.

A detailed discussion of various applications of our framework is provided in Section 7 and Appendix A.

3 Strongest Consequences and Weakest Implicants

In this section, we describe our framework, which extends the strongest $\mathcal{L}$-consequences synthesis framework of Park et al. [34] in two key ways: (*i*) allowing existentially quantified variables, and (*ii*) enabling the synthesis of both the strongest $\mathcal{L}$-consequences and weakest $\mathcal{L}$-implicants.

We describe what inputs a user of the framework provides, and what they obtain as output.

Input 1: Query. The *query* Ψ is a first-order formula of the form $\exists h.\, \psi(v, h)$, where ψ is a quantifier-free formula ψ. The inclusion of the existentially quantified variables h in the query is a key novelty of this paper: it enables many new applications such as reasoning about nondeterministic programs, and forward and backward reasoning in program logics (Section 7.2).

We use the symbol h (for hidden) to represent existentially quantified variables and the symbol v (for visible) to represent free variables. In practice, both h and v can be tuples and denote multiple variables. In our motivating examples, queries are given in Figure 1a.

Input 2: Grammar of $\mathcal{L}$-properties. The grammar of the DSL $\mathcal{L}$ in which the synthesizer is to express properties for the query. Each formula φ in the DSL $\mathcal{L}$ is a predicate defined over the free variables v of the query $\exists h.\, \psi(v, h)$. An example of a DSL is the language $\mathcal{O}$ in Figure 1b.

Input 3: Semantics of the program and operators. Semantics of the function symbols that appear in query Ψ (e.g., `modhash`) and in the DSL $\mathcal{L}$ (e.g., `isPrime`). In our implementation, semantic definitions are given as a program in the SKETCH language [44], which automatically transforms them into first-order formulas. For example, one may provide a C-style function written in SKETCH that checks whether a number between 2 and $\sqrt{n}$ is a divisor of n as the semantic of `isPrime`. We discuss in Section 6 how SKETCH works and examine the limitations of this approach.

Output: Strongest $\mathcal{L}$-consequences and weakest $\mathcal{L}$-implicants. Our goal is to synthesize a set of incomparable *strongest* $\mathcal{L}$-consequences and a set of incomparable *weakest* $\mathcal{L}$-implicants of query Ψ. Ideally, both strongest $\mathcal{L}$-consequence and weakest $\mathcal{L}$-implicant would be the formula that is exactly equivalent to $\exists h.\, \psi(v, h)$, but in general, the DSL $\mathcal{L}$ might not be expressive enough to do so. As argued by Park et al. [34], this feature is actually a desired one as it allows for the application of our framework to various use cases, as demonstrated in Section 7. Because in general there might not be an $\mathcal{L}$-consequence and an $\mathcal{L}$-implicant that are equivalent to $\exists h.\, \psi(v, h)$, the goal becomes instead to find $\mathcal{L}$-properties that tightly approximate $\exists h.\, \psi(v, h)$.

We denote the set of models (over the free variables of Ψ) of a formula φ as $[\![\varphi]\!]$. For example in Section 2.1, $[\![y \geq 0]\!]$ represents the set of models $\{(y, a, M) \mid y \geq 0\}$. We say φ is stronger than φ' (or φ' is weaker than φ) when $[\![\varphi]\!] \subseteq [\![\varphi']\!]$, and φ is strictly stronger than φ' when $[\![\varphi]\!] \subset [\![\varphi']\!]$.

Definition 3.1 (A strongest $\mathcal{L}$-consequence). *An $\mathcal{L}$-property φ is* a strongest $\mathcal{L}$-consequence *for a query Ψ if and only if*

(i) *φ is a* consequence *of the query Ψ:*

$$\textit{IsCons}_\Psi(\varphi) := \forall v.\, [\exists h.\, \psi(v, h) \Rightarrow \varphi(v)]$$

(ii) *φ is* strongest *with respect to Ψ and $\mathcal{L}$:*

$$\neg \exists \varphi' \in \mathcal{L}.\, \textit{IsCons}_\Psi(\varphi') \wedge [\![\varphi']\!] \subset [\![\varphi]\!]$$

Definition 3.2 (A weakest $\mathcal{L}$-implicant). *An $\mathcal{L}$-property φ is* a weakest $\mathcal{L}$-implicant *for a query Ψ if and only if*

(i) *φ is an* implicant *of the query Ψ:*

$$\textit{IsImpl}_\Psi(\varphi) := \forall v.\, [\varphi(v) \Rightarrow \exists h.\, \psi(v, h)]$$

(ii) *φ is* weakest *with respect to Ψ and $\mathcal{L}$:*

$$\neg \exists \varphi' \in \mathcal{L}.\, \textit{IsImpl}_\Psi(\varphi') \wedge [\![\varphi']\!] \supset [\![\varphi]\!]$$

Throughout the paper, we also use the term *most-precise* to mean strongest for $\mathcal{L}$-consequences and weakest for $\mathcal{L}$-implicants. We use $SCons_{\mathcal{L}}(\Psi)$ and $WImpl_{\mathcal{L}}(\Psi)$ to denote the set of all strongest $\mathcal{L}$-consequences and the set of all best $\mathcal{L}$-implicants for Ψ, respectively. Because $\mathcal{L}$ may not be closed under conjunction (and disjunction), strongest $\mathcal{L}$-consequences (and weakest $\mathcal{L}$-implicants) may not be semantically unique. In Section 2.1, both formulae $0 \leq y$ and $y < M$ are strongest $\mathcal{L}$-consequences of query Ψ, and neither implies the other.

The goal of our framework is to find a semantically strongest conjunction of incomparable strongest $\mathcal{L}$-consequences and a weakest disjunction of incomparable weakest $\mathcal{L}$-implicants.

Definition 3.3 (Best $\mathcal{L}$-conjunction)**.** *A potentially infinite set of $\mathcal{L}$-consequences $\Pi = \{\varphi_i\}$ forms a* best $\mathcal{L}$-conjunction $\varphi_\wedge = \bigwedge_i \varphi_i$ *for query Ψ if and only if*
*(*i*) each $\varphi_i \in \Pi$ is a strongest $\mathcal{L}$-consequence of Ψ;*
*(*ii*) every distinct $\varphi_i, \varphi_j \in \Pi$ are* incomparable*—i.e., $[\![\varphi_i]\!] \setminus [\![\varphi_j]\!] \neq \emptyset$ and $[\![\varphi_j]\!] \setminus [\![\varphi_i]\!] \neq \emptyset$;*
*(*iii*) the set is* semantically minimal*—i.e., for every strongest $\mathcal{L}$-consequence φ we have $[\![\varphi_\wedge]\!] \subseteq [\![\varphi]\!]$.*

Definition 3.4 (Best $\mathcal{L}$-disjunction)**.** *A potentially infinite set of $\mathcal{L}$-implicants $\Pi = \{\varphi_i\}$ forms a* best $\mathcal{L}$-disjunction $\varphi_\vee = \bigvee_i \varphi_i$ *for query Ψ if and only if*
*(*i*) each $\varphi_i \in \Pi$ is a weakest $\mathcal{L}$-implicant of Ψ;*
*(*ii*) every distinct $\varphi_i, \varphi_j \in \Pi$ are* incomparable*—i.e., $[\![\varphi_i]\!] \setminus [\![\varphi_j]\!] \neq \emptyset$ and $[\![\varphi_j]\!] \setminus [\![\varphi_i]\!] \neq \emptyset$;*
*(*iii*) the set is* semantically maximal*—i.e., for every weakest $\mathcal{L}$-implicant φ we have $[\![\varphi_\vee]\!] \supseteq [\![\varphi]\!]$.*

Best $\mathcal{L}$-conjunctions and best $\mathcal{L}$-disjunctions are not necessarily unique, but they are all logically equivalent. Specifically, a best $\mathcal{L}$-conjunction is equivalent to the conjunction of all possible strongest $\mathcal{L}$-consequences, and a best $\mathcal{L}$-disjunction is equivalent to the disjunction of all possible weakest $\mathcal{L}$-implicants. Note that *best* means more than semantic optimality because a strongest $\mathcal{L}$-conjunction is not necessarily a best $\mathcal{L}$-conjunction; predicates $\varphi_1(x) := x \geq 0$ and $\varphi_2(x) := x \geq 1$ could form a strongest $\mathcal{L}$-conjunction, but it is not a best one because φ_1 is strictly stronger than φ_2–i.e., φ_1 and φ_2 are comparable.

Theorem 3.1 (Semantic Optimality). *If $\varphi_\wedge$ is a best $\mathcal{L}$-conjunction, then its interpretation coincides with the conjunction of all possible strongest $\mathcal{L}$-consequences: $[\![\varphi_\wedge]\!] = [\![\bigwedge_{\varphi \in SCons_{\mathcal{L}}(\Psi)} \varphi]\!]$. If $\varphi_\vee$ is a best $\mathcal{L}$-disjunction, then its interpretation coincides with the disjunction of all possible best $\mathcal{L}$-implicants: $[\![\varphi_\vee]\!] = [\![\bigvee_{\varphi \in WImpl_{\mathcal{L}}(\Psi)} \varphi]\!]$.*

We are now ready to state our problem definition:

Definition 3.5 (Best $\mathcal{L}$-conjunction and $\mathcal{L}$-disjunction Synthesis)**.** *Given query Ψ, the concrete semantics for the function symbols in Ψ, and a domain-specific language $\mathcal{L}$ with its corresponding semantic definition, synthesize a best $\mathcal{L}$-conjunction and $\mathcal{L}$-disjunction for Ψ.*

Practical remarks. The algorithm presented in Section 4 computes finite $\mathcal{L}$-conjunctions and $\mathcal{L}$-disjunctions, but it is possible that a solution to a given LOUD problem instance requires an infinite number of conjuncts (or disjuncts). Even in this case, because the algorithm incrementally computes incomparable strongest $\mathcal{L}$-consequences (or weakest $\mathcal{L}$-implicants), one can stop it at any point and all computed $\mathcal{L}$-consequences (or $\mathcal{L}$-implicants) will form a valid $\mathcal{L}$-conjunction (or $\mathcal{L}$-disjunction), just not a best one. The benchmarks considered in Section 7 do not encounter this problem since they consider finite (though very large) languages.

Note that Definition 3.3 and 3.4 only guarantee that each *pair* of synthesized properties is incomparable. However, ensuring that each φ_i is incomparable to the entire set of properties $\Pi \setminus \{\varphi_i\}$ only requires checking a single implication for each φ_i, which can be trivially done in a postprocessing phase. In practice, it is quite rare to encounter a redundant conjunct or disjunct.

4 Counterexample-Guided Inductive Specification Synthesis

In this section, we present algorithms for synthesizing best $\mathcal{L}$-conjunctions and best $\mathcal{L}$-disjunctions. We follow the example-guided approach proposed by Park et al. [34] that synthesizes strongest $\mathcal{L}$-consequences. In tandem, we present a dual algorithm to synthesize weakest $\mathcal{L}$-implicants, and we extend both algorithms to allow existentially quantified query formulas.

We first present the primitives necessary to instantiate the synthesis algorithms (Section 4.1). Then, we present the algorithms for synthesizing a single $\mathcal{L}$-consequence or $\mathcal{L}$-implicant that is incomparable to all the ones synthesized so far (Section 4.2). Finally, we present the algorithms for iteratively synthesizing the properties forming an $\mathcal{L}$-conjunction or an $\mathcal{L}$-disjunction (Section 4.3).

4.1 Synthesis from Positive and Negative Examples

The algorithms for synthesizing strongest $\mathcal{L}$-consequences and weakest $\mathcal{L}$-implicants maintain two sets of examples: a set of positive examples E^+, which should be accepted by the synthesized predicates, and negative examples E^-, which should be rejected by the synthesized predicates.

Definition 4.1 (Examples)**.** *Given a query $\Psi := \exists h.\, \psi(v, h)$ and a model e over the free variable v of query Ψ, we say that e is a* positive example *if $\psi(e, h)$ holds true for* some *value of h (i.e., $e \in [\![\Psi]\!]$) and a* negative example *if $\psi(e, h)$ does not hold for* all *values of h (i.e., $e \notin [\![\Psi]\!]$).*

Example 4.1 (Positive and Negative Examples). *Given the query $\Psi := \exists x.\, y = ax \bmod M$, the model that assigns y to the integer 1, a to the integer 6, and M to the integer 5 is a positive example, because the choice of value $x = 1$ makes the equation $1 = 6 \bmod 5$ holds true. For brevity, we represent such example as $(1, 6, 5)$, where it denotes a valuation to the tuple (y, a, M). The following examples are negative ones because no value of x satisfies $y = ax \bmod M$: $(-1, 1, 3)$, $(3, 1, 3)$, $(3, 2, 6)$.*

Because the DSL $\mathcal{L}$ might not be expressive enough to capture the exact behavior of the query Ψ, in general there is no predicate capable of accepting all the positive examples and rejecting all the negative examples. Intuitively, a strongest $\mathcal{L}$-consequence must accept all positive examples while also excluding as many negative examples as possible.

Example 4.2 (Examples and $\mathcal{L}$-consequences). *Consider again the query $\Psi := \exists x.\, y = ax \bmod M$ and the set of strongest O-consequences $\{0 \le y, y < M,\ a = 0 \implies y = 0,\ a = M \implies y = 0,\ a = -M \implies y = 0\}$ from Figure 1c. While a positive example $(y, a, M) = (1, 6, 5)$ is accepted by all O-consequences, the negative example $(3, 2, 6)$ is not rejected by any of them. In fact, the O-consequences in Figure 1c form a best $\mathcal{L}$-conjunction, so $(3, 2, 6)$ cannot be rejected by* any *strongest O-consequence.*

As illustrated by Example 4.2 when attempting to synthesize $\mathcal{L}$-consequences, we can consider positive examples as hard constraints but need to treat negative examples as soft constraints.

For $\mathcal{L}$-implicants, the role of positive and negative examples is inverted. A weakest $\mathcal{L}$-implicant must reject all negative examples while also accepting as many positive examples as possible.

Example 4.3 (Examples and $\mathcal{L}$-implicants). *Consider again the query $\Psi := \exists x.\, y = ax \bmod M$ and the set of weakest $\mathcal{U}$-implicants $\{y = 0,\ 0 \le a < M \wedge a = y,\ 0 \le y < M \wedge -M < a < M \wedge a \neq 0 \wedge \texttt{isPrime}(M)\}$ from Figure 3. While the negative example $(y, a, M) = (3, 2, 6)$ is rejected by all $\mathcal{U}$-implicants, the positive example $(1, 6, 5)$ is not accepted by any of them. The $\mathcal{U}$-implicants in Figure 3 form a best $\mathcal{L}$-disjunction, so $(1, 6, 5)$ must be rejected by* every *weakest $\mathcal{U}$-implicants.*

We are now ready to introduce the generalizations of the key operations used by Park et al. [34] to synthesize strongest $\mathcal{L}$-consequences: Synthesize (Section 4.1.1), CheckImplication (Section 4.1.2) and CheckStrongest (Section 4.1.3). Additionally, we introduce CheckWeakest (Section 4.1.3), an operation used alongside Synthesize and CheckImplication to synthesize weakest $\mathcal{L}$-implicants.

4.1.1 Synthesis from Examples. While strongest $\mathcal{L}$-consequences and weakest $\mathcal{L}$-implicants can effectively treat some of the examples as soft constraints, as we will show in Section 4.2, our synthesis algorithm can find such properties by iteratively calling a synthesis primitive Synthesize that treats a carefully chosen set of examples as hard constraints. Avoiding soft constraints was one of the key innovations of Park et al. [34] with respect to prior work [25].

Given a set of positive examples E^+ and a set of negative examples E^-, the procedure $\textsc{Synthesize}(E^+, E^-)$ returns an $\mathcal{L}$-property φ that accepts all the positive examples in E^+ and rejects all the negative examples in E^-, if such an $\mathcal{L}$-property φ exists. If no such $\mathcal{L}$-property exists, then $\textsc{Synthesize}(E^+, E^-)$ returns $\bot$. Given a set of examples E, we write $\varphi(E)$ to denote the conjunction $\bigwedge_{e \in E} \varphi(e)$ and $\neg\varphi(E)$ to denote the conjunction $\bigwedge_{e \in E} \neg\varphi(e)$. The operation $\textsc{Synthesize}(E^+, E^-)$ can be expressed as the formula $\exists\varphi.\, \varphi(E^+) \wedge \neg\varphi(E^-)$.

Example 4.4 (Synthesize). *Example 4.2 showed there can be a negative example that no $\mathcal{L}$-consequences can reject. With the DSL $\mathcal{O}$ defined in Figure 1b, if $E^+ = \{(1, 6, 5)\}$ and $E^- = \{(3, 2, 6)\}$, then* $\textsc{Synthesize}(E^+, E^-)$ *can return the formula $\varphi(y, a, M) := y < a$, which is not a consequence of the query $\Psi := \exists x.\, y = ax \bmod M$. In this case, once more positive examples are added to E^+ (which is something our synthesis algorithm automatically takes care of),* Synthesize *will return $\bot$. For example, if E^+ is augmented to the set $\{(1, 6, 5), (1, 1, 5), (1, -4, 5), (6, 2, 8)\}$, then* $\textsc{Synthesize}(E^+, E^-)$ *returns $\bot$—i.e., the negative example $(3, 2, 6)$ cannot be rejected by any $\mathcal{O}$-consequences and our synthesis algorithm will later remove it from E^-.*

4.1.2 Checking Implication. The CheckImplication primitive described in this section allows us to check whether a formula is valid $\mathcal{L}$-implicant or a valid $\mathcal{L}$-consequence.

Given two predicates φ and φ', the primitive $\textsc{CheckImplication}(\varphi, \varphi')$ checks whether φ is an implicant of φ' (or dually, whether φ' is consequence of φ). In logical terms, $\textsc{CheckImplication}(\varphi, \varphi')$ checks whether there does not exist an example e that is accepted by φ but rejected by φ'; it returns $\top$ or an example if the check fails. This check can be expressed as $\neg\exists e.\, \neg\varphi'(e) \wedge \varphi(e)$.

$\textsc{CheckImplication}(\Psi, \varphi)$ returns $\top$ if a predicate φ is a consequence of Ψ. Similarly, $\textsc{CheckImplication}(\varphi, \Psi)$ returns $\top$ if a predicate φ is an implicant of Ψ.

Example 4.5 (CheckImplication). *Consider again the query $\Psi := \exists x.\, y = ax \bmod M$. Because the formula $\varphi(y, a, M) := y < a$ is not a* consequence *of Ψ, the primitive* $\textsc{CheckImplication}(\Psi, \varphi)$ *would return a positive example that is rejected by φ, such as $(1, -4, 5)$. On the other hand, calling* $\textsc{CheckImplication}(\Psi, \varphi')$ *on the formula $\varphi'(y, a, M) := y \geq 0$ would instead return $\top$ because the formula φ' is indeed a* consequence *of Ψ.*

Similarly, for the formula $\varphi(y, a, M) := y < a$, which is also not a implicant *of Ψ, running* $\textsc{CheckImplication}(\varphi, \Psi)$ *(where this time the query Ψ is the second parameter) would return a negative example that is accepted by φ, such as $(-1, 1, 3)$. On the other hand, running* $\textsc{CheckImplication}(\varphi', \Psi)$ *on the formula $\varphi'(y, a, M) := y = 0$ would instead return $\top$ because the formula φ' is an* implicant *of Ψ.*

The CheckImplication procedure can be implemented using a constraint solver. However, the presence of quantifiers in implicants can result in a constraint with alternating quantifiers, making the check computationally harder, and most importantly, outside the capabilities of solvers that do not support quantifiers. We discuss a practical procedure for performing this check in Section 5.

4.1.3 Checking Precision. Checking precision—i.e., whether an $\mathcal{L}$-consequence is strongest or whether an $\mathcal{L}$-implicant is weakest—requires more sophisticated queries than the one described above. Specifically, one cannot simply ask whether there exists a negative example that is accepted by φ to check whether φ is a strongest $\mathcal{L}$-consequence, because, as shown in Example 4.2, there might be some negative example that must be accepted by every strongest $\mathcal{L}$-consequence.

In theory, to prove or disprove that an $\mathcal{L}$-consequence φ is strongest one needs to check whether there exists an $\mathcal{L}$-property φ' that is (*i*) a consequence of the query Ψ (i.e., φ' accepts all the positive examples) and (*ii*) strictly stronger than φ (i.e., φ' rejects at least one more negative example while rejecting all the negative examples that were already rejected by φ). Such a check would be too expensive as it effectively asks one to synthesize a provably sound $\mathcal{L}$-consequence.

Our algorithm does not require such a powerful primitive, and instead approximates (*i*) and (*ii*) using a set of positive examples E^+ accepted by φ and a set of negative examples E^- rejected by φ. By combining implication and precision checks in a counterexample-guided fashion, our algorithm improves the approximation over time and is thus sound.

Given an $\mathcal{L}$-consequence φ, a set of positive examples E^+ accepted by φ, a set of negative examples E^- rejected by φ, a query Ψ, and the $\mathcal{L}$-consequences $\varphi_\wedge$ we have already synthesized, CheckStrongest$(\varphi, \varphi_\wedge, \Psi, E^+, E^-)$ checks if there do not exist an $\mathcal{L}$-property φ' and an negative example $e \notin [\![\Psi]\!]$ satisfying $\varphi_\wedge$ such that: (*i*) φ' accepts all the positive examples in E^+; (*ii*) φ' rejects e and all the negative examples in E^-, whereas φ accepts e. In our algorithm, the formula $\varphi_\wedge$ is used to ensure that the example produced by CheckStrongest is not already rejected by best $\mathcal{L}$-consequences we have already synthesized. The above check can be logically stated as follows:

$$\textsc{CheckStrongest}(\varphi,\varphi_\wedge,\Psi,E^+,E^-) = \neg\exists\varphi', e.\neg\Psi(e) \wedge \varphi_\wedge(e) \wedge \varphi(e) \wedge \neg\varphi'(e) \wedge \varphi'(E^+) \wedge \neg\varphi'(E^-) \quad (1)$$

The highlighted part of the formula is what changes when checking if the formula is weakest.

Example 4.6 (CheckStrongest). *Consider again the query* $\Psi := \exists x.\, y = ax \bmod M$*, and an* $\mathcal{O}$*-consequence* $\varphi(y, a, M) := y \neq M$*, which is* not *a strongest one.*

CheckStrongest$(\varphi, \top, \Psi, \{(1,6,5)\}, \{(3,1,3)\})$ *can return a strictly stronger* $\mathcal{O}$*-consequence* $\varphi'_1(y, a, M) := y < M$ *with a negative example* $e_1^- = (6,1,5)$*.*

However, because CheckStrongest *only considers whether the formula* φ *is strongest with respect to the examples* $\{(1,6,5)\}, \{(3,1,3)\}$*, it can also alternatively return a property that is not an actual* $\mathcal{O}$*-consequence—e.g.,* $\varphi'_2(y, a, M) := y < a$ *with a negative example* $e_2^- = (6,1,5)$*. The formula* φ'_2 *is not a* $\mathcal{O}$*-consequence of* Ψ *as it rejects the positive example* $(1,-4,5)$*.*

When checking if an $\mathcal{L}$-implicant φ is a weakest one, we can perform a dual check and ask if there does not exist an $\mathcal{L}$-property that can accept one more example than the current formula. That is, CheckWeakest$(\varphi, \varphi_\vee, \Psi, E^+, E^-)$ checks whether there do not exist an $\mathcal{L}$-property φ' and a positive example $e \in [\![\Psi]\!]$ satisfying $\varphi_\vee$ such that: (*i*) φ' accepts all the positive examples in E^+; (*ii*) φ' accepts e and all the positive examples in E^+, whereas φ rejects e. This check can be logically stated as

$$\textsc{CheckWeakest}(\varphi,\varphi_\vee,\Psi,E^+,E^-) = \neg\exists\varphi', e.\Psi(e) \wedge \neg\varphi_\vee(e) \wedge \neg\varphi(e) \wedge \varphi'(e) \wedge \varphi'(E^+) \wedge \neg\varphi'(E^-) \quad (2)$$

Example 4.7 (CheckWeakest). *Consider again the query* $\Psi := \exists x.\, y = ax \bmod M$*, and a* $\mathcal{U}$*-implicant* $\varphi(y, a, M) := y = 0 \wedge a = 0$*, which is* not *a weakest* $\mathcal{U}$*-implicant.* CheckWeakest $(\varphi, \bot, \Psi, \{(0,0,5)\}, \{(3,2,6)\})$ *can return a strictly weaker* $\mathcal{U}$*-implicant* $\varphi'_1(y, a, M) := y = 0$ *with a positive example* $e_1^+ = (0,1,5)$*. However, for the same reasons outlined in Example 4.6, the returned property may not be a* $\mathcal{U}$*-implicant.*

The CheckStrongest and CheckWeakest procedures are effectively solving a synthesis problem—i.e., they are looking for a formula—and implementing them requires a form of example-based synthesis. Again, the presence of quantifiers in the negation of the query $\neg\Psi$ for CheckStrongest can result in constraint (1) containing alternating quantifiers, thus bringing us outside of the capability of many program synthesizers. We discuss a practical procedure for sidestepping the quantifier-alternation problem and performing this check in Section 5.

Algorithm 1:
SynthStrongestConsequence$(\Psi, \varphi_\wedge, \varphi_{init}, E^+, E^-)$

```
1  φ, φ_last ← φ_init; E⁻_may ← ∅
2  while true do
3      e⁺ ← CheckImplication(Ψ, φ)
4      if e⁺ ≠ ⊤ then
5          E⁺ ← E⁺ ∪ {e⁺}
6          φ′ ← Synthesize(E⁺, E⁻ ∪ E⁻_may)
7          if φ′ ≠ ⊥ then
8              φ ← φ′
9          else
10             φ ← φ_last;   E⁻_may ← ∅
11     else
12         E⁻ ← E⁻ ∪ E⁻_may;   E⁻_may ← ∅
13         φ_last ← φ
14         e⁻, φ′ ← CheckStrongest(φ, φ_∧, Ψ, E⁺, E⁻)
15         if e⁻ ≠ ⊤ then
16             E⁻_may ← {e⁻}
17             φ ← φ′
18         else
19             return φ, E⁺, E⁻
```

Algorithm 2:
SynthWeakestImplicant $(\Psi, \varphi_\vee, \varphi_{init}, E^+, E^-)$

```
1  φ, φ_last ← φ_init; E⁺_may ← ∅
2  while true do
3      e⁻ ← CheckImplication(φ, Ψ)
4      if e⁻ ≠ ⊤ then
5          E⁻ ← E⁻ ∪ {e⁻}
6          φ′ ← Synthesize(E⁺ ∪ E⁺_may, E⁻)
7          if φ′ ≠ ⊥ then
8              φ ← φ′
9          else
10             φ ← φ_last;   E⁺_may ← ∅
11     else
12         E⁺ ← E⁺ ∪ E⁺_may;   E⁺_may ← ∅
13         φ_last ← φ
14         e⁺, φ′ ← CheckWeakest(φ, φ_∨, Ψ, E⁺, E⁻)
15         if e⁺ ≠ ⊤ then
16             E⁺_may ← {e⁺}
17             φ ← φ′
18         else
19             return φ, E⁺, E⁻
```

4.2 Synthesizing One Strongest $\mathcal{L}$-Consequence and One Weakest $\mathcal{L}$-Implicant

We are now ready to describe our main procedures: SynthStrongestConsequence (Algorithm 1) and SynthWeakestImplicant (Algorithm 2). We first recall the description of SynthStrongestConsequence by Park et al. [34], the algorithm that synthesizes a strongest $\mathcal{L}$-consequence that is incomparable with the $\mathcal{L}$-consequences we already synthesized (the algorithm will be used in Section 4.3 to synthesize one $\mathcal{L}$-consequence at a time). We then describe one of the contributions of this paper, i.e., how the algorithm changes for its under-approximated dual SynthWeakestImplicant.

4.2.1 Synthesizing One Strongest $\mathcal{L}$-Consequence. Given a query formula Ψ and a conjunction of $\mathcal{L}$-consequences we have already synthesized $\varphi_\wedge$, the procedure SynthStrongestConsequence synthesizes a strongest $\mathcal{L}$-consequence φ for the query Ψ that is incomparable to the already synthesized formulas in $\varphi_\wedge$. We say an $\mathcal{L}$-consequence φ for the query Ψ is strongest *with respect to* $\varphi_\wedge$ if there does not exist an $\mathcal{L}$-consequence φ' for Ψ such that $\varphi' \wedge \varphi_\wedge$ is strictly stronger than $\varphi \wedge \varphi_\wedge$—i.e., the $\mathcal{L}$-consequence φ is incomparable to all the $\mathcal{L}$-consequences in $\varphi_\wedge$.

In each iteration, SynthStrongestConsequence performs two steps. First, it uses CheckImplication to check whether the current candidate φ is a consequence of Ψ (line 3). Second, if the candidate φ is a consequence of Ψ, it uses CheckStrongest to check whether φ is strongest with respect to $\varphi_\wedge$ (line 14). The algorithm terminates once a formula passes both checks (line 19).

If the current candidate φ is not a consequence of Ψ, CheckImplication returns a positive example e^+ (line 3). The algorithm then adds e^+ to the set of positive examples E^+ and uses it to Synthesize a new candidate $\mathcal{L}$-property (lines 5 and 6).

If the current candidate φ is a consequence of Ψ but there is an $\mathcal{L}$-property φ' that aligns with the current set of positive and negative examples, E^+ and E^-, and can reject one more negative example e^-, CheckStrongest returns this property φ' along with the negative example e^-. The example e^- is then temporarily stored in E^-_{may} without immediately updating E^- (line 16). Updating E^- is delayed because φ' may not be a consequence of the query Ψ (Example 4.6), and in the worst

case, there might not exist an $\mathcal{L}$-consequence that rejects e^- (Example 4.2). The example stored in E^-_{may} can be safely added to E^- when CheckImplication verifies that $\mathcal{L}$-property φ returned by CheckStrongest is indeed a consequence of the query Ψ (line 12); at this point we are certain that the example in E^-_{may} can be rejected by at least one $\mathcal{L}$-consequence, as witnessed by φ.[1]

The candidate $\mathcal{L}$-property returned by CheckStrongest in line 14 is only guaranteed to be consistent with the examples, and therefore must be checked again by CheckImplication in line 3. If the candidate fails to pass CheckImplication, the algorithm keeps adding more positive examples until either (*i*) it finds a $\mathcal{L}$-consequence that rejects all the negative examples in $E^- \cup E^-_{may}$, or (*ii*) Synthesize in line 6 fails to find an $\mathcal{L}$-property. In the latter case, SynthStrongestConsequence concludes that the example in E^-_{may} cannot be rejected by any $\mathcal{L}$-consequence and thus restarts after discarding the example in E^-_{may} (line 10). For efficiency, whenever E^- is updated in line 12, the algorithm stores the current $\mathcal{L}$-consequence φ that rejects all the negative examples in E^- in a variable φ_{last} (line 13). In this way, the algorithm can revert to φ_{last} when E^-_{may} is discarded (line 10).

Example 4.8 (Algorithm 1 Run). *Consider the query* $\Psi := \exists x.\, y = ax \bmod M$. *The table below shows the last 4 iterations in a possible execution of* SynthStrongestConsequence $(\Psi, \top, \top, \emptyset, \emptyset)$. *Specifically, it shows the value of* φ, E^+, *and* E^- *at the start of each iteration (Line 2). In the table,* $e^+_1 = (y, a, M) = (1, 6, 5), e^+_2 = (1, 1, 5), e^+_3 = (1, -4, 5), e^+_4 = (6, 2, 8)$, $e^-_1 = (8, 1, 8)$, $e^-_2 = (3, 2, 6)$, $e^-_3 = (6, 2, 5)$.

Iteration $n-3$. *So far the algorithm has computed an* $\mathcal{O}$*-consequence* $y \neq M$ *that is not a strongest one. Therefore,* CheckImplication *passes but* CheckStrongest *fails, and the execution reaches Line 16. Then* φ *is set to a new candidiate* `isPrime`(M), *and a negative example* e^-_2 *is added to* E^-_{may}.

Iteration $n-2$. *The property* `isPrime`(M) *is not an* $\mathcal{O}$*-consequence, so* CheckImplication *fails. The execution reaches Line 5, and a positive example* e^+_4 *is added to* E^+. *As discussed in Example 4.4,* e^-_2 *cannot be rejected by any* $\mathcal{O}$*-consequence, so* Synthesize *fails,* φ *is reverted to* $y \neq M$, *and* E^-_{may} *is cleared.*

Iteration $n-1$. *Similar to iteration* $n-3$ *but with a new positive example* e^+_4 *added,* CheckStrongest *returns a new candidiate* $y < M$ *and a new negative example* e^-_3, *which is added to* E^-_{may}.

Iteration n. *The property* $y < M$ *is indeed a strongest* $\mathcal{O}$*-consequence, so it passes both the* CheckImplication *and* CheckStrongest *and is finally returned.*

Iter.	φ	E^+	E^-	E^-_{may}
$n-3$	$y \neq M$	$\{e^+_1, e^+_2, e^+_3\}$	$\{e^-_1\}$	$\emptyset$
$n-2$	`isPrime`(M)	$\{e^+_1, e^+_2, e^+_3\}$	$\{e^-_1\}$	$\{e^-_2\}$
$n-1$	$y \neq M$	$\{e^+_1, e^+_2, e^+_3, e^+_4\}$	$\{e^-_1\}$	$\emptyset$
n	$y < M$	$\{e^+_1, e^+_2, e^+_3, e^+_4\}$	$\{e^-_1\}$	$\{e^-_3\}$

Once SynthStrongestConsequence terminates, it returns a strongest $\mathcal{L}$-consequence for Ψ (with respect to $\varphi_\wedge$) that accepts all the examples in E^+ and rejects all the examples in E^-. Informally, if CheckImplication returns $\top$, then the property is an $\mathcal{L}$-consequence, and if CheckStrongest returns $\top$, then the property is a strongest property.

SynthStrongestConjunction is also guaranteed to terminate for a finite DSL, when every call to primitives Synthesize, CheckImplication and CheckStrongest terminates. Informally, CheckImplication at Line 3 can only return a counterexample (i.e., φ is not a consequence) finitely many times, and CheckStrongest at Line 14 can only return a counterexample (i.e., φ is a consequence but not a strongest one) finitely many times between successive instances where CheckImplication returns $\top$. The full proof is in Appendix C.

[1] Delaying the update of negative examples enables treating all negative examples in E^- as hard constraints. This is a key innovation by Park et al. [34] over prior work [25].

4.2.2 Synthesizing One Weakest $\mathcal{L}$-Implicant. Given a query formula Ψ, a disjunction of $\mathcal{L}$-implicants we already synthesized $\varphi_\vee$, the goal of the procedure SynthWeakestImplicant is to synthesize a strongest $\mathcal{L}$-implicant φ for the query Ψ that is incomparable to the already synthesized formulas in $\varphi_\vee$.

We say an $\mathcal{L}$-implicant φ for the query Ψ is weakest *with respect to* $\varphi_\vee$ if there does not exist an $\mathcal{L}$-implicant φ' for Ψ such that $\varphi' \vee \varphi_\vee$ is strictly weaker than $\varphi \vee \varphi_\vee$—i.e., the $\mathcal{L}$-implicant φ is incomparable to the already synthesized $\mathcal{L}$-implicants.

SynthWeakestImplicant solves the dual of the problem solved by SynthStrongestConsequence, and the two algorithms share the same structure. Due to the duality (*i*) the roles of positive and negative examples are inverted; (*ii*) CheckImplication in line 3 checks whether φ is an *implicant* of Ψ, instead of checking that φ is a *consequence* of Ψ; and (*iii*) precision is checked by CheckWeakest instead of CheckStrongest. These changes are highlighted in violet in Algorithm 2.

4.3 Synthesizing a Best $\mathcal{L}$-Conjunction and $\mathcal{L}$-Disjunction

We conclude by briefly recalling how SynthStrongestConjunction works (as described by Park et al. [34]) and present the dual algorithm SynthWeakestDisjunction. These two algorithms use SynthStrongestConsequence and SynthWeakestImplicant to synthesize a best $\mathcal{L}$-conjunction and $\mathcal{L}$-disjunction, respectively. The detailed algorithms are illustrated in Appendix B.

The algorithm SynthStrongestConjunction iteratively synthesizes incomparable strongest $\mathcal{L}$-consequences. At each iteration, SynthStrongestConjunction keeps track of the conjunction of synthesized strongest $\mathcal{L}$-consequences $\varphi_\wedge$, and calls SynthStrongestConsequence to synthesize a strongest $\mathcal{L}$-consequence for Ψ with respect to $\varphi_\wedge$.

If SynthStrongestConsequence returns an $\mathcal{L}$-consequence φ that does not reject any example that was not already rejected by $\varphi_\wedge$, the formula $\varphi_\wedge$ is a best $\mathcal{L}$-conjunction, and thus SynthStrongestConjunction returns the set of synthesized $\mathcal{L}$-consequences.

If SynthStrongestConsequence returns an $\mathcal{L}$-consequence φ that rejects some example that was not rejected by $\varphi_\wedge$, SynthStrongestConjunction needs to further strengthen φ to a strongest $\mathcal{L}$-consequence for Ψ with respect to examples that might already be rejected by $\varphi_\wedge$. Without this step the returned $\mathcal{L}$-consequence may be imprecise for examples that were not considered by SynthStrongestConsequence because they were outside of $\varphi_\wedge$. To achieve this further strengthening, SynthStrongestConjunction makes another call to SynthStrongestConsequence with the example sets E^+ and E^- returned by the previous call to SynthStrongestConsequence together with φ and $\varphi_{init} := \varphi$, but with $\varphi_\wedge := \top$.

Again, because SynthWeakestDisjunction solves the dual problem of the one solved by SynthStrongestConjunction, the two algorithms share the same structure. SynthWeakestDisjunction uses SynthWeakestImplicant in a similar manner, but it maintains the disjunction of synthesized weakest $\mathcal{L}$-implicants instead of conjunction. For weakening, it also makes another call to SynthWeakestImplicant, but $\varphi_\wedge := \top$ is replaced by $\varphi_\vee := \bot$.

Using the argument of Park et al. [34] for SynthStrongestConjunction, assuming all the primitives always terminate, our algorithms satisfy the following soundness and completeness theorems, Specifically, the number of iterations in Algorithm 1 and 2 is bounded by the number of properties in the DSL $\mathcal{L}$ and the number of examples in the domain, whichever is smaller.

Theorem 4.1 (Soundness). *If* SynthStrongestConjunction *terminates, $\varphi_\wedge$ is a best $\mathcal{L}$-conjunction for Ψ. If* SynthWeakestDisjunction *terminates, $\varphi_\vee$ is a best $\mathcal{L}$-disjunction for Ψ.*

Theorem 4.2 (Relative Completeness). *Suppose that either $\mathcal{L}$ contains finitely many formulas, or the example domain is finite.*

If Synthesize, CheckImplication *and* CheckStrongest *are decidable on* $\mathcal{L}$*, then* SynthStrongestConsequence *and* SynthStrongestConjunction *always terminate.*

If Synthesize, CheckImplication *and* CheckWeakest *are decidable on* $\mathcal{L}$*, then* SynthWeakestImplicant *and* SynthWeakestDisjunction *always terminate.*

Part of the proof is stated in the end of Section 4.2.1. The full proof is in Appendix C.

Also, our procedure for solving problems expressed in the loud framework iteratively synthesizes strongest $\mathcal{L}$-consequences (resp. $\mathcal{L}$-implicants) that can strengthen (resp. weaken) the current $\mathcal{L}$-conjunction (resp. $\mathcal{L}$-disjunction) until no further strengthening (resp. weakening) is possible. Thanks to this iterative approach, even when the procedure does not terminate, one can output intermediate results, which are properties that are guaranteed to be strongest (resp. weakest), though they might not form a best $\mathcal{L}$-conjunction (resp. $\mathcal{L}$-disjunction).

5 Counterexample-Guided Quantifier Instantiation

In Section 4, when we discussed the main specification-synthesis loop, we assumed we were given implementations of all the needed primitives. In this section, we explain how each primitive can be implemented for existentially quantified queries.

CheckStrongest (line 14) in SynthStrongestConsequence and CheckImplication (line 3) in SynthWeakestImplicant check for the existence of a new negative example (along with additional constraints). However, when dealing with an existentially quantified query $\Psi := \exists h.\, \psi(v, h)$, a negative example e must be such that the formula $\neg\psi(e, h)$ is valid for *all* values of the existentially quantified variable h. Therefore, checking the existence of a negative example e requires solving a formula that has alternating quantifiers. To handle these primitives involving quantifier alternation, we propose a CounterExample-Guided Quantifier Instantiation (CEGQI) algorithm similar to the one by Reynolds et al. [40], which can implement the primitives that require finding negative counterexamples using only existentially-quantified formulas.

Counterexample-Guided Quantifier Instantiation for Weakest $\mathcal{L}$-implicant. We start with the simpler of the two queries, CheckImplication in SynthWeakestImplicant (line 3), which requires solving a formula with alternating quantifiers of the following form (by negating formulas in §4.1.2):

$$\exists e.\, \forall h.\, \neg\psi(e, h) \wedge \varphi(e) \tag{3}$$

The CEGQI algorithm for solving Equation (3) iteratively builds a set H of possible values for h and finds a value of e that is consistent with the finite set of values H. The set H is updated by repeating the following two operations until a solution that holds for all values of h is found.

Generating Candidate Negative Example. Given formulae φ, ψ, and a finite set H of values the existentially-quantified variable h can take, GenCandidateNegEx(φ, ψ, H) generates an example $e \in [\![\varphi]\!]$ such that the formula $\psi(e, h)$ does not hold for all the values h in the set H, if such an example e exists. If no such example exists, GenCandidateNegEx(φ, ψ, H) returns $\bot$. Formally:

$$\text{GenCandidateNegEx}(\varphi, \psi, H) = \exists e.\ \textstyle\bigwedge_{h \in H} \neg\psi(e, h) \wedge \varphi(e) \tag{4}$$

Checking Candidate Negative Example. Given a formula ψ, and a candidate negative example e, the function CheckCandidateNegEx(ψ, e) checks if there does not exist a value for the existentially quantified variable h such that $\psi(e, h)$ holds true (i.e., whether there exists a value of h that makes the example actually positive); it returns $\top$ or the value of h if the check fails. Formally:

$$\text{CheckCandidateNegEx}(\psi, e) = \neg\exists h.\, \psi(e, h) \tag{5}$$

Counterexample-Guided Quantifier Instantiation. The CEGQI algorithm for CheckImplication (Algorithm 3) iteratively generates candidate negative examples using GenCandidateNegEx and

Algorithm 3: CheckImplication(φ, Ψ)

```
1  assume Ψ = ∃h. ψ(v, h)
2  H ← ∅
3  while ⊤ do
4      e ← GenCandidateNegEx(φ, ψ, H)
5      if e = ⊥ then
6          return ⊤
7      else
8          h ← CheckCandidateNegEx(ψ, e)
9          if h = ⊤ then
10             return e
11         else
12             H ← H ∪ {h}
```

Algorithm 4: CheckStrongest$(\varphi, \varphi_\wedge, \Psi, E^+, E^-)$

```
1  assume Ψ = ∃h. ψ(v, h)
2  H ← ∅
3  while ⊤ do
4      e, φ′ ← GenCandidateSpecNegEx (φ, ψ, φ_∧, E⁺, E⁻, H)
5      if e = ⊥ then
6          return ⊤
7      else
8          h ← CheckCandidateNegEx(ψ, e)
9          if h = ⊤ then
10             return e
11         else
12             H ← H ∪ {h}
```

checks whether they are actually negative using CheckCandidateNegEx. Across iterations, it maintains the set of values of h returned by CheckCandidateNegEx in H, and uses GenCandidateNegEx to find an example e that behaves well for all the values in H discovered so far—i.e., e satisfies $\neg\psi(e, H) \wedge \varphi(e)$ (line 4).

If GenCandidateNegEx fails to find an example, it means that there is no example e satisfying $\neg\psi(e, H) \wedge \varphi(e)$, thereby a stronger condition in Equation (3) also cannot be satisfied. Therefore, CEGQI returns $\top$ (line 6)—i.e., there does not exist a valid negative example.

If GenCandidateNegEx returns an example e, the example is tested by CheckCandidateNegEx to check whether $\psi(e, h)$ does not hold for every possible value of h and not only for values found so far (line 8). The algorithm returns the example e once it passes the check (line 10), but if it fails the check, a new counterexample h returned by CheckCandidateNegEx is added to the set H, and the algorithm restarts at line 4. Note that the set of instances H can be cached and reused across different calls to CEGQI.

Example 5.1 (Algorithm 3 Run). *Consider again the query* $\Psi := \exists x.\, y = ax \bmod M$ *and the formula* $\varphi := 0 \le y < M$. *The table on the right shows a possible execution of how* CheckImplication$(\varphi, \Psi) := \forall e.\, \varphi(e) \Rightarrow \exists h.\, \psi(e, h)$ *can find a negative example and prove that* φ *is not a* $\mathcal{U}$*-implicant. Specifically, it shows the values of* $e = (y, a, M)$ *and* $h = x$ *at the end of each iteration (Line 12) within Algorithm 3. For each iteration,* $\varphi(e)$ *is always true for* $e = (y, a, M)$ *, while* $\psi(e, h)$ *is false for all previous* $h = x$ *but true for* h *in the current iteration.*

	e			h
Iter.	y	a	M	x
1	2	2	4	1
2	3	1	4	3
3	0	2	4	2
4	0	3	4	0
5	3	2	4	$\top$

Counterexample-Guided Quantifier Instantiation for Strongest $\mathcal{L}$-consequence. The call to CheckStrongest in SynthStrongestConsequence (line 14) requires solving a formula that has alternating quantifiers and the following form (by negating appropriate formulas in §4.1.3):

$$\exists e, \varphi'.\, \forall h.\, \neg\psi(e, h) \wedge \varphi_\wedge(e) \wedge \varphi(e) \wedge \neg\varphi'(e) \wedge \varphi'(E^+) \wedge \neg\varphi'(E^-) \tag{6}$$

This formula looks more complicated due to the presence of the existential variable φ'. However, a similar CEGQI approach to the one presented in Section 5 can also be used to solve Equation (6), by finding a negative example and a formula in tandem.

The only change in the CEGQI algorithm for solving CheckStrongest (Algorithm 4) is that GenCandidateNegEx is replaced by a new operation, GenCandidateSpecNegEx, defined as

follows. Given formulae $\varphi, \psi, \varphi_\wedge$, set of examples E^+, E^-, and a finite set H of values the existentially-quantified variable h can take, GenCandidateSpecNegEx$(\varphi, \psi, \varphi_\wedge, E^+, E^-, H)$ generates an $\mathcal{L}$-property φ' and an example e satisfying $\varphi_\wedge$ such that (*i*) the formula $\psi(e, h)$ does not hold for all the values h in the set H (*ii*) φ' accepts all the positive examples in E^+; (*iii*) φ' rejects e and all the negative examples in E^-, whereas φ accepts e, if such an example e and formula φ' exist. If no such example exists, then GenCandidateSpecNegEx$(\varphi, \psi, \varphi_\wedge, E^+, E^-, H)$ returns $\bot$. Stated formally:

$$\text{GenCandidateSpecNegEx}(\varphi, \psi, \varphi_\wedge, E^+, E^-, H) = \\ \exists e, \varphi'.\ \textstyle\bigwedge_{h \in H} \neg\psi(e, h) \wedge \varphi_\wedge(e) \wedge \varphi(e) \wedge \neg\varphi'(e) \wedge \varphi'(E^+) \wedge \neg\varphi'(E^-) \quad (7)$$

Because the variable h only appears in the constraint $\neg\psi(e, h)$, whether e is indeed a negative example can still be tested using CheckCandidateNegEx (5).

Similar to the Algorithm 3, if GenCandidateNegEx fails to find an example, it means that there is no example e satisfying Equation (7), thereby a stronger condition in Equation (6) also cannot be satisfied. The example is only returned after it has been tested by CheckCandidateNegEx to ensure that $\psi(e, h)$ does not hold for every possible value of h.

Correctness. The above observations are summarized as the following soundness theorem.

Theorem 5.1 (Soundness of CEGQI). *(*i*) If* CheckImplication *terminates with an example e, the example e is a valid solution to the existential quantifier in Equation (3). If* CheckImplication *terminates with $\bot$, there is no example e that satisfies Equation (3). (*ii*) If* CheckStrongest *terminates with an example e and φ', the example e and φ' are valid solution to the existential quantifier in Equation (6). If* CheckStrongest *terminates with $\bot$, there is no example e and φ' that satisfy Equation (6).*

Because Algorithm 3 and 4 monotonically increases the size of the set H, as long as the domain of one of the variables e and h is finite, both algorithms always terminate.

Theorem 5.2 (Completeness of CEGQI). *Suppose at least one of the domains of the variables e or h is finite. If* GenCandidateNegEx *and* CheckCandidateNegEx *are decidable for ψ and φ, then* CheckImplication *always terminates. If* GenCandidateSpecNegEx *and* CheckCandidateNegEx *are decidable for ψ and φ, then* CheckStrongest *always terminates.*

Therefore, when the domain is finite, the specification synthesis for an existentially quantified query can be solved using only calls with the quantifier free part of the query.

Note that, in the worst case, CEGQI can enumerate the entire domain of h. As we demonstrate in our evaluation, this exhaustive enumeration (which is common for CEG-style algorithms [43]) is practically rare and a small number of examples are usually sufficient to solve the problem.

6 Implementation

We implemented our algorithms for solving synthesis problems in the loud framework in a tool called aspire. aspire is implemented in Java, on top of the sketch program synthesizer (v.1.7.6) [44].

Following Section 3, aspire takes the following four inputs: (*i*) A query Ψ for which aspire is to find $\mathcal{L}$-consequences or $\mathcal{L}$-implicants where each variable in Ψ should be labeled either as free or existentially quantified. (*ii*) The context-free grammar of the DSL $\mathcal{L}$ in which properties are to be expressed. (*iii*) A piece of code in the sketch programming language that expresses the concrete semantics of the function symbols in Ψ and $\mathcal{L}$. (*iv*) The bounded domain of each variable in the query Ψ—i.e., each variable is assigned a range of possible input values.

Take the motivating problem in Section 2.1 as an example: input (*i*) corresponds to the blocks **`Variables`** and **`Query`** in Figure 1a, input (*ii*) corresponds to the block **`Language`** in Figure 1b, and

input (*iii*) are SKETCH implementations of `modhash` and `isPrime`—i.e., simple imperative functions. Input (*iv*) is provided via the **`Examples`** block in Figure 4, which we will discuss next.

From Grammars to SKETCH Generators. As synthesis needs to be performed over properties in the DSL $\mathcal{L}$, the context-free grammar for $\mathcal{L}$ is automatically translated to a SKETCH generator. A SKETCH generator is a construct that allows one to describe a recursively defined search space of programs. In a generator, one is allowed to use holes (denoted with ??) to allow the synthesizer to make choices about what terms to output. In our setting, holes are used to select which production is expanded at each node in a recursively defined derivation tree.

ASPIRE also uses grammars, which in turn are translated to generators, to specify the values each variable in the query Ψ can assume. Figure 4 shows how the user specifies the bounded domain of each variable for the problem in Section 2.1. The nonterminal `IG` is translated to a generator that can produce an integer from $[-15, 15]$ (the notation `??(4)` denotes a 4-bit hole), and is used to define the domain for variables `a`, `y`, and `x`. Similarly, the nonterminal `PosIG` defines the domain of the positive integer variable `M` to be the range $[1, 16]$. ASPIRE also supports inductive datatypes, e.g. a generator of lists of integers in the range $[-15, 15]$ can be defined as `[ list LG -> nil() | cons(IG, LG) ]`.

```
Examples {
    int IG -> ??(4) | -??(4)
        for a, y, x;
    int PosIG -> ??(4) + 1;
        for M;
}
```

Fig. 4. The bounded domains of variables in problem Section 2.1

Synthesis Primitives in SKETCH. The primitives SYNTHESIZE, CHECKIMPLICATION in Algorithm 1, and CHECKWEAKEST in Algorithm 2 are implemented as calls to the SKETCH synthesizer. Typically, a SKETCH program contains 3 elements: (*i*) a harness procedure that defines what should be synthesized, (*ii*) holes associated with a corresponding generator, and (*iii*) assertions. The harness procedure is the entry point of the SKETCH program, and together with the assertion it serves as a specification for what values the holes can assume to form a valid solution to the synthesis problem. Multiple harnesses in one SKETCH program are also allowed, where the goal of the SKETCH synthesizer is to find the same assignment to shared holes that make all assertions pass. For example, when encoding SYNTHESIZE, each example is implemented as a harness with assertions indicating that it should be positive or negative. Both CHECKSTRONGEST in SYNTHSTRONGESTCONSEQUENCE (Algorithm 1) and CHECKIMPLICATION in SYNTHWEAKESTIMPLICANT (Algorithm 2) are implemented using the CEGQI approach described in Section 5 (Algorithms 4 and 3, respectively). These algorithms are implemented as separate procedures where each call to GENCANDIDATENEGEX and CHECKCANDIDATENEGEX only has an existential quantifier, and can thus be implemented as a single call to the SKETCH synthesizer.

Bounds. SKETCH allows one to provide bounds for recursion and loops to make synthesis tractable. In ASPIRE, we need to consider two kinds of bounds.

First, one has to bound the depth of each recursive generator. Concretely, this bound means that ASPIRE can only support DSLs where the derivation trees have bounded height (ASPIRE allows one to specify the bound for each DSL). As recursive generators are used to produce examples for inductive datatypes—e.g. list—one also has to bound the height of such examples. Second, one has to bound how many times a loop can be unrolled/executed. We will discuss in Section 7 what benchmarks are in theory affected by these bounds. Additionally, these bounds limit DSLs and example domains to a finite size, ensuring that our algorithms in theory terminate (Theorem 4.2).

Timeout. We use a timeout of 20 minutes, after which ASPIRE returns the current $\mathcal{L}$-consequences (or $\mathcal{L}$-implicants). Although it might not be the strongest $\mathcal{L}$-consequences (or weakest $\mathcal{L}$-implicants), each individual $\mathcal{L}$-consequence (or $\mathcal{L}$-implicant) is a strongest (or a weakest) one.

7 Evaluation

Our evaluation of ASPIRE consists of two parts. The first part consists of simple deterministic and nondeterministic programs (from [3, 8, 34]) for which we could manually inspect whether ASPIRE computed the correct properties (§7.1). The second part showcases the capabilities of ASPIRE and the flexibility of the LOUD framework through three case studies: forward/backward reasoning for incorrectness logic (§7.2), synthesizing properties of concurrent programs (§7.3), and solving two-player games (§7.4). The evaluation highlights that the programmable queries and customized DSLs in the LOUD framework enable it to express a wide range of problems. All our case studies involve underapproximation and/or existential quantifiers. For each case study, we describe how we model the problem in LOUD, how we collected the benchmarks, how we designed the DSLs, and we present an analysis of the running time and effectiveness of ASPIRE and of the quality of synthesized $\mathcal{L}$-consequences and $\mathcal{L}$-implicants.

We ran all experiments on an Apple M1 8-core CPU with 8GB RAM. Each benchmark was run 3 times (timeout 20 minutes), with different random seeds for the SKETCH solver. [2] All results in this section are for the median of three runs (by synthesis time).

7.1 Testing ASPIRE with Simple Programs

We conducted two experiments on simple programs for which we could manually check whether the synthesized properties were sound and strongest/weakest.

We provide a brief description of each experiment, but details about each problem, running times, concrete DSLs, and synthesized properties are discussed in Appendix D.1 (Test set I) and Appendix D.2 (Test set II).

Test Set I: Mining Under-Approximation Specifications. We used ASPIRE to compute $\mathcal{U}$-implicants for the 35 programs for which Park et al. [34] computed $\mathcal{O}$-consequences using SPYRO. These programs include integer functions, data structure manipulations, and small imperative programs. To get the dual DSL $\mathcal{U}$ of each DSL $\mathcal{O}$ used in the original benchmarks, we replaced every top-level production of the form $S \to \vee[AP, 0..n]$ with a production $S \to \wedge[AP, 0..n]$. Note that the queries for all the SPYRO benchmarks *do not* contain existential quantifiers.

ASPIRE could synthesize properties for 35/35 benchmarks, and guaranteed that all of them were best $\mathcal{U}$-implicants (i.e., CHECKWEAKEST succeeded). Each benchmark takes at most 7 minutes (the largest grammar contained $1.48 \cdot 10^{13}$ properties). Beause the DSL was originally designed for overapproximations, the synthesized underapproximation properties were often trivial. For example, for the list reverse function, ASPIRE only tells us any singleton list can be obtained as output when providing the same one as input, but provides no properties describing the behavior for lists of lengths greater than 1. While the synthesized properties were not informative, this simple benchmark set allowed us to test ASPIRE's ability to synthesize $\mathcal{U}$-implicants.

Test set II: Nondeterministic programs. To test ASPIRE on problems involving existential quantifiers, we designed 15 simple imperative programs where nondeterministic values serve as operands or array indices—e.g., a nondeterministic sorting algorithm. To model the sequence of nondeterministic choices taken by a program, we use an existentially quantified array of values h in the query. Whenever the program execution reaches a non-deterministic command, the command takes the next value of the array h.

[2] Our synthesis primitives are nondeterministic—i.e., SKETCH can return *any* possible valid counterexample or property. The different random seeds will result in the SKETCH solver selecting different such examples/formulas. ASPIRE can therefore generate different sets of properties with different seeds, but as stated in Theorem 3.1, all best $\mathcal{L}$-conjunctions (or $\mathcal{L}$-disjunctions) are semantically equivalent.

ASPIRE synthesizes both $\mathcal{O}$-consequences and $\mathcal{U}$-implicants for 14/15 benchmarks, taking less than 400 seconds for each benchmark and guaranteeing that the synthesized properties are best ones. All benchmarks that terminated for one run terminated for all 3 runs, with fastest and slowest runs differing by at most 2x. The synthesized properties provide intuition for both the demonic and angelic behaviors of these programs. For example, consider a program that nondeterministically swaps elements of an array of length 4 to sort it: ASPIRE can tell us which initial arrays *may* be sorted within n swaps (for a given n), and also identifies what kind of arrays will never be sorted in less than n swaps. The one timeout benchmark models a merge sort that computes the number of inverse pairs in an array of nondeterministic values; ASPIRE fails due to the complexity of the program, which involves nested recursions and loops.

7.2 Application 1: Incorrectness Reasoning

Thanks to the support for both over- and under-approximation, some forms of forward/backward reasoning for both Hoare logic [23] and incorrectness logic [31] can be captured in the LOUD framework. Because there has been a lot of research and there are many tools on precondition/post-condition inference of Hoare triples, we only discuss the relation between the LOUD framework and incorrectness logic in this subsection, along with an evaluation. A complete formalization of the relation between the LOUD framework and Hoare/incorrectness logic can be found in Appendix A.

7.2.1 Relation to Incorrectness Logic. An *incorrectness triple* $[P]\ s\ [Q]$ consists of a presumption P, a statement s, and a result Q, and it has the following meaning: every final state satisfying Q is reachable by executing program s starting from *some* state that satisfies presumption P:

$$\forall \sigma'.\, Q(\sigma') \Rightarrow \exists \sigma.\, [P(\sigma) \wedge [\![s]\!](\sigma, \sigma')] \tag{8}$$

Forward Reasoning: Weakest Under-approximate Postcondition. Given a program s and a presumption P, the *weakest under-approximate postcondition* $wupo(s, P)$ is the weakest predicate Q such that the triple $[P]\ s\ [Q]$ holds. We use $wupo_{\mathcal{L}}(s, P)$ to denote the weakest under-approximation post-condition expressible as a *disjunction* of predicates in the DSL $\mathcal{L}$. From Equation (8), $wupo_{\mathcal{L}}(s, P)$ can be obtained by synthesizing weakest $\mathcal{L}$-implicants for the query $\exists \sigma.\, P(\sigma) \wedge [\![s]\!](\sigma, \sigma')$.

Let's say we are interested in reasoning about the possible behaviors of the (incorrect) implementation of a modular hash function `remhash` shown in Figure 5, where `%` is the remainder operator (instead of the modulus). The `%` operator is often misused when implementing a modular operation, as $a \,\%\, b$ may yield a negative output when either a or b is negative.

```
int remhash (int a, M, x) {
    return a * x % M;
}
```

Fig. 5. `remhash` function

A summary of the behaviors of `remhash` can be identified by under-approximating the query $\Psi_{rem} := (\exists x.\, y = ax \,\%\, M)$. From the perspective of incorrectness logic, under-approximating Ψ_{rem} corresponds to performing forward reasoning to find results for y when no presumption on x is given—i.e., the presumption $P(x)$ is $\top$. For capturing under-approximations of the query Ψ_{rem}, we reuse the DSL $\mathcal{U}$ from the example in Section 2.2. A mutually incomparable set of weakest $\mathcal{U}$-implicants for query Ψ_{rem} is shown in Figure 6, which shows that `remhash` can indeed yield negative values for some choice of parameters a and M, as evidenced by the occurrence of a state in both the second and last formulae where y is negative. In other words, we recognize that some choices of input value can result in incorrect outcomes—i.e., negative numbers—but we do not know which ones. As we will show next, the inputs that lead to incorrect behaviors can be identified by backward reasoning.

```
I1: y == 0
I2: -M <= a /\ a < M /\ a == y
I3: -M <= y /\ y < M
    /\ -M < a /\ a < M
    /\ a != 0 /\ isPrime(M)
```

Fig. 6. Synthesized $\mathcal{U}$-implicants

```
Variables {
    int a, M, x;
    exist int y;
}
Query {
    y = remhash(a, M, x);
    y < 0;
}
```

(a) Ψ_{wpp}

```
Language {
    C -> /\[AP,0..6];
    AP -> I {<=|<|==|!=} I
        | isPrime(M)
        | !isPrime(M)
    I -> 0 | a | x | M | -M
}
```

(b) $\mathcal{U}_{wpp}$

```
I1: -M < x /\ x < 0
    /\ 0 < a /\ a < M
    /\ isPrime(M)
I2: 0 < x /\ x < M
    /\ -M < a /\ a < 0
    /\ isPrime(M)
```

(c) $\mathcal{U}_{wpp}$-implicants

Fig. 7. (a) A query Ψ_{wpp} that allows identifying *weakest possible preconditions* that cause a bug in the `remhash` function (i.e., outputing a negative number). (b) A DSL $\mathcal{U}_{wpp}$ for expressing weakest possible preconditions. (c) $\mathcal{U}_{wpp}$-implicants synthesized by ASPIRE.

Backward Reasoning: Weakest Possible Precondition. Surprisingly, backward predicate transformers for incorrectness logic do not always exist because valid presumptions may not exist. For example, there is no predicate P making the triple $[P]\ y = ax \bmod M\ [y = -1]$ true because no values of a, M and x satisfy $ax \bmod M = -1$. To address this shortcoming O'Hearn [31] suggests using the weakest possible precondition $wpp(s, Q)$, which is termed by Hoare [24] as "possible correctness". Intuitively, $wpp(s, Q)$ captures the set of initial states from which it is *possible* to execute s and terminate in a state that satisfies Q. When considering the `remhash` function from Figure 5, if Q encodes that the output is negative, $wpp(s, Q)$ will show which input possibly leads to a negative output. Formally, $wpp(s, Q)$ is the weakest P satisfying

$$\forall\sigma.\, P(\sigma) \Rightarrow [\exists\sigma'.\, Q(\sigma') \wedge [\![s]\!](\sigma, \sigma')] \tag{9}$$

Note that $P = wpp(s, Q)$ forms neither a Hoare nor an incorrectness triple with the program s and the postcondition Q. As proposed by O'Hearn, we can use $P = wpp(s, Q)$ to compute a new postcondition $Q' = wupo(s, P)$ and obtain a valid incorrectness triple $[P]\ s\ [Q']$.

We use $wpp_{\mathcal{L}}(s, Q)$ to denote the weakest possible precondition expressible as a *disjunction* of predicates in the DSL $\mathcal{L}$. From Equation (9), $wpp_{\mathcal{L}}(s, Q)$ can be obtained by synthesizing weakest $\mathcal{L}$-implicants of the query $\exists\sigma'.\, Q(\sigma') \wedge [\![s]\!](\sigma, \sigma')$.

For the `remhash` function, we have shown the existence of a bug through forward reasoning—i.e., the output y can be negative. Now we want to compute the weakest possible precondition (expressible in a DSL) that leads to the error state $Q(y) := y < 0$. Looking at Equation (9), we can spell out that a weakest possible precondition of $Q(y)$ for `remhash` is a weakest implicant of the formula $\exists y.\, [y = ax \,\%\, M \wedge Q(y)]$; one can provide to LOUD the corresponding query Ψ_{wpp} in Figure 7a.

To capture implicants of the query Ψ_{wpp}, we define the DSL $\mathcal{U}_{wpp}$ in Figure 7b by substituting every occurrence of y in $\mathcal{U}$ with x. An incomparable set of weakest $\mathcal{U}_{wpp}$-implicants for query Ψ_{wpp} is shown in Figure 7c, where each formula states sufficient conditions under which $ax \,\%\, M$ produces a negative output—i.e., when either a or x falls within the interval $(-M, 0)$ and the other falls within the interval $(0, M)$.

7.2.2 Evaluation on Incorrectness Reasoning. We collected a total of 14 benchmarks: (*i*) the 2 example problems `remwupo` and `remwpp` from Section 7.2.1. (*ii*) 6 simple illustrative examples from the incorrectness logic paper [31], and (*iii*) 6 more complicated problems we crafted to illustrate how ASPIRE's handling of incorrectness reasoning differs from incorrectness logic. Among these benchmarks, 7 are about $\mathcal{L}$-weakest under-approximate postcondition and the other 7 are about $\mathcal{L}$-weakest possible precondition. It takes ASPIRE less than 4 seconds to solve each benchmark

from [31] and less than 50 seconds to solve each benchmark we crafted. Evaluation details are shown in Table 1.

Analysis of Benchmarks from [31]. We collected all the 3 triples $[P]s[Q]$ from the examples used in [31] where s is a nondeterministic program. However, in these triples, there was no guarantee that Q was the weakest underapproximate postcondition of P, or P was the weakest possible precondition of Q. We used ASPIRE to synthesize $wupo_{\mathcal{L}}(s, P)$ and $wpp_{\mathcal{L}}(s, Q)$ (the DSL $\mathcal{L}$ contained the same primitives appearing in the examples in [31]), thus 3+3=6 benchmarks. We examined that each of 3 synthesized $wupo_{\mathcal{L}}(s, P)$ by ASPIRE was indeed a subset of $wupo(s, P)$, and for 2 cases the two were equal. For the one that is not equal, $wupo(s, P)$ is the set of all perfect squares numbers, whereas $wupo_{\mathcal{L}}(s, P)$ is the perfect squares numbers lower than a bound (this difference was due to our query limiting the sample space of each variable). The 3 $wpp_{\mathcal{L}}(s, Q)$ synthesized by ASPIRE are equal to $wpp(s, Q)$.

More Complex Benchmarks. The 6 more complex benchmarks for which we performed incorrectness are `arit1-wupo`, `arit1-wpp`, `arit2-wupo`, `arit2-wpp`, `hashcoll`, and `coin`.

The benchmarks `arit1` and `arit2` model two arithmetic functions "$x' = \texttt{ite}(h_0, x, -x)$" and "$x' = (h_1 + 1) \cdot x + h_2$", where each $h_i \in \{0, 1\}$ is a nondeterministic value. For both cases, we set $a \le x \le b$ as a precondition P (or $a \le x' \le b$ as a postcondition Q) to synthesize $wpp_{\mathcal{L}}(s, Q)$ (or $wupo_{\mathcal{L}}(s, P)$), and thus get 4 benchmarks in total. To use ASPIRE, we need to mark x as existentially quantified variables when synthesizing $wupo_{\mathcal{L}}(s, P)$, whereas mark x' as existentially quantified variables when synthesizing $wpp_{\mathcal{L}}(s, P)$. Given a DSL containing basic arithmetic and comparison operators, ASPIRE synthesizes $wpp_{\mathcal{L}}(s, Q)$ and $wupo_{\mathcal{L}}(s, P)$ that are equal to $wpp(s, Q)$ and $wupo(s, P)$.

For example, to synthesize $wpp_{\mathcal{L}}(s, Q)$ for `arit1`, one can construct a query "$\exists x', h_0.\ x' = \texttt{ite}(h_0, x, -x) \wedge a \le x' \le b$" and supply the DSL in Figure 8. ASPIRE will synthesize the $\mathcal{L}$-implicants $\{-b \le x, x \le -a, a \le x, x \le b\}$.

We briefly summarize the findings on other benchmarks. The `coin` benchmark models the values one can produce using two coins that have co-prime denominations; ASPIRE can identify a lower bound above which all possible values can be produced using these coins. The `hashcoll` benchmark models a parametric hash function; ASPIRE can synthesize the condition that possibly causes a hash collision. More details are discussed in Appendix D.3.

Table 1. Applications 1 to 3. LoC is the number of lines of SKETCH code used to write the semantics of programs and operators. $|\exists|$ is the size of the domain of the existentially quantified variables. #P and T(s) are the number of properties and synthesis time for both $\mathcal{L}$-consequences and $\mathcal{L}$-implicants. Incorrectness reasoning does not require synthesizing $\mathcal{L}$-consequences.

	Problem	LoC	$\lvert\exists\rvert$	$\mathcal{L}$-cons.		$\mathcal{L}$-impl.	
				#P	T(s)	#P	T(s)
Incorrectness	remwupo	10	32	/	/	3	27.05
	remwpp	14	32	/	/	2	26.27
	inc1wupo	13	256	/	/	3	1.10
	inc2wupo	28	64	/	/	9	2.15
	inc3wupo	16	2	/	/	2	0.41
	inc1wpp	12	256	/	/	2	0.42
	inc2wpp	33	4,096	/	/	7	6.01
	inc3wpp	21	2	/	/	1	0.15
	arit1wupo	8	32	/	/	2	4.03
	arit2wupo	8	64	/	/	3	2.96
	arit1wpp	8	4	/	/	2	3.76
	arit2wpp	8	8	/	/	5	62.52
	hashcoll	87	16	/	/	3	15.94
	coin	23	1,024	/	/	1	20.33
Concurrency	philo	79	$\sim 10^6$	4	11.85	3	6.15
	race1	82	64	1	0.71	3	0.85
	race2	86	1,024	1	2.08	3	2.06
	race3	88	4,096	1	18.79	5	4.69
	rsrc1	81	16	4	2.73	4	1.67
	rsrc2	85	256	4	5.08	4	5.37
	rsrc3	114	$\sim 10^6$	4	51.83	4	38.31
	rsrc4	96	$\sim 10^5$	6	145.61	6	81.41
	obdet	52	1,057	3	2.11	/	/
Game	num1	47	8	10	16.13	19	13.99
	num2	47	32	25	21.32	15	8.54
	rg	29	32	2	0.38	2	0.49
	nim2	59	$\sim 10^{22}$	2	99.23	4	22.19
	temp	34	120	10	125.24	10	74.65

```
C -> /\[AP,0..5];
AP -> N {<=|<|==|!=} N
N ->  N' | -N'
N' -> 0 | a | b | x
```

Fig. 8. The DSL for `arit1`

```
O -> /\[AP,0..5] => D
U -> /\[AP,0..5] /\ D
AP -> I == 'L' | I == 'R'
I -> o1 | ... | oN
D -> dl | !dl
```

(a) The DSLs $\mathcal{O}$ (rooted at O) and $\mathcal{U}$ (rooted at U)

```
C1: (o1 == L /\ o2 == R)
    => !dl
C2: (o2 == L /\ o3 == R)
    => !dl
C3: (o3 == L /\ o1 == R)
    => !dl
```

(b) Synthesized $\mathcal{O}$-consequences

```
I1: o1 == L /\ o2 == L
    /\ o3 == L /\ dl
I2: o1 == R /\ o2 == R
    /\ o3 == R /\ dl
I3: !dl
```

(c) Synthesized $\mathcal{U}$-implicants

Fig. 9. The DSLs and synthesized properties for the philo benchmark.

7.3 Application 2: Reasoning about Concurrent Programs

We show how ASPIRE can be used to reason about bugs in concurrent programs by considering 8 variants of 3 concurrency problems by Arpaci-Dusseau and Arpaci-Dusseau [1] (2 problems related to deadlocks, and 1 to race conditions), and one benchmark obdet that requires synthesizing hyperproperties. Similar to how we model nondeterminism, we introduce an array h to represent the order in which threads are scheduled.

In the philo benchmark, we show how ASPIRE can synthesize conditions under which deadlock can be reached or avoided for the dining-philosophers problem, where N processes arranged in a circle contend N resources that are shared by neighboring processes. A deadlock happens when no process can access both of their Left and Right resources indefinitely. ASPIRE models this problem with a query $\exists h.\ dl = schedule(o_1, \cdots, o_N, h)$, where $o_i \in \{L, R\}$ indicates which resource the process i always takes first; dl denotes that a deadlock has happened.

For the case involving three processes/philosophers ($N = 3$), when given a DSL $\mathcal{O}$ in Figure 9a that contained predicates of the form $o_i = \{L|R\}$, ASPIRE synthesizes the $\mathcal{O}$-consequences in Figure 9b, which state that deadlock can be prevented by having two of the processes disagree on their fork choice. For the same N, and a dual DSL $\mathcal{U}$ in Figure 9a, ASPIRE synthesizes the $\mathcal{U}$-implicants in Figure 9c, which exactly characterize the two cases that lead to a deadlock (first two properties) and also capture that there always exists an execution that does not lead to a deadlock (last property).

Whereas similar tools for concurrent programs only deal with properties over a single schedule [28, 47], the next benchmark obdet demonstrates how ASPIRE can also synthesize properties that involve multiple schedules (i.e., hyperproperties). The obdet benchmark models two threads `p_o := p_i + s | p_o := p_i - s` where s is a secret variable and p_o and p_i are public variables. We say that the system is observational deterministic [48] if the observations made by a public observer (i.e., one that can only observe p_o and p_i) are deterministic, regardless of scheduling orders and secret input data (i.e., the values of the variables s). The query in the obdet benchmark is "$\exists h_1, h_2.\ p_{o1} = schedule(p_{i1}, s_1, h_1) \wedge p_{o2} = schedule(p_{i2}, s_2, h_2)$" where h_1 and h_2 describe 2 different schedules. Given a DSL $\mathcal{L}$ containing only public variables, ASPIRE synthesizes the $\mathcal{L}$-consequence $p_{i1} = p_{i2} \Rightarrow p_{o1} = p_{o2}$, ensuring the system is observationally deterministic.

Each of the 4 rsrc benchmarks describes a simple resource allocator; ASPIRE synthesizes properties describing the minimum number of resources that must (or may) cause a deadlock. Each of the 3 race benchmarks describes two threads; ASPIRE can discover the necessary (or sufficient) ways to place a critical section to prevent race conditions. Details are shown in Appendix D.4.

7.4 Application 3: Solving Two-Player Games

In this section, we show how ASPIRE can be used to synthesize generalized strategies for solving two-player games. In particular, by carefully designing the DSL, ASPIRE can synthesize sets of winning strategies expressed in a symbolic form, rather than a single concrete strategy.

We illustrate the idea using an example by Bloem et al. [5], called rg (for request/grant). The two players take on the roles of client and server, and in each round, the server decides whether to grant (g) or not ($\bar{g}$) the request for that round, and then the client decides whether to send (r) or not ($\bar{r}$) a request in that round. To win the game, the server must grant every request in the same or next round. Bloem et al. [5] show the server player can be in 3 possible states: (*i*) q_0: no ungranted request (*ii*) q_1: an ungranted request in the last round (*iii*) q_2: ungranted requests 2 or more rounds ago. The server should prevent entering state q_2.

One of the winning strategies from the server side is to always grant on both state q_0 and q_1. We denote such a strategy as $\alpha[q_0] = g \wedge \alpha[q_1] = g$—i.e., $\alpha[q] = a$ denotes that strategy α chooses action a when in state q. We can find winning strategies by modeling the rg game as a query "$\exists\beta.\ w = \text{play}(\alpha, \beta)$", where the client's strategy β is existentially quantified and $\text{play}(\alpha, \beta)$ is the game controller that takes the strategy of both players and produces a Boolean value w denoting whether the server wins after playing the game. Note that the way we model 2-player games can also be extended to multi-player games by introducing a set of opponent strategies $\{\beta_1, \cdots, \beta_k\}$.

The generality of the LOUD framework allows ASPIRE to solve two-player games using the following queries: (*i*) *Must-win strategy*: what strategy α can guarantee a win for *any* strategy β (Equation (10))?

$$\forall\alpha, w.\ \big(\exists\beta.\ w = \text{play}(\alpha, \beta)\big) \Rightarrow \big(P_{must}(\alpha) \Rightarrow w = T\big) \tag{10}$$

and (*ii*) *May-win strategy*: what strategy α can win for *at least one* strategy β (Equation (11))?

$$\forall\alpha, w.\ \big(P_{may}(\alpha) \wedge w = T\big) \Rightarrow \big(\exists\beta.\ w = \text{play}(\alpha, \beta)\big) \tag{11}$$

Looking at Equation (10), if we provide a DSL $\mathcal{O}$ that expresses formulas in the form "$P_{must}(\alpha) \Rightarrow w = T$", we can extract the must-win strategy in the P_{must} part of the synthesized formulas. By replacing T with F we can get the must-lose strategy. For the rg game, ASPIRE synthesized the following $\mathcal{O}$-consequences, which tells us that the server will always win if they grant requests in either of states q_0 and q_1—i.e., ASPIRE finds "a set of" winning strategies.

$$\alpha[q_0] = g \Rightarrow w = T \qquad \alpha[q_1] = g \Rightarrow w = T \tag{12}$$

When provided with the dual DSL $\mathcal{U}$ ASPIRE also synthesized the following $\mathcal{U}$-implicants:

$$\alpha[q_0] = \bar{g} \wedge \alpha[q_1] = \bar{g} \wedge w = F \qquad w = T \tag{13}$$

The first $\mathcal{U}$-implicant states that the server may lose if they do not grant requests at both states q_0 and q_1, whereas the second $\mathcal{U}$-implicant states that whatever strategy the server uses there exists a strategy of the requester (i.e., the one that never issues requests) that causes the server to win.

Other benchmarks. We consider a total of 5 benchmarks: rg (discussed above), nim2 (the Nim game), and temp (a temperature controller), which are adapted from linear reachability games by Farzan and Kincaid [18][3], and num1 and num2, which are games designed by us in which two players manipulate an integer where one player's goal is to keep the integer in a certain range. Because of the implementation bounds discussed in section 6, we stipulate that player 1 (typically the player that needs to stay in safe states) wins, after a finite number (we set as 15) of rounds of play.

It takes ASPIRE less than 85 seconds to synthesize must/may strategies for each benchmark. Compared to the work by Farzan and Kincaid [18], ASPIRE synthesizes (*i*) not only *must*- but also *may*-strategies, (*ii*) properties on desired strategies instead of a concrete strategy, and (*iii*) general strategies that work for games with parameters (e.g. the initial number of pebbles in nim2).

Details of DSL design and synthesized properties of benchmarks are provided in Appendix D.5.

[3]All other games studied by Farzan and Kincaid cannot be modeled in ASPIRE due to the restricted features of SKETCH languages, such as limited support to floating point numbers.

8 Related Work

Abstract Interpretation. Many static program-analysis and verification techniques represent large program state spaces symbolically as predicates, using *abstract interpretation* [9]. While the majority of works on abstract interpretation has been focused on over-approximation, it can also be used to describe under-approximations of the program behavior [6, 21, 41]. In particular, the best $\mathcal{L}$-conjunction synthesis problem is an instance of *strongest-consequence problem* [39]. Given a formula φ in logic $\mathcal{L}_1$ (with interpretation $[\![\cdot]\!]_1$), the goal of strongest-consequence problem is to determine the strongest formula ψ that is expressible in a different logic $\mathcal{L}_2$ (with interpretation $[\![\cdot]\!]_2$) such that $[\![\varphi]\!]_1 \subseteq [\![\psi]\!]_2$. One existing technique to solve this problem identifies a chain of weaker implicants until one becomes a consequence of φ [38], whereas other techniques take the opposite direction, identifying a chain of stronger consequences [45, 46]. Unlike these approaches, our framework LOUD (like SPYRO [34]) supports a customizable DSL, avoiding requirements to perform operations on elements within the DSL $\mathcal{L}$, such as join [46]. The ability to modify the DSL is what makes the LOUD framework applicable to many domains.

Best $\mathcal{L}$-term Synthesis. The idea of synthesizing a "best" term from a customizable DSL $\mathcal{L}$ was first proposed by Kalita et al. [25], where the goal was to synthesize a most-precise abstract transformer for a given abstract domain. Park et al. [34] generalized the idea and introduced the setting to define and solve the problem of synthesizing best $\mathcal{L}$-conjunctions. In these work, the "best" term should be (*i*) *sound*: it is a valid approximation to the best transformer in Kalita et al. [25] or the semantics of query in Park et al. [34], and (*ii*) *precise*: it is minimal w.r.t. a preorder defined on $\mathcal{L}$.

The LOUD framework takes a step further: it further generalizes the queries to allow existential quantifiers and introduces the problem of synthesizing weakest $\mathcal{L}$-implicants and best $\mathcal{L}$-disjunctions. Logically, the LOUD framework subsumes both SPYRO and the work by Kalita et al..

At the algorithmic level, the tools solving the above problems all use two kinds of examples for synthesis, where one is treated as hard constraints to guarantee soundness and the other one is treated as soft constraints to guarantee precision. SPYRO improved the algorithm by Kalita et al. by introducing the idea of freezing examples, thus avoiding the need for a synthesizer with hard and soft constraints. The CEGQI algorithm we present Section 5 is a new approach that is not present in the aforementioned works as none of them supports existential quantifiers in their queries.

Program Logic. Hoare [23] and incorrectness logic [10, 31] can reason about program properties through preconditions and postconditions. If one treats the DSL $\mathcal{L}$ as an assertion language, the problems of computing *strongest postcondition* [12] and *weakest liberal precondition* [11] in Hoare logic, and *weakest under-approximation postcondition* and *weakest possible precondition* [24] in incorrectness logic, can be expressed within the LOUD framework (see Appendix A).

One key distinction between our approach and the one used in automating computing the above operations in program logics is that in the LOUD framework, one can specify what DSL $\mathcal{L}$ they want their properties to be expressed in. In contrast, the properties produced automatically for, e.g., weakest possible preconditions in incorrectness logic, are the results of syntactic rewrites that often result in complex properties with potentially many quantifiers.

Invariant inference. Many data-driven, CEGIS-style algorithms can infer program invariants—e.g., Elrond [49], abductive inference [14], ICE-learning [19], LoopInvGen [32], Hanoi [30], and Data-Driven CHC Solving [50]. Dynamic techniques like Daikon [16, 17], QuickSpec [42] and Precis [2] can also synthesize invariants through program traces or random tests. The LOUD framework differs from the above works in three key ways: (*i*) The language $\mathcal{L}$ is customizable and is not limited to a set of predefined predicates, and thus the LOUD framework can be used in a domain-agnostic way (as showcased by the many applications presented in Section 7); (*ii*) the LOUD framework supports

both over-approximated and under-approximated reasoning, and (*iii*) the properties synthesized by LOUD are *provably sound strongest $\mathcal{L}$-consequences and sound weakest $\mathcal{L}$-implicants*.

Quantifier Elimination. Many algorithms [7, 13, 15, 27] are built on abductive inference, specifically, approximate quantifier elimination. Gulwani and Musuvathi [22] defined overapproximate existential quantifier elimination as a "cover operation", where the goal is, given a formula $\exists V.\phi$, to find a quantifier-free formula φ such that $(\exists V.\phi) \Rightarrow \varphi$. If φ is restricted to be in a DSL $\mathcal{L}$, the cover problem corresponds to synthesizing $\mathcal{L}$-consequences for queries with an existential quantifier in LOUD framework. Some algorithms [13, 27] also define underapproximate existential quantifier elimination, which corresponds to synthesis of $\mathcal{L}$-implicants. Other approaches are limited to specific theories or require nontrivial, theory-specific primitives [4, 18]. LOUD framework differs from above works because it allows custom DSLs that express the target quantifier-free formulas and thus is not restricted to any fixed theory.

Logic-Based Learning. The angelic and demonic behaviors of nondeterminism correspond to the concepts of brave entailment (i.e., entailment from *some* answer set) and cautious entailment (i.e., entailment from *every* answer set) in logic-based learning [29]. However, the angelic notion in LOUD does not precisely correspond to cautious entailment. While angelic specifications represent an under-approximation of possible behavior, cautious entailment does not necessarily imply an under-approximation. This distinction mirrors the difference between forward reasoning in incorrectness logic and backward reasoning in Hoare's possible correctness, as discussed in Appendix A.

Under-approximation. The LOUD framework could potentially be combined with existing compositional under-approximate reasoning techniques, such as incorrectness logic [31] or compositional symbolic execution [20]. An inherited limitation of syntax-directed under-approximate reasoning is the inability to effectively reason about statements or procedures involving constraints beyond the scope of the theory $\mathcal{T}$ assumed by the under-approximate reasoning framework. We expect one could synthesize weakest $\mathcal{L}$-implicants to approximate such constraints into summaries that are expressible in the theory $\mathcal{T}$ assumed by under-approximation frameworks.

9 Conclusion

This paper presented LOUD, a general framework for synthesizing over- and under-approximated specifications of both deterministic and nondeterministic programs, thus enabling broad applications—e.g., describing sources of bugs in concurrent code and finding winning strategies in two-player games. The paper also presents general procedures for solving LOUD problems using simple synthesis primitives that do not involve complex quantifier alternations.

Currently, our tool ASPIRE is implemented on top of the SKETCH synthesizer, which results in some limitations. First, synthesized formulas are only sound for inputs up to a given bound. Such an issue could be addressed by combining our approach with an off-the-shelf verifier; however, we are not aware of verifiers that can reason about $\mathcal{L}$-implicants—i.e., under-approximated specifications. Our work provides a motivation for building such verifiers. Second, SKETCH limits us from exploring applications that involve inputs of unbounded length—e.g., reasoning about infinite traces, LTL formulas, and reactive systems. Our work thus opens an opportunity for the research community: by improving efficiency and providing stronger soundness guarantees for the primitives used to solve LOUD problems, researchers can tackle the many applications supported by the framework.

Acknowledgement

Supported, in part, by a Jacobs Faculty Scholarship, and by NSF grants CCF-2422214, CCF-2506134, CCF-2446711. Any opinions and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

Data-Availability Statement

Our implementation of ASPIRE that instantiates the LOUD framework is built on top of the SKETCH synthesizer. We provide a comprehensive Docker image on Zenodo that contains the source code and binary of ASPIRE, all the necessary dependencies, and scripts and datasets used in the experiments described in Section 7 [33].

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[2] Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. doi:10.1145/3485481

[3] D. Beyer. 2024. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *Proc. TACAS (3) (LNCS 14572)*. Springer, 299–329. doi:10.1007/978-3-031-57256-2_15

[4] Nikolaj Bjorner and Mikolas Janota. 2015. Playing with Quantified Satisfaction. In *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations (EPiC Series in Computing, Vol. 35)*, Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov (Eds.). EasyChair, 15–27. doi:10.29007/vv21

[5] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. 2018. *Graph Games and Reactive Synthesis.* Springer International Publishing, Cham, 921–962. doi:10.1007/978-3-319-10575-8_27

[6] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2022. Abstract interpretation repair. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 426–441. doi:10.1145/3519939.3523453

[7] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. 2020. Combined Covers and Beth Definability. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 181–200.

[8] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 672–687. doi:10.1145/3385412.3385969

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. doi:10.1145/512950.512973

[10] Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171.

[11] Edsger W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall. https://books.google.com/books?id=MsUmAAAAMAAJ

[12] Edsger W Dijkstra and Carel S Scholten. 1990. *Predicate calculus and program semantics.* Springer-Verlag.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise reasoning for programs using containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 187–200. doi:10.1145/1926385.1926407

[14] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. *ACM SIGPLAN Notices* 47, 6 (2012), 181–192.

[15] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 443–456. doi:10.1145/2509136.2509511

[16] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Software Eng.* 27, 2 (2001), 99–123. doi:10.1109/32.908957

[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. doi:10.1016/j.scico.2007.01.015

[18] Azadeh Farzan and Zachary Kincaid. 2017. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.* 2, POPL, Article 61 (dec 2017), 30 pages. doi:10.1145/3158149

[19] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 69–87. doi:10.1007/978-3-319-08867-9_5

[20] Patrice Godefroid. 2007. Compositional dynamic test generation. *SIGPLAN Not.* 42, 1 (jan 2007), 47–54. doi:10.1145/1190215.1190226

[21] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. 2005. Proof-guided underapproximation-widening for multi-process systems. *SIGPLAN Not.* 40, 1 (jan 2005), 122–131. doi:10.1145/1047659.1040316

[22] Sumit Gulwani and Madan Musuvathi. 2008. Cover Algorithms and Their Combination. In *Programming Languages and Systems*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207.

[23] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. doi:10.1145/363235.363259

[24] C. A. R. Hoare. 1978. Some Properties of Predicate Transformers. *J. ACM* 25, 3 (jul 1978), 461–480. doi:10.1145/322077.322088

[25] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas W. Reps, and Subhajit Roy. 2022. Synthesizing Abstract Transformers. *Proc. ACM Program. Lang.* OOPSLA (2022). doi:10.1145/3563334

[26] Jinwoo Kim. 2022. Messy-Release. https://github.com/kjw227/Messy-Release.

[27] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 17–34.

[28] Sandeep Kumar. 2011. Specification mining in concurrent and distributed systems. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1086–1089. doi:10.1145/1985793.1986002

[29] Mark Law, Alessandra Russo, and Krysia Broda. 2019. *Logic-Based Learning of Answer Set Programs.* Springer International Publishing, Cham, 196–231. doi:10.1007/978-3-030-31423-1_6

[30] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3385412.3385967

[31] Peter W. O'Hearn. 2019. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (dec 2019), 32 pages. doi:10.1145/3371078

[32] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 42–56. doi:10.1145/2908080.2908099

[33] Kanghee Park. 2025. *Loud: Synthesizing Strongest and Weakest Specifications.* doi:10.5281/zenodo.14934344

[34] Kanghee Park, Loris D'Antoni, and Thomas Reps. 2023. Synthesizing Specifications. 7, OOPSLA2, Article 285 (oct 2023), 30 pages. doi:10.1145/3622861

[35] Kanghee Park, Keith J. C. Johnson, Loris D'Antoni, and Thomas W. Reps. 2023. Modular System Synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, Alexander Nadel and Kristin Yvonne Rozier (Eds.). IEEE, 257–267. doi:10.34727/2023/ISBN.978-3-85448-060-0_34

[36] Kanghee Park, Xuanyu Peng, and Loris D'Antoni. 2024. LOUD: Synthesizing Strongest and Weakest Specifications. arXiv:2408.12539 [cs.PL] https://arxiv.org/abs/2408.12539

[37] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[38] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. 252–266. doi:10.1007/978-3-540-24622-0_21

[39] Thomas W. Reps and Aditya V. Thakur. 2016. Automating Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 3–40. doi:10.1007/978-3-662-49122-5_1

[40] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 198–216.

[41] David A. Schmidt. 2007. A calculus of logical relations for over- and underapproximating static analyses. *Sci. Comput. Program.* 64, 1 (jan 2007), 29–53. doi:10.1016/j.scico.2006.03.008

[42] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017), e18.
[43] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Ph. D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.
[44] Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495. doi:10.1007/s10009-012-0249-7
[45] Aditya V. Thakur, Matt Elder, and Thomas W. Reps. 2012. Bilateral Algorithms for Symbolic Abstraction. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. 111–128. doi:10.1007/978-3-642-33125-1_10
[46] Aditya V. Thakur and Thomas W. Reps. 2012. A Method for Symbolic Computation of Abstract Operations. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 174–192. doi:10.1007/978-3-642-31424-7_17
[47] Rong Wang, Zuohua Ding, Ning Gui, and Yang Liu. 2017. Detecting Bugs of Concurrent Programs With Program Invariants. *IEEE Transactions on Reliability* 66, 2 (2017), 425–439. doi:10.1109/TR.2017.2681107
[48] S. Zdancewic and A.C. Myers. 2003. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings*. 29–43. doi:10.1109/CSFW.2003.1212703
[49] Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-driven abductive inference of library specifications. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. doi:10.1145/3485493
[50] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721. doi:10.1145/3192366.3192416

Received 2024-10-16; accepted 2025-02-18