Note on AI Assistance. This project was developed with the help of AI tools. AI was used to assist with parts of the implementation, and also to support the drafting of this report. All final design decisions, code integration, experimental results, and write-up choices were reviewed and finalized by the authors.

SMT-Based Invariant Synthesis for Program Verification

Yuvraj Verma Vedant Rana

December 12, 2025

Abstract

Invariant synthesis is a central challenge in automated program verification, particularly for programs with loops. This project implements a template-based invariant synthesis framework that reduces invariant discovery to satisfiability checking in SMT. For each loop, we extract a symbolic transition relation from the loop guard and assignments, and enumerate candidate invariants from fixed syntactic templates. The supported templates include linear inequalities, Boolean disjunctive normal forms over linear atoms, explicit disjunctive invariants, and restricted quadratic polynomials over a small number of variables. Each candidate invariant is validated using Z3 by checking initialization and inductiveness conditions, ensuring that the invariant holds at loop entry and is preserved by one iteration of the loop. We further extend the approach to selected array and list-manipulating programs by encoding indexed accesses using SMT arrays and restricting attention to outer-loop transitions in the presence of nested loops. Experimental results on a suite of benchmarks show that bounded template enumeration combined with SMT-based validation can automatically synthesize loop invariants for many programs, while also revealing limitations arising from template expressiveness, coefficient bounds, and solver performance for disjunctive, non-linear, and array-aware invariants.

Repository: github.com/yuvraj-verma01/Slaps-2025-Yuvraj-Vedant-

Contents

1 Introduction

1.1 Motivation

Formal verification aims to establish program correctness by proving that all executions of a program satisfy a given specification. For imperative programs with loops, this task typically requires discovering **loop invariants**: assertions over program variables that hold at the loop entry and are preserved by every iteration of the loop. Loop invariants are essential for reasoning about safety properties such as array bounds and functional correctness.

Despite their importance, loop invariants are often difficult to construct manually and challenging to infer automatically. Classical approaches based on abstract interpretation compute invariants by iteratively approximating reachable states, but they may lose precision or require carefully tuned heuristics. An alternative line of work treats invariant synthesis as a constraint-solving problem, where invariants are generated by searching over formulas of a fixed syntactic shape and validating them using logical reasoning. Advances in satisfiability modulo theories (SMT) solvers make this approach particularly attractive, as they allow expressive arithmetic and logical constraints to be checked efficiently.

This project explores an SMT-guided, template-based approach to loop invariant synthesis. Rather than computing invariants through iterative approximation, the system explicitly constructs candidate invariants from predefined templates and uses an SMT solver to check whether each candidate is inductive. This design emphasizes clarity of semantics, direct logical validation, and extensibility to richer invariant classes such as disjunctive and non-linear formulas.

1.2 Project Problem

The core problem addressed in this project is the automatic synthesis of inductive loop invariants for imperative programs. Given a loop with a guard, a loop body consisting of assignments, and a precondition describing the program state at loop entry, the goal is to construct an invariant I such that: (i) I holds at the beginning of the loop, and (ii) if I holds before an iteration of the loop and the loop guard is satisfied, then I also holds after executing the loop body once.

Formally, the task is to find a formula I over program variables that is both **initialized** and **inductive** with respect to the loop's transition relation. In this project, the transition relation is extracted directly from the loop guard and assignments and models a single iteration of the loop. Candidate invariants are drawn from fixed syntactic templates, such as linear inequalities or disjunctions of linear constraints, and are validated using satisfiability queries to an SMT solver.

The input to the system is a simplified loop program written in a Dafny-like syntax, together with a choice of invariant template class. The output is either a synthesized invariant that satisfies the inductiveness conditions or a report that no invariant was found within the given template bounds.

1.3 Goals

The specific goals of this project are as follows:

- To design and implement a template-based framework for loop invariant synthesis that reduces invariant validation to SMT satisfiability checks.
- To support multiple classes of invariants, including linear arithmetic invariants, Boolean disjunctive normal forms over linear atoms, explicit disjunctive invariants, and restricted quadratic invariants.
- To integrate the synthesis procedure with the Z3 SMT solver and use Dafny-style semantics for validation of candidate invariants.
- To extend the approach to selected programs involving arrays, lists, and nested loops using SMT array theories and controlled abstraction.
- To evaluate the effectiveness and limitations of bounded template enumeration through experiments on a suite of benchmark programs.

2 Background: How SMT Solving Works

2.1 SAT vs. SMT and Theories

The Boolean satisfiability problem (SAT) asks whether there exists an assignment of truth values to Boolean variables that satisfies a propositional formula. Modern SAT solvers are highly optimized and can scale to formulas with millions of variables, but they are limited to purely Boolean structure and cannot directly reason about arithmetic expressions, arrays, or program variables.

Satisfiability Modulo Theories (SMT) extends SAT by allowing formulas to include constraints drawn from background theories such as integer arithmetic, arrays, and uninterpreted functions. An SMT formula combines Boolean connectives with theory atoms, for example linear inequalities over integers or array read/write expressions. A formula is satisfiable if there exists a valuation of both Boolean variables and theory-level variables that satisfies all constraints simultaneously.

In the context of program verification, SMT is particularly well-suited because program semantics naturally involve arithmetic relations between variables, branching conditions, and structured data such as arrays. In this project, SMT is used primarily over the theories of integer arithmetic and arrays to reason about loop guards, assignments, and candidate invariants generated during synthesis.

2.2 DPLL(T) and Modern SMT Solvers

Most modern SMT solvers, including Z3, are based on the DPLL(T) architecture. At a high level, DPLL(T) separates reasoning about the Boolean structure of a formula from reasoning within individual background theories. The solver first constructs a Boolean abstraction of the input formula, treating each theory atom as a propositional variable, and uses a SAT solver to search for a satisfying Boolean assignment.

Whenever the SAT solver proposes a candidate Boolean assignment, the corresponding set of theory constraints is passed to specialized theory solvers. If these constraints are inconsistent within a theory, the theory solver reports a conflict, which is translated into a learned clause and

added back to the Boolean search. This process iterates until either a consistent assignment is found (the formula is satisfiable) or all possibilities are exhausted (the formula is unsatisfiable).

This architecture allows SMT solvers to efficiently handle formulas that mix Boolean control flow with arithmetic and data-structure constraints. In particular, it enables SMT solvers to decide the satisfiability of formulas encoding loop transitions and invariant violations, which is central to the invariant validation strategy employed in this project.

2.3 Why SMT Helps Invariant Synthesis

SMT solving serves as a validation mechanism for candidate loop invariants generated from fixed syntactic templates (linear, quadratic, etc). Once a candidate formula is constructed, the task reduces to determining whether there exists a program state that violates the invariant at loop entry or after a single iteration of the loop. These checks can be expressed as satisfiability queries over arithmetic and logical constraints derived directly from the program semantics.

SMT solvers are well-suited to this role because they can reason precisely about the combination of arithmetic constraints, Boolean structure, and array operations that arise from loop guards, assignments, and invariant templates. Rather than approximating program behavior, the solver is asked to find a concrete counterexample to a candidate invariant; if no such counterexample exists, the invariant is accepted as valid.

This method enables invariant synthesis to be formulated as a bounded search over candidate formulas, with SMT solving providing exact validation. The ability of SMT solvers to produce concrete counterexamples when a candidate fails also supports systematic debugging and refinement of invariant templates.

3 Related Work

The automatic synthesis of loop invariants has long been a central problem in program verification. Broadly, existing approaches differ in how invariants are represented and how they are discovered: either by iteratively approximating reachable states, or by searching directly for invariants of a fixed syntactic form using constraint solving.

Early verification techniques for loops and hybrid systems relied on abstract interpretation and reachability analysis, where invariants are computed as fixed points over abstract domains. While these methods are general, they often require widening or extrapolation heuristics to ensure termination, which can lead to a loss of precision and make it difficult to recover precise arithmetic relationships between program variables.

Constraint-based and template-based approaches offer an alternative by restricting attention to invariants of a predefined shape. In this line of work, an invariant template with unknown coefficients is fixed in advance, and the task reduces to solving constraints that ensure the invariant is inductive. Colon et al. [1] demonstrated that linear loop invariants can be generated by encoding inductiveness conditions as constraints over unknown coefficients, establishing a foundational connection between invariant generation and constraint solving.

Subsequent work extended this idea to richer invariant classes. Gulwani and Tiwari [2] showed how disjunctive and polynomial invariants of bounded degree can be synthesized by

translating verification conditions into $\exists\forall$ constraints and solving them using algebraic techniques such as Farkas' Lemma. This work highlights a key insight that motivates the present approach: when the invariant shape is known, deep inductive invariants can be found without computing fixpoints over program states.

Rybalchenko [3] further developed constraint-based methods for program verification, illustrating how invariants, ranking functions, and related assertions can be synthesized by separating constraint generation from constraint solving and delegating the solving phase to off-the-shelf SMT solvers. This separation allows verification tools to leverage advances in SMT technology without hard-coding domain-specific reasoning.

The approach taken here follows this template-based, constraint-driven tradition. Candidate invariants are generated from restricted syntactic families and validated using SMT satisfiability checks derived from the loop semantics. Unlike abstract interpretation, this avoids iterative state-space approximation, and unlike full $\exists\forall$ elimination frameworks, it relies directly on SMT solvers to validate candidate invariants within bounded search spaces. This positioning emphasizes clarity and modularity, while exposing the trade-offs inherent in template expressiveness and solver scalability.

4 Implementation Overview

This section describes the design and implementation of the invariant synthesis system. The implementation follows a uniform pipeline across multiple invariant classes, differing primarily in the structure of the invariant templates and the corresponding SMT encodings.

4.1 System Architecture

The system operates as a source-level invariant synthesizer for Dafny programs containing imperative loops. Its architecture consists of four main stages:

1. **Program Parsing and Loop Extraction.** The input Dafny program is parsed using the Week 5 parser to extract method summaries, loop conditions, loop bodies, and relevant variables. When parser information is unavailable or incomplete, lightweight syntactic fallback parsing is used to recover loop guards and assignments directly from the source text.

2. **Transition Relation Construction.** For each loop, the system constructs a symbolic transition relation that captures the effect of one loop iteration. Program variables are duplicated into pre-state and post-state versions, and assignments in the loop body are translated into constraints relating these states. Variables not assigned in the body are implicitly preserved across iterations.

3. **Candidate Invariant Generation.** Invariant candidates are generated from fixed syntactic templates, such as linear inequalities, Boolean combinations of linear atoms, or bounded-degree polynomial expressions. Template parameters (e.g., coefficients and constants) are instantiated by bounded enumeration. Depending on the synthesis mode,

candidates may correspond either to full invariant formulas or to atomic constraints that are later combined syntactically.

4. **SMT-Based Validation.** Invariant validation is performed using Z3 by checking satisfiability, initialization, and inductiveness conditions. In Boolean synthesis modes, these checks are applied to entire candidate formulas. In disjunctive–linear modes, individual atomic invariants are validated independently, while disjunctive combinations are formed syntactically after validation.

This modular structure allows different invariant classes to reuse the same parsing and validation pipeline while varying only the template generation logic.

4.2 Invariant Templates and Search

Invariant synthesis is performed by enumerating candidates from restricted template families. Each template defines the syntactic form of admissible invariants, while the search procedure instantiates concrete candidates by enumerating bounded coefficient ranges.

Linear Templates. Linear invariants are of the form

$$\sum_{i=1}^{n} a_i x_i \leq c,$$

where the coefficients a_i and constant c range over small bounded integer domains. Candidate templates with all-zero coefficients are discarded, and normalization is applied to avoid generating redundant scalar multiples of the same inequality.

Quadratic Templates. Quadratic invariants are generated from fixed-degree polynomial templates over at most two variables, of the form

$$ax^2 + by^2 + cxy + dx + ey + f \leq 0.$$

The search is bounded both by coefficient ranges and by the number of non-zero terms, prioritizing simpler expressions. Quadratic candidates are validated using the same SMT-based inductiveness checks as linear invariants.

Arrays, Lists, and Nested Loops. For programs involving arrays or lists, scalar invariants are extended with terms involving array lengths and selected array accesses. The transition relation models array updates using functional store expressions, while nested loops are conservatively excluded from the transition relation to keep SMT queries tractable. This design choice focuses the analysis on outer-loop invariants while maintaining soundness.

4.3 Integration with Dafny and Z3

The system integrates Dafny and Z3 at the level of logical semantics rather than through Dafny's internal verifier. Dafny programs are treated as a source of loop structure and variable declarations, while invariant checking is performed entirely through explicit SMT encodings.

For each invariant candidate or atomic constraint I, the following checks may be encoded as SMT queries:

- **Satisfiability:** I is not contradictory.
- **Initialization:** The invariant holds in all states induced by assignments before the loop.
- **Inductiveness:** Assuming I holds before an iteration and the loop guard is true, executing the loop body preserves I.

These checks are discharged by Z3 using standard satisfiability queries. Inductiveness is verified by checking the unsatisfiability of the negated post-invariant under the transition relation. This approach avoids fixpoint computation and relies entirely on solver-driven validation.

5 Invariant Classes

Invariant synthesis is organized around a small number of template families. Each family fixes the syntactic form of candidate invariants and determines the search space explored. Across all families, candidates are validated via SMT checks for satisfiability, initialization, and inductiveness; the primary differences lie in (i) the template language and (ii) how candidates are enumerated and combined. This section summarizes each strategy and its observed limitations.

5.1 Linear Invariants

Template form. Linear invariants are instantiated from inequalities of the form

$$\sum_{k=1}^{d} a_k x_k \leq c,$$

where $x_1, \ldots, x_d$ are loop variables and the coefficients a_k and constant c are integers drawn from fixed, bounded ranges. Each invariant therefore describes a half-space in the program state space.

Search strategy. Synthesis proceeds by bounded enumeration of coefficient vectors and constants. Trivial templates (such as those with all-zero coefficients) are discarded, and simple normalization is applied to eliminate obvious scalar redundancies. Each candidate inequality is then validated using Z3 against three conditions: (i) satisfiability, (ii) initialization (the invariant holds before loop entry), and (iii) inductiveness (the invariant is preserved by one loop iteration under the loop guard and transition relation). Candidates that satisfy all checks are retained until a fixed limit is reached.

Example. Consider the following benchmark:

```
1 method DoubleIncrement(n: int) returns (i: int, j: int)
2 {
3   i := 0;
4   j := 0;
5   while (i < n && j < 2*n)
```

```
  {
    i := i + 1;
    j := j + 2;
  }
}
```

Listing 1: Benchmark `DoubleIncrement`

Running linear invariant synthesis on this loop yields the following inductive constraints:

$$-i \leq 0, \quad -i - j \leq 0, \quad -2i - j \leq 0.$$

These invariants collectively express non-negativity and simple linear relationships between the loop counters. For instance, $-i \leq 0$ captures $i \geq 0$, while $-i - j \leq 0$ captures the fact that both counters increase monotonically from zero and never become negative. All of these constraints are inductive under the loop transition and sufficient to satisfy Dafny's loop verification conditions.

Limitations. The linear approach is strongly template- and bound-dependent. If the required invariant lies outside the enumerated coefficient or constant ranges, it will not be discovered. Linear templates also cannot express non-linear relationships (such as $s = i^2$) or semantic properties of data structures (e.g., sortedness or permutation of arrays). Finally, the procedure is *goal-agnostic* i.e. it checks inductiveness of candidates but does not verify whether an invariant is strong enough to establish the method's postcondition. As a result, it may return inductive but semantically weak invariants that do not capture the intended correctness argument.

5.2 Boolean and Disjunctive Invariants

Template form. Both strategies ultimately produce invariants in *disjunctive normal form (DNF) over linear atoms*:

$$\bigvee_{r=1}^{m} \left(\bigwedge_{\ell=1}^{k_r} A_{r,\ell} \right), \qquad A_{r,\ell} \equiv \sum_i a_i x_i \leq c.$$

Thus, the invariant language is identical in both cases; the difference lies entirely in how such DNFs are synthesized, validated, and reported.

Boolean synthesis (Week 9, `-bool`). The Boolean strategy attempts to synthesize a DNF invariant *directly*. It first enumerates a small pool of linear atomic predicates using tight coefficient and constant bounds. Each atom is filtered via a satisfiability check under the loop guard to discard predicates that are impossible in reachable states. It then constructs candidate DNFs by combining small conjunctions of these atoms into bounded-size disjunctions. Each full DNF candidate is validated using Z3 by checking satisfiability, initiation, and inductiveness with respect to the loop transition relation.

Crucially, this synthesis procedure is greedy and bounded: it prioritizes simple atoms and small DNFs and terminates as soon as the first inductive candidate is found (or the search budget is exhausted). As a result, Boolean synthesis often returns weak but easily provable

invariants, such as non-negativity constraints, even when stronger relational invariants exist. This behavior arises from the enumeration order and restricted template space rather than a limitation of the underlying SMT solver.

Disjunctive–linear synthesis (Week 10, `-no-bool`). The Week 10 disjunctive–linear mode (`-no-bool`) follows a two-phase workflow. First, the tool synthesizes multiple *independent linear invariants* using the standard linear template enumerator. Each invariant is validated individually for initiation and inductiveness.

Second, no further semantic reasoning is performed. The validated linear invariants are grouped into a disjunctive normal form (DNF) *purely at the reporting level*. The resulting disjunction does not arise from joint synthesis or case-sensitive reasoning, but from post-processing already verified linear facts.

Importantly, the disjunctive structure is *not* synthesized as a semantic invariant. Each linear constraint is discovered and validated in isolation, and the final DNF is formed without checking whether individual disjuncts correspond to reachable execution modes or mutually exclusive cases. As a result, the reported disjunction should be interpreted as a presentation-level aggregation of independently valid invariants rather than a semantically meaningful case split.

Worked example: `Toggle`. Consider the following mode-switching loop controlled by a Boolean flag:

```
method Toggle()
{
  var i := 0;
  var x := 0;
  var y := 0;
  var flip := 0;
  while (i < 4)
  {
    if (flip == 0) {
      x := x + 1;
      y := y + 2;
      flip := 1;
    } else {
      x := x + 2;
      y := y + 1;
      flip := 0;
    }
    i := i + 1;
  }
}
```

Boolean synthesis produces the invariant:

```
(-flip <= 0 && -i <= 0 && -x <= 0)
```

which expresses simple non-negativity constraints. This invariant is inductive and therefore accepted, but it does not distinguish between the two update modes of the loop.

In contrast, disjunctive–linear synthesis produces:

```
(flip <= 1 && flip + i <= 5) ||
(2*flip + i <= 10 && 3*flip <= 4 && 3*flip + i <= 7)
```

This invariant arises from combining multiple linear invariants discovered independently. The resulting DNF reflects an aggregation of linear facts rather than a jointly synthesized semantic case split, but nonetheless exposes a two-branch structure that is obscured by Boolean synthesis.

Relationship to linear synthesis. Running linear synthesis alone (Week 6 or Week 9 without `-bool`) produces a *set* of independent linear invariants such as `flip <= 1`, `flip + i <= 5`, and related bounds. Boolean synthesis collapses this information by stopping at the first inductive DNF found, whereas disjunctive–linear synthesis aggregates these linear facts into a disjunctive presentation without altering their individual validity.

Boolean vs. non-Boolean modes in Week 10. The Week 10 implementation exposes two execution modes that differ in where disjunction is introduced. When invoked with `-bool`, the tool delegates invariant discovery to the Week 9 Boolean synthesis procedure. If the Boolean engine returns a single conjunctive invariant φ, the Week 10 wrapper enforces a DNF-shaped presentation by emitting $\varphi \vee \varphi$. This transformation does not strengthen the invariant and serves only to standardize output format; no additional semantic reasoning is introduced at this stage.

In contrast, when invoked with `-no-bool`, Boolean synthesis is disabled. The tool instead enumerates multiple independent linear invariants using the linear template language. These invariants are not combined semantically, but are grouped into disjunctive cases at the reporting level. While this may yield disjunctions with syntactically distinct branches, the disjunction itself is not jointly verified and should not be interpreted as encoding mutually exclusive execution modes.

Limitations. Both Boolean and disjunctive–linear modes may produce invariants that are syntactically disjunctive but semantically weak. Boolean synthesis may terminate prematurely due to its greedy stopping criterion, often yielding simple non-negativity constraints.

The Week 10 disjunctive–linear mode does not perform joint reasoning across disjuncts, enforce reachability of individual branches, or eliminate redundant cases. Consequently, reported DNFs may contain overlapping or subsumed disjuncts and should be interpreted as aggregated linear facts rather than true semantic case splits. A fully semantic disjunctive invariant synthesis procedure would require jointly synthesizing and validating disjuncts under reachability, non-redundancy, and mutual usefulness constraints, which is outside the scope of the present implementation.

5.3 Quadratic Invariants

Template form. Quadratic invariants are instantiated from restricted degree-2 polynomial inequalities over a small number of variables (typically two):

$$ax^2 + by^2 + cxy + dx + ey + f \leq 0, \qquad a, b, c, d, e, f \in \mathbb{Z}.$$

The variable set is deliberately limited to keep SMT queries tractable, and templates are interpreted over integer arithmetic.

Search strategy. Candidates are generated by bounded enumeration of coefficient tuples, with priority given to simpler templates (i.e., fewer non-zero coefficients and smaller absolute values). For each candidate, Z3 is used to check satisfiability, initialization, and inductiveness under the loop guard and the extracted transition relation. As in the linear case, no fixpoint iteration or widening is performed; synthesis relies entirely on solver-based validation of individual templates.

Example. Consider the following benchmark, which computes the sum of the first n odd numbers:

```
method SumOfSquares(n: int) returns (i: int, s: int)
  requires n >= 0
  ensures s == n * n
{
  i := 0;
  s := 0;
  while i < n
  {
    s := s + 2 * i + 1;
    i := i + 1;
  }
}
```

Listing 2: Benchmark `SumOfSquares`

Quadratic synthesis produces inductive constraints such as:

$$-i^2 + s \leq 0, \qquad i^2 - s \leq 0,$$

along with small shifted variants (e.g., $i^2 - s - 1 \leq 0$). Together, these inequalities characterize the quadratic relationship $s = i^2$ maintained by the loop and are sufficient to establish the postcondition when combined.

Limitations. While quadratic templates significantly increase expressive power compared to linear invariants, the synthesis procedure remains bounded and incomplete.

First, the search space grows rapidly with coefficient ranges and the number of variables. To keep enumeration feasible, the implementation restricts attention to a small subset of loop

variables and small integer bounds. As a result, valid quadratic invariants may lie outside the explored template space and remain undiscovered.

Second, the procedure is postcondition-agnostic: candidates are accepted solely on the basis of satisfiability, initialization, and inductiveness, without explicitly checking whether they imply the method's ensures clause. While this design keeps synthesis modular and reusable, it may produce invariants that are correct but insufficient on their own to discharge the full specification.

Finally, quadratic synthesis inherits the limitations of the extracted transition relation. Although the implementation correctly handles standard Dafny loop syntax with invariants and decreases clauses, any abstraction or omission in the transition model may restrict the class of discoverable invariants.

5.4 Lists and Nested Loop

Encoding approach. To extend invariant synthesis to programs manipulating arrays and lists, the system augments the scalar variable vocabulary with explicit length terms such as $\text{len}(a)$ (corresponding to `a.Length` in Dafny). Array reads and writes are modeled at the SMT level using functional array semantics: reads via Select and updates via Store. Lengths are treated as immutable across loop iterations.

Invariant templates remain quantifier-free arithmetic constraints over scalar variables, index variables, and length expressions. The system does not generate universally quantified invariants over array indices; array accesses are treated as additional symbolic terms when they appear syntactically in assignments or guards.

Example. Consider the loop inside the `Partition` procedure used by quicksort:

```
while j < high
  invariant low <= i <= j <= high
{
  if a[j] <= pivot {
    Swap(a, i, j);
    i := i + 1;
  }
  j := j + 1;
}
```

Applying length-aware synthesis to this loop produces inductive arithmetic constraints involving index variables and array length, such as:

$$high + i - j + \text{len}(a) \leq c,$$

for small constants c. These invariants reflect that the template language can combine loop indices with array length terms and that such relationships are preserved across iterations. However, they do not express semantic properties such as the partitioning condition (elements less than the pivot appearing before index i).

On benchmarks such as `FibIter`, where array or list structure is absent but functional relationships dominate, length-aware synthesis similarly defaults to simple arithmetic bounds

(e.g., $i - n \leq c$) when no template captures the intended recurrence.

Limitations. The array and list extension improves syntactic coverage by supporting array updates and length expressions, but it remains fundamentally limited to quantifier-free arithmetic templates. As a result, it cannot express content-level properties essential to many array algorithms, including sortedness, permutation invariants, or correctness of partitioning.

Treating array accesses and length terms as additional symbolic variables also increases the dimensionality of the template space, significantly worsening scalability under bounded enumeration. Finally, nested loops are handled conservatively to maintain tractability: synthesis targets a single loop level at a time, which prevents discovery of invariants that relate variables across multiple nested iterations or recursive calls.

6 Benchmark Results

This section summarizes the synthesized invariants produced on the Week 3 benchmark suite and additional quadratic benchmarks. Reported invariants are grouped by the invariant classes defined in Section 5. Unless stated otherwise, each benchmark was executed with the corresponding synthesizer and the resulting instrumented Dafny file was checked using the Dafny verifier.

6.1 DNF Benchmarks: Boolean vs. Disjunctive-Linear

Several benchmarks were designed to exercise mode-dependent behavior (e.g., guarded updates controlled by a flag). On these programs, the DNF-based strategies expose a practical difference between (i) *direct* Boolean synthesis that searches for a small DNF formula (Week 9, `-bool`) and (ii) *disjunctive presentation* obtained by collecting independently valid linear constraints and formatting them as a DNF with shallow cases (Week 10, `-no-bool`). In our runs, the Boolean engine frequently terminated early with simple non-negativity-style constraints, while the disjunctive-linear mode returned lightweight two-case DNFs built from single-atom disjuncts.

`benchmark_boolean.dfy`. On `benchmark_boolean.dfy`, Boolean synthesis returned the conjunction

$$(-a \leq 0) \ \wedge \ (-flag \leq 0),$$

i.e. $a \geq 0$ and $flag \geq 0$. In contrast, disjunctive-linear synthesis returned a two-branch DNF with singleton cases:

$$(3 \cdot flag \leq 10) \ \vee \ (b - 3i \leq 10).$$

This illustrates the characteristic behavior of the two modes in our bounded search setting: the Boolean strategy quickly finds a trivially inductive conjunction, while the disjunctive-linear mode surfaces at least one additional guard-/update-related linear bound as a separate case.

`booleanproblem1.dfy (Toggle).` For the mode-switching loop in `booleanproblem1.dfy`, Boolean synthesis returned:

$$(-flip \leq 0) \ \wedge \ (-i \leq 0),$$

i.e. $flip \geq 0$ and $i \geq 0$. Disjunctive-linear synthesis returned the DNF:

$$(flip + i \leq 5) \ \vee \ (2 \cdot flip + i \leq 10).$$

Although each disjunct is a single linear atom (rather than a structured conjunction describing mutually exclusive modes), the resulting formula is syntactically disjunctive and captures two distinct linear bounds that hold across iterations under the extracted transition relation.

`evenodd.dfy`. On `evenodd.dfy`, Boolean synthesis again produced a basic conjunction:

$$(-i \leq 0) \ \wedge \ (-neg \leq 0),$$

corresponding to $i \geq 0$ and $neg \geq 0$. The disjunctive-linear mode returned:

$$(i - pos \leq 0) \ \vee \ (3i - 2 \cdot neg - pos \leq 10).$$

As in the previous benchmarks, the disjunctive result should be interpreted as a shallow case split over independently discovered linear constraints rather than a semantically meaningful partition of execution modes; nevertheless, it often exposes relationships between counters (e.g. i and pos) that are not visible in the early-terminating Boolean output.

6.2 Array/List Benchmarks

Week 12 extends linear synthesis to incorporate array lengths and (when present) array access terms. The following summarizes results on the Week 3 problems processed through the array/list-capable pipeline.

`problem1.dfy`. For the loop with condition `i < a.Length`, synthesis produced a set of simple length-relative bounds, e.g.:

$$i - \text{len}(a) \leq 1, \quad i - \text{len}(a) \leq 2, \quad i - \text{len}(a) \leq 3,$$

and scaled variants such as $2i - 2\text{len}(a) \leq 3, 5$. The debug trace confirms that $\text{len}(a)$ and the access term $a[i]$ were included among template variables, although the returned invariants remained purely index/length-relational.

`problem2.dfy`. For the loop with condition `y != 0`, no invariants were synthesized by the array/list pipeline. This is consistent with the limitation that the template language and transition extraction target arithmetic relations of a restricted syntactic form; when the extracted transition model is incomplete or the needed invariant lies outside the explored template space, the search may return an empty set.

problem3.dfy. Two loops were detected. For the first loop (`a < n`) and the second loop (`b < n`), synthesis returned simple bounds of the form:

$$a - n \leq 1, 2, 3 \quad \text{and} \quad b - n \leq 1, 2, 3,$$

again with scaled variants such as $2a - 2n \leq 3, 5$ and $2b - 2n \leq 3, 5$.

problem4.dfy, problem5.dfy, problem6.dfy. For these benchmarks (all containing a loop of the form `i < r`), synthesis consistently produced the same family of bounds:

$$i - r \leq 1, 2, 3, \qquad 2i - 2r \leq 3, 5.$$

In `problem6.dfy`, the first loop received these bounds, while the second loop (with additional array terms such as `mods[i]` and `vals[i]`) returned no invariants, highlighting the scalability and expressiveness limits when the template variable set expands substantially.

problem7.dfy (Quicksort Partition). For the loop `j < high`, synthesis produced length-aware constraints:

$$high + i - j + \text{len}(a) \leq 1, 2, 3,$$

and scaled variants such as $2high + 2i - 2j + 2\text{len}(a) \leq 3, 5$. These invariants demonstrate that the template language can combine index and length terms, though they do not encode semantic partition correctness.

problem8.dfy (FibIter). For the loop `i < n`, synthesis returned simple bounds of the form:

$$i - n \leq 1, 2, 3, \qquad 2i - 2n \leq 3, 5.$$

This matches the expectation that, without templates capturing the intended functional relationship between (a, b) and Fibonacci values, bounded linear search tends to recover only weak arithmetic bounds.

6.3 Quadratic Benchmarks

Quadratic synthesis was evaluated on benchmarks where the intended invariant is non-linear and naturally expressible as a degree-2 arithmetic relation. In both cases, the synthesizer enumerated bounded quadratic templates over two variables and validated candidates using satisfiability, initialization, and inductiveness checks against the extracted loop transition relation.

quadratic_sum_of_squares.dfy. For the loop in `SumOfSquares` (which computes n^2 by summing the first n odd numbers), quadratic synthesis produced the following inductive constraints:

$$-i^2 + s \leq 0, \qquad i^2 - s \leq 0,$$

along with several scaled and shifted variants, including

$$-i^2 + s - 1 \leq 0, \quad -i^2 + s - 2 \leq 0, \quad -2i^2 + 2s \leq 0.$$

These inequalities collectively characterize the quadratic relationship $s = i^2$ maintained by the loop, expressed in an inequality-based form compatible with the template language. The presence of scaled variants reflects the normalization-free enumeration strategy: multiple algebraically equivalent inequalities are accepted as long as they are inductive under the transition relation.

`quadratic_triangular.dfy`. For `TriangularSum`, which accumulates the sum of the first $n - 1$ natural numbers, quadratic synthesis produced inductive constraints consistent with a $2 \cdot sum \approx k^2$ relationship:

$$-k^2 + 2 \cdot sum \leq 0,$$

together with shifted variants such as

$$-k^2 + 2 \cdot sum - 1 \leq 0, \quad -k^2 + 2 \cdot sum - 2 \leq 0, \quad -k^2 + 2 \cdot sum - 3 \leq 0.$$

In addition, the synthesizer returned simple linear side invariants, including

$$-k \leq 0, \quad -k - 1 \leq 0,$$

which capture non-negativity and lower bounds on the loop counter. These linear constraints are inductive under the loop transition and coexist with the quadratic invariants in the final result set.

Overall, the quadratic benchmarks demonstrate that bounded quadratic template enumeration can recover meaningful non-linear relationships maintained by loops. However, as with linear synthesis, the results are postcondition-agnostic: while the discovered invariants are inductive and semantically correct, they may need to be combined or strengthened to directly imply the method's full postcondition (e.g., $sum = k(k-1)/2$).

7 LLM-Aided Python-to-Dafny Translation

In addition to direct invariant synthesis on Dafny programs, the project includes an LLM-aided pipeline for translating of Python programs into Dafny and subsequently verifying them using the same SMT-based invariant machinery. This component is designed to separate deterministic syntax translation from semantic refinement, using large language models (LLMs) only where symbolic reasoning or specification inference is required.

The translation pipeline is structured as a sequence of clearly separated stages, ensuring that the use of LLMs remains constrained, auditable, and reproducible.

7.1 Stage 1: Deterministic Python-to-Dafny Draft Generation

The initial translation stage converts Python source code into a syntactically valid Dafny *draft* without invoking any machine learning component. This stage is implemented in `week12/python_to_dafny.py`

and is entirely algorithmic.

Python programs are parsed using the standard `ast` module. The abstract syntax tree is then traversed and emitted into Dafny code using a lightweight emitter that manages indentation and block structure. The translation supports a restricted but sufficient subset of Python, including:

- function definitions,
- variable assignments,
- `while` loops,
- `for` loops over `range`, lowered into explicit initialization and `while` loops,
- conditional statements,
- return statements, and
- basic arithmetic and Boolean expressions.

This stage intentionally avoids inferring precise specifications or invariants. Instead, it emits a conservative Dafny program augmented with *stable placeholders* that act as anchors for later refinement. For example, each method contains placeholders for `requires` and `ensures` clauses, and each loop is emitted with a trivial invariant and decreases clause:

```
while (i < n)
  // @loop_id:0
  invariant true // @inv_placeholder:0
  decreases * // @dec_placeholder:0;
{
  ...
}
```

These placeholders make the draft robust to later modification without relying on fragile parsing or structural reconstruction.

7.2 Metadata Extraction

Alongside the Dafny draft, the translator produces structured metadata that summarizes semantic information about the original Python program. This metadata includes loop identifiers, loop guards, modified variables, variables appearing in conditions, and coarse-grained type information inferred from usage.

The metadata is not used for verification directly; rather, it constrains and guides later LLM interactions by making explicit which variables and loops are relevant, thereby reducing ambiguity and hallucination in generated patches.

7.3 Stage 2: Prompt Construction and LLM Invocation

The second stage constructs a structured prompt that combines: (i) the original Python source, (ii) the generated Dafny draft, and (iii) the extracted metadata. This prompt is assembled by the orchestrator in `week12/orchestrator/run.py` using a fixed template.

The prompt explicitly requires the LLM to output *JSON only*, conforming to a predefined patch schema. The schema allows the LLM to propose refinements such as:

- method preconditions and postconditions,
- loop invariants and decreases clauses,
- type annotations, and
- small syntactic rewrites.

LLM interaction is performed with temperature set to zero, ensuring deterministic responses for a fixed prompt. To further guarantee reproducibility, all requests are cached on disk using a hash of the prompt, model, and backend configuration.

7.4 Stage 3: Patch Validation and Deterministic Application

LLM output is treated as a *proposed patch*, not as executable code. The patch is first validated against a lightweight schema to ensure structural correctness. If validation succeeds, the patch is applied deterministically to the Dafny draft using placeholder-based text replacement.

Crucially, the patching mechanism is conservative: it can only modify locations explicitly marked by placeholders or apply rewrites from a small, fixed whitelist (e.g., Boolean literal normalization). No free-form code generation or structural rewriting is permitted at this stage. As a result, applying the same patch to the same draft always yields the same Dafny candidate.

7.5 Stage 4: Verification and Repair Loop

Optionally, the resulting Dafny candidate is passed to the Dafny verifier. If verification succeeds, the pipeline terminates. If verification fails, the error output and local context are incorporated into a *repair prompt*, which is sent back to the LLM to request a refined patch.

This repair loop is bounded by a fixed iteration limit and produces a sequence of versioned artifacts (drafts, patches, verification logs). At no point is the LLM allowed to bypass verification; correctness is determined solely by Dafny.

7.6 Design Rationale and Limitations

This architecture deliberately isolates the role of the LLM. All syntactic translation, patch application, and verification steps are deterministic and auditable. The LLM is used only to propose semantic refinements—such as invariants or specifications—that are difficult to infer algorithmically but easy to validate formally.

The approach inherits limitations from both sides. The Python subset supported by the translator is intentionally restricted, and type inference is shallow. Conversely, the LLM may propose invariants that are inductive but too weak to prove the desired postcondition, requiring further repair iterations.

Despite these limitations, the pipeline demonstrates that LLM assistance can be integrated into formal verification workflows in a controlled manner, where soundness is enforced by SMT solving and verification rather than by trust in model output.

References

[1] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.

[2] Sumit Gulwani and Ashish Tiwari. Constraint-based approach for analysis of hybrid systems. In *Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 190–203. Springer, 2008.

[3] Andrey Rybalchenko. Constraint solving for program verification. In *Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2010.