An Automated Program Verifier in First Order Logic

Pritom Rajkhowa and Fangzhen Lin

Abstract In this chapter, we describe VIAP, a fully automated verifier for programs with loops. Given a program and a requirement to verify, it first translates the given program into a set of first-order axioms independent of the requirement. The requirement is then verified as a query of the translated axioms. What distinguishes VIAP is its handling of loops. The iterations are systematically encoded as inductive definitions, and the termination is axiomatised by constraints on specially introduced constants. The efficiency of VIAP comes from many simplifying techniques, including a dedicated recurrence solver to compute the closed-form solutions of inductive definitions whenever possible. As a result, VIAP is able to automatically prove some non-trivial properties of many benchmark programs that require either manually encoded loop invariants or powerful loop invariant generators. VIAP was entered at the SV-COMP 2019 competition and scored first in the ReachSafety-Arrays and ReachSafety-Recursive sub-categories of the ReachSafety category.

1 Introduction

Program verification has been a key problem in computer science since its beginning. Various approaches have been proposed for different types of programs. Despite significant progress made over the years, verification of programs with loops is still considered a challenge in the field. Here we consider a class of imperative programs composed of integer assignments, sequences, conditionals and loops. We describes the theory and implementation of a fully automated system called VIAP

Pritom Rajkhowa
The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong,
e-mail: prajkhowa@cse.ust.hk

Fangzhen Lin
The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
e-mail: flin@cse.ust.hk

for proving the safety property of these programs with respect to specifications given as assertion(s) and assumption(s).

VIAP first translates the given program to a set of first-order axioms with natural number quantifications. This part is independent of the proof obligation. It then attempts to prove safety properties from translated axioms, and assumption(s) using SMT solver Z3 [15] as the basic theorem prover.

The translation of our approach part is based on Lin [28]. It also tries to compute closed-form solutions of inductive definitions in Lin's translation using a recurrence solver called recurrence solver (RS). That we developed the motivation developing our own recurrence solver came from the fact that we need to compute closed form solution of conditional recurrences which cannot be handled by the current recurrence solvers as far as we know. The theorem proving part uses Z3 directly at first. If it fails, it tries a simple inductive proof using Z3 as the base theorem prover.

1.1 Challenges

In the following we use the example and illustrate how our system works compared to traditional approach using Hoare's logic, consider the following program for computing the integer square root of a given non-negative integer X:

```
a=0;
su=1;
t=1;
assume(X>=0);
while ( su <= X ) {
       a=a+1;
       t=t+2;
       su=su+t;
}
assert((a+1)*(a+1)>X);
assert((a)*(a)<=X);
```

Goal : The goal is to prove that assertions $(a + 1) \times (a + 1) > X$ and $a \times a \leq X$ holds at the end of the program, for every input value of non-negative integer X.

We first explain how Hoare logic is used to reason about program correctness. A Hoare triples of Hoare logic is a set of inferen rules in the following form:

$$\{P\}$$
$$\text{while(Cond) } \{Body\}$$
$$\{Q\}$$

where P and Q the pre-condition and post-condition respectively of program $while(Cond)\{Body\}$. The goal is to prove the Hoare triple. If the program contains loops for a given program, we must come up with loop invariant Inv such that

the following three conditions hold:

$$P \Rightarrow Inv$$
$$\{Inv \wedge Cond\}\ Body\ \{Inv\}$$
$$Inv \wedge \neg Cond \Rightarrow Q$$

For our running example the pre-condition P is $X \geq 0$, and the post-condition Q are $(a+1) \times (a+1) > X$ and $a \times a \leq X$, the post-condition to be proven. We need to choose suitable loop invariants Inv where Inv is represented as follows:

$$Inv = \{t = 2 \times a + 1 \wedge 4s = t \times t + 2 \times t + 1 \wedge s = (a+1) \times (a+1) \wedge s \geq t\}$$

The goal is to prove the following Hoare triple using Inv

$$(X \geq 0) \Rightarrow Inv$$
$$\{Inv \wedge su \leq X\} a = a + 1; t = t + 2; su = su + t; \{Inv\}$$
$$Inv \wedge su > X \Rightarrow (a+1) \times (a+1) > X \wedge a \times a \leq X$$

These loop invariants are non-trivial to infer.

With our approach, we first translate the program into the following set of axioms:

$$X_1 = X, a_1 = a_3(N), t_1 = t_3(N), su_1 = su_3(N)$$
$$a_3(0) = 0, \quad su_3(0) = 1, \quad t_3(0) = 1,$$
$$a_3(n+1) = a_3(n) + 1, \quad t_3(n+1) = t_3(n) + 2, \quad su_3(n+1) = su_3(n) + t_3(n) + 2,$$
$$\neg(su_3(N) \leq X), \quad (n < N) \rightarrow (su_3(n) \leq X)$$

where a_1 is the output of the program variable a, and $a_3(n)$ is a temporary function denoting the value of a after the n^{th} iteration of the while loop. Similar conventions apply to X, t, su and their subscripted versions. The constant N is introduced to denote the number of iterations of the while loop before it exits. The variable n ranges over natural numbers and is universally quantified.

By using our recurrence solver , our system computes closed-form solutions for $a_3()$, $t_3()$, and $su_3()$, eliminates them, and produces the following axioms:

$$X_1 = X, \quad a_1 = N, \quad t_1 = 2N + 1, \quad su_1 = N \times N + 2N + 1,$$
$$\neg(N \times N + 2N + 1 \leq X), (n < N) \rightarrow (n \times n + 2n + 1 \leq X).$$

With this set of axioms, Z3 can then be made to prove the following two postconditions

$$(a_1 + 1) \times (a_1 + 1) > X, \quad a_1 \times a_1 \leq X$$

given the precondition $X \geq 0$.

This is an example where all recurrence relations happen to have closed-form solutions that can be computed by our recurrence solver. Sometime the recurrences relations cannot be solved by RS. When this happens, the proof cannot be done directly by Z3 as it needs to use mathematical induction which is not builtin Z3.

The rest of this chapter is organized as follows: Section 3 presents the description of VIAP 's architecture and followed by a brief description of control flow simplification in section 4. Section 5 presents a brief description of Lin's translation. Section 6 presents recurrence solver algorithm. Section 7 presents proof strategies implemented in VIAP . Section 8 we discuss VIAP 's participation in the seven and eight editions of the International Competition on Software Verification (SV-COMP). Finally, Sections 10 and 11 discuss related work and conclusion.

2 Motivational Examples

In this example, we illustrate how closed-form solutions of conditional recurrence help VIAP verify safety property of a program. In 2.1, we present an iterative implementation of Zohar Manna's version of integer division algorithm [30]. In 2.2, we present translated set axioms generated from the input program by translator module of VIAP . In 2.3, we give an explanation of how our system automatically proves the correctness of programs by proving the assertions in the program.

2.1 An Iterative Implementation Zohar Manna's Version of Integer division algorithm

The C code snippet P with variables $\mathbf{X} = \{a, b, x, y, z\}$. The two integer variables a and b are assigned with two non-deterministic function call. The algorithm for dividing a number a by another number b. When the algorithm terminates, it coming up with a quotient x and a remainder y under the assumption of $a \geq 0$ and $b > 0$.

```
int a, b, x, y, z;
a = nondetint();
b = nondetint();
x = 0; y = 0; z = a;
assume(a >= 0);
assume(b > 0);
while(z != 0) {
    if (y + 1 == b)
    {
        x = x + 1;
        y = 0;
        z = z - 1;
    }
    else
```

```
    {
        y = y + 1;
        z = z - 1;
    }
}
assert(x * b + y == a);
```

2.2 Translation of Program

The translation module of VIAP implements Lin's [28] translation and generates the following set of axioms $\Pi_P^{\mathbf{X}}$ for program P, where $\mathbf{X} = \{a, b, x, y, z\}$.

$$
\begin{aligned}
&a_1 = nondetint_2, y_1 = y_6(N), b_1 = nondetint_3,\\
&x_1 = x_6(N), z_1 = z_6(N),\\
&y_6(0) = 0, x_6(0) = 0, z_6(0) = nondetint_2,\\
&\forall n.y_6(n+1) = ite((y_6(n)+1) = nondetint_3, 0, y_6(n)+1),\\
&\forall n.x_6(n+1) = ite((y_6(n)+1) = nondetint_3, x_6(n)+1, x_6(n)),\\
&\forall n.z_6(n+1) = z_6(n) - 1,\\
&\neg(z_6(N) \neq 0),\\
&\forall n.n < N \rightarrow z_6(n) \neq 0
\end{aligned}
$$

where a_1, b_1, z_1, x_1 and y_1 denote the output values of a, b, z, x and y, respectively, $x_6(n)$, $y_6(n)$ and $z_6(n)$ the values of x, y and z, respectively during the nth iteration of the loop. The conditional expression $ite(c, e_1, e_2)$ has value e_1 if c holds and e_2 otherwise. Also N is a natural number constant, and the last two axioms say that it is exactly the number of iterations the loop executes before exiting.

The translation of assumption results as follows:

$$nondetint_2 \geq 0 \tag{1}$$

$$nondetint_3 > 0 \tag{2}$$

The translation of assertion results as follows:

$$x_6(N) \times nondetint_3 + y_6(N) = nondetint_2 \tag{3}$$

2.3 Proving Assertion

There are three recurrence relations in the above axioms. The recurrence relation for $z_6(n)$ is non-conditional which can be solved by our recurrence solving tool module RS, which yields the closed-form solution $z_6(n) = nondetint_2 - n$ which can then

be used to simplify the rest of axioms by substituting $z_6(n) = nondetint_2 - n$ in $\Pi_P^{\mathbf{X}}$ and then eliminate $z_6()$ from $\Pi_P^{\mathbf{X}}$. The resulted set of axioms are as follows:

$$
\begin{aligned}
&a_1 = nondetint_2, y_1 = y_6(N), b_1 = nondetint_3,\\
&x_1 = x_6(N), z_1 = nondetint_2 - N,\\
&y_6(0) = 0, x_6(0) = 0,\\
&\forall n.y_6(n+1) = ite((y_6(n)+1) = nondetint_3, 0, y_6(n)+1),\\
&\forall n.x_6(n+1) = ite((y_6(n)+1) = nondetint3, x_6(n)+1, x_6(n)),\\
&\neg((nondetint_2 - N) \neq 0),\\
&\forall n.n < N \rightarrow (nondetint_2 - n) \neq 0
\end{aligned}
$$

Then RS can find the following closed-form solutions of $x_6(n)$ and $y_6(n)$ using applying conditional recurrence solver which we have described in the section 6.

$$
\begin{aligned}
&y_6(n) = ite(0 \leq n \wedge n < nondetint_3, n, ite(n = nondetint_3, 0, n - nondetint_3)),\\
&x_6(n) = ite(0 \leq n \wedge n < nondetint_3, 0, 1).
\end{aligned}
$$

After computing the closed-form solutions for $x_6()$ and $y_6()$ by RS, it substituting them in $\Pi_P^{\mathbf{X}}$ and then VIAP eliminates them, and updated $\Pi_P^{\mathbf{X}}$ which is represented by the following axioms:

$$
\begin{aligned}
&a_1 = nondetint_2, z_1 = (nondetint_2 - N), b_1 = nondetint_3,\\
&y_1 = ite(0 \leq N \wedge N < nondetint_3, N, ite(N = nondetint_3, 0, N - nondetint_3)),\\
&x_1 = ite(0 \leq N \wedge N < nondetint_3, 0, 1),\\
&\neg((nondetint_2 - N) \neq 0),\\
&\forall n.n < N \rightarrow (nondetint_2 - n) \neq 0
\end{aligned}
$$

After substituing closed form solution of $x_6()$ and $y_6()$ in assertion 3 results the following :

$$
\begin{aligned}
ite(0 \leq N \wedge N < nondetint_3, 0, 1) \times nondetint_3 + \\
ite(0 \leq N \wedge N < nondetint_3, N, \\
ite(N = nondetint_3, 0, N - nondetint_3)) = nondetint_2. \quad (4)
\end{aligned}
$$

With this set of axioms $\Pi_P^{\mathbf{X}}$, SMT solvers like Z3 can then be made to prove the assertion 4 with respect to assumption $nondetint_3 > 0$ and $nondetint_2 \geq 0$.

State-of-the-art automatic verification tool such as ICRA[25], VeriAbs [11], UAutomizer [21], Seahorn [20], and CPAChecker [6] failed to prove the assertion.

3 Architecture

Figure 1 depicts the VIAP architecture. It consists of seven major modules control flow simplification, translator, recurrence solver (RS), proof engine, SMT solver, *SymPy* and result.

- **Front-End** : The system accepts a program written in C (C99 language) as input and pre-process the input program for simplification of control flow. Then the simplified pre-process program is translated to first order axioms. The recurrence solver (RS) solves the recurrence relations generated during the translation if closed-form solutions are available.
- **Back-End** : The system takes the set of translated first-order axioms. The resulting axioms are also simplified using Sympy, then converts them to Z3 input. Then the proof engine applies different strategies and tries to prove post-conditions in Z3 [15]. Using output of Z3, the result module constructs the suitable witness.

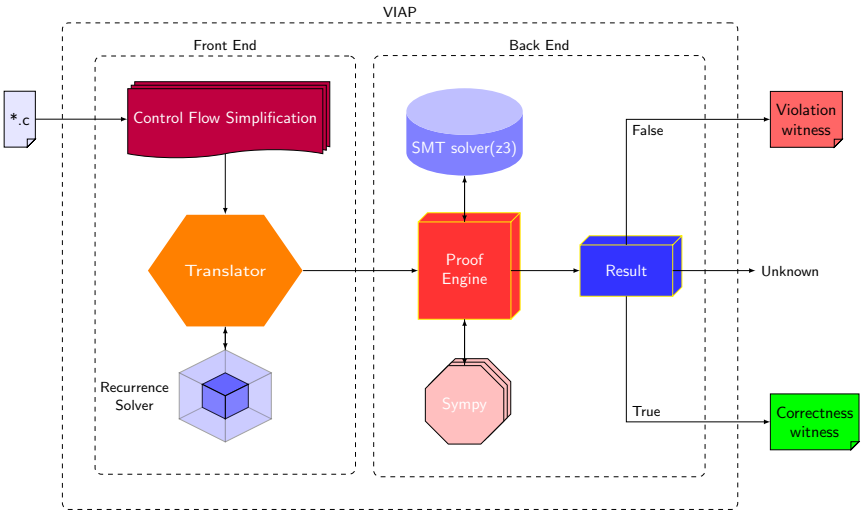

Fig. 1: The VIAP architecture

4 Control Flow Simplification

Our translator is designed for structured programs with whlie loop only. The system translates other constructs such as for-loop and do-while-loop to their equivalent while-loop. The system translates unstructured programs (with *goto*, *break*

and *continue*) to their equivalent structured program. The system has also implemented the goto-removal algorithm proposed in [16]. For instances do-while loop (`do P1 while B`) is converted to equivalent while loop (`P1; while B do P1`). Similarly one more example, program segment with goto (`P1; if C then goto L1; P2; L1: P3`) is converted to equivalent structured program segment (`P1; if !C then P2; P3`).

5 Translation

Our translator consider programs in the following language:

```
E ::= array(E,...,E) |
operator(E,...,E)
B ::= E = E |
boolean-op(B,...,B)
P ::= array(E,...,E) = E |
if B then P else P |
P; P |
while B do P
```

where the tokens `E, B, P` stand for integer expressions, Boolean expressions and programs, respectively. The token `array` stands for program variables, and the tokens `operator` and `boolean-op` stand for built-in integer functions and Boolean functions, respectively. Notice that for `array`, if its arity is 0, then it stands for an integer program variable. Otherwise, it is an array variable. Notice also that while the notation `array[i][j]` is commonly used in programming languages to refer to an array element, we use the notation `array(i,j)` here which is more common mathematics and logic.

Given a program P, and a language $\mathbf{X}$, our system generates a set of first-order axioms denoted by $\Pi_P^{\mathbf{X}}$ that captures the changes of P on $\mathbf{X}$. Here, a language means a set of functions and predicate symbols. For $\Pi_P^{\mathbf{X}}$ to be correct, $\mathbf{X}$ needs to include all program variables in P as well as any functions and predicates that can be changed by P. The axioms in the set $\Pi_P^{\mathbf{X}}$ are generated inductively on the structure of P. The definition of each induction rule in [28] is as follows:

Definition 1 When `P` is `P1; P2` then $\Pi_P^{\mathbf{X}}$ is the set of following axioms:

$\varphi(\mathbf{X_1}/\mathbf{Y}) \rightarrow \varphi$, for each $\varphi \in \Pi_{P1}^{\mathbf{X}}$,

$\varphi(\mathbf{X}/\mathbf{Y}) \rightarrow \varphi$, for each $\varphi \in \Pi_{P2}^{\mathbf{X}}$

where $\mathbf{Y} = (Y^1, ..., Y^k)$ is a tuple of new function symbols such that each Y^i is of the same arity as X^i in $\mathbf{X}$, $\varphi(\mathbf{X_1}/\mathbf{Y})$ is the result of replacing in φ each occurrence of X_1^i by Y^i, and similarly for $\varphi(X/\mathbf{Y})$. By renaming them if necessary, we assume here that $\Pi_{P1}^{\mathbf{X}}$ and $\Pi_{P2}^{\mathbf{X}}$ have no common program variables except those in $\mathbf{X}$.

Definition 2 When `P` is `if B then P1 else P2` then $\Pi_P^{\mathbf{X}}$ is the set of following axioms:

$B \rightarrow \varphi$, for each $\varphi \in \Pi_{P1}^{\mathbf{X}}$,

$\neg B \rightarrow \varphi$, for each $\varphi \in \Pi_{P1}^{\mathbf{X}}$,

Definition 3 When `P` is `while B do P1` then $\Pi_P^{\mathbf{X}}$ is the set of following axioms: $\varphi(n)$, for each $\varphi \in \Pi_{P1}^{\mathbf{X}}$,

$X^i(\mathbf{x}) = X^i(\mathbf{x}, 0)$, for each $X^i \in \mathbf{X}$

$smallest(N, n, \neg B(n))$,

$X_1^i(\mathbf{x}) = X^i(\mathbf{x}, N)$, for each $X^i \in X$ where n is a new natural number variable not

already in φ, and N a new constant not already used in $\Pi_{P1}^{\mathbf{X}}$. For each formula or term α, $\alpha(n)$ is defined inductively as follows: it is obtained from α by performing the following recursive substitutions:

- for each $X^i \in \mathbf{X}$, replace all occurrences of $X_1^i(e_1, ..., e_k)$ by $X^i(e_1(n), ..., e_k(n), n+1)$, and
- for each program variable X in α, replace all occurrences of $X(e_1, ..., e_k)$ by $X(e_1(n), ..., e_k(n), n)$. Notice that this replacement is for every program variable X, including those not in $\mathbf{X}$.

$smallest(N, n, \neg B(n))$ is a shorthand for the following formula

$$\neg B(N), \tag{5}$$

$$\forall n.n < N \rightarrow B(n) \tag{6}$$

We consider arrays as first-order objects that can be parameters of functions, predicates, and can be quantified over. In first-order logic, this means that we have sorts for arrays, and one sort for each dimension. In the following, we denote by *int* the integer sort, and $array_k$ the k-dimensional array sort, where $k \geq 1$.

To denote the value of an array at some indices, for each $k \geq 1$, we introduce a special function named $d\mathrm{k}array$ of arity k:

$$d\mathrm{k}array : \ array_k \times int^k \rightarrow int,$$

as we consider only integer - valued array elements. Thus, $d1array(a, i)$ denotes the value of a one-dimensional array a at index i, i.e. $a[i]$ under a conventional notation, and $d2array(b, i, j)$ stands for $b[i][j]$ for two-dimensional array b. We can also introduce a function to denote the size of an array.

Recall that we generate a set of axioms for a program P under a language $\mathbf{X}$. The generated set of axioms captures the changes of P on $\mathbf{X}$, so $\mathbf{X}$ needs to include all functions and predicates that can be changed by P. Therefore, if a program makes changes to, say a two-dimensional array, then $\mathbf{X}$ must include $d2array$.

When we translate a program to first-order axioms, we need to convert expressions in the program to terms in first-order logic. This is straightforward, given how we have decided to represent arrays. For example, if E is $a(1,2) + b(1)$, where a is a two-dimensional array and b a one-dimensional array, then $\hat{E}$, the first-order term that corresponds to E, is $d2array(a,1,2) + d1array(b,1)$.

We are now ready to describe how we generate axioms for assignments, First, for integer variable assignments:

Definition 4 If P is `V = E`, and $V \in \mathbf{X}$, then $\Pi_P^{\mathbf{X}}$ is the set of the following axioms:

$$\forall \mathbf{x}.X_1(\mathbf{x}) = X(\mathbf{x}), \text{ for each } X \in \mathbf{X} \text{ that is different from } V,$$
$$V_1 = \hat{E}$$

where `V` is a non-array program variable, for each $X \in \mathbf{X}$, we introduce a new symbol X_1 with the same arity standing for the value of X after the assignment, and $\hat{E}$ is the translation of the expression `E` into its corresponding term in logic as described above.

For example, if P_1 is

```
I = a(1,2)+b(1)
```

and $\mathbf{X}$ is $\{I, a, b, d1array, d2array\}$ (a and b are for the two array variables in the assignment, respectively), then $\Pi_{P_1}^{\mathbf{X}}$ is the set of following axioms:

$$I_1 = d2array(a,1,2) + d1array(b,1),$$
$$a_1 = a,$$
$$b_1 = b,$$
$$\forall x,i.d1array_1(x,i) = d1array(x,i),$$
$$\forall x,i,j.d2array_1(x,i,j) = d2array(x,i,j).$$

Again, we remark that we assume all array accesses are legal. Otherwise, we would need axioms like the following to catch array errors:

$$\neg in\text{-}bound(1,b) \rightarrow arrayError,$$
$$\neg in\text{-}bound((1,2),a) \rightarrow arrayError,$$

where $in\text{-}bound(\mathbf{i}, array)$ means that the index $\mathbf{i}$ is within the bound of $array$, and can be defined using array sizes.

Definition 5 If P is `V(e1,e2,...,ek) = E`, then $\Pi_P^{\mathbf{X}}$ is the set of the following axioms:

$$\forall \mathbf{x}.X_1(\mathbf{x}) = X(\mathbf{x}), \text{ for each } X \in \mathbf{X} \text{ which is different from } d\text{k}array,$$
$$d\text{k}array_1(x,i_1,...,i_k) =$$
$$ite(x = V \wedge i_i = \hat{e_1} \wedge \cdots \wedge i_k = \hat{e_k}, \hat{E}, d\text{k}array(x,i_1,...,i_k)),$$

where `V` is a array program variable, $ite(c, e, e')$ is the conditional expression: if c then e else e'.

For example, if P_2 is `b(1)=a(1,2)+b(1)`, and $\mathbf{X}$ is $\{I, a, b, d1array, d2array\}$, then $\Pi^{\mathbf{X}}_{P_2}$ is the set of following axioms:

$$
\begin{aligned}
&I_1 = I,\\
&a_1 = a,\\
&b_1 = b,\\
&\forall x, i.d1array_1(x, i) =\\
&\qquad ite(x = b \wedge i = 1, d2array(a, 1, 2) + d1array(b, 1), d1array(x, i)),\\
&\forall x, i, j.d2array_1(x, i, j) = d2array(x, i, j).
\end{aligned}
$$

Notice that $b_1 = b$ means that while the value of b at index 1 has changed, the array itself as an *object* has not changed. If we have array assignments like `a=b` for array variables `a` and `b`, they will generate axioms like $a_1 = b$.

We now give two simple examples of how the inductive cases work. Consider P_3 which is the sequence of first P_1 then P_2:

```
I = a(1,2)+b(1);
b(1)=a(1,2)+b(1)
```

The axiom set $\Pi^{\mathbf{X}}_{P_3}$ is generated from $\Pi^{\mathbf{X}}_{P_1}$ and $\Pi^{\mathbf{X}}_{P_2}$ by introducing some new symbols to connect the output of P_1 with the input of P_2:

$$
\begin{aligned}
&I_2 = d2array(a, 1, 2) + d1array(b, 1),\\
&a_2 = a,\\
&b_2 = b,\\
&\forall x, i.d1array_2(x, i) = d1array(x, i),\\
&\forall x, i, j.d2array_2(x, i, j) = d2array(x, i, j),\\
&I_1 = I_2,\\
&a_1 = a_2,\\
&b_1 = b_2,\\
&\forall x, i.d1array_1(x, i) =\\
&\qquad ite(x = b_2 \wedge i = 1, d2array_2(a_2, 1, 2) + d1array_2(b_2, 1), d1array_2(x, i)),\\
&\forall x, i, j.d2array_1(x, i, j) = d2array_2(x, i, j),
\end{aligned}
$$

where $I_2, a_2, b_2, d1array_2, d2array_2$ are new symbols to connect P_1's output with P_2's input. If we do not care about the intermediate values, these temporary symbols can often be eliminated. For this program, eliminating them yields the following set of axioms:

$$
\begin{aligned}
&I_1 = d2array(a, 1, 2) + d1array(b, 1),\\
&a_1 = a,\\
&b_1 = b,\\
&\forall x, i.d1array_1(x, i) =\\
&\qquad ite(x = b \wedge i = 1, d2array(a, 1, 2) + d1array(b, 1), d1array(x, i)),\\
&\forall x, i, j.d2array_1(x, i, j) = d2array(x, i, j).
\end{aligned}
$$

The most important feature of the approach in [28] is in the translation of loops to a set of first-order axioms. The main idea is to introduce an explicit counter for loop iterations and an explicit natural number constant to denote the number of iterations the loop executes before exiting. It is best to illustrate by a simple example. Consider the following program P_4:

```
while I < M { I = I+1; }
```

Let $\mathbf{X} = \{I, M\}$. To compute $\Pi^{\mathbf{X}}_{P_4}$, we need to generate first the axioms for the body of the loop, which in this case is straightforward:

$$
\begin{aligned}
&I_1 = I + 1,\\
&M_1 = M
\end{aligned}
$$

Once the axioms for the body of the loop are computed, they are turned into inductive definitions by adding a new counter argument to all functions and predicates that may be changed by the program. For our simple example, we get

$$\forall n.I(n + 1) = I(n) + 1, \tag{7}$$

$$\forall n.M(n + 1) = M(n), \tag{8}$$

where the quantification is over all natural numbers. We then add the initial case, and introduce a new natural number constant N to denote the terminating index:

$$
\begin{aligned}
&I(0) = I \wedge M(0) = M,\\
&I_1 = I(N) \wedge M_1 = M(N),\\
&\neg(I(N) < M(N)),\\
&\forall n.n < N \rightarrow I(n) < M(n).
\end{aligned}
$$

One advantage of making counters explicit and quantifiable is that we can then either compute closed-form solutions to recurrences like (7) or reason about them using mathematical induction. This is unlike proof strategies like k-induction where the counters are hard-wired into the variables. Again, for more details about this approach, see [28] which has discussions about related work as well as proofs of the correctness under an operational semantics.

6 Recurrence Solver (RS)

Our recurrence solver RS takes a set of recurrences and other constraints, returns a set of closed-form solutions it found for some of the recurrences and the remaining recurrences and constraints simplified using the computed closed-form solutions. We do not give a formal definition of recurrences and constraints here. Informally recurrences are just axioms in a formal language like first-order logic. This informal description should be enough to understand the ideas and algorithms used in our solver.

Algorithm (1) outlines our recurrence solver. It makes use of three sub-solvers and two helper functions:

- A base non-conditional recurrence solver ($NCRS$, Algorithm (2)) that is called rsolve() in SymPy is for computing closed-form solutions of non-conditional recurrences: given a set Π of recurrences and other constraints, $NCRS$ (Π) returns three sets C, Δ, and a new Π, where C is the set of closed-form solutions computed; Δ the set of non-conditional recurrences that it tried but failed to find a closed-form solution; and Π the set of remaining recurrences.
- A mutual recurrence solver (MRS, Algorithm (3)) for computing the closed-form solution of a specific type of mutual recurrences: given a set Π of recurrences, MRS (Π) returns three sets C, Δ, and a new Π, where C is the set of closed-form solutions computed; Δ the set of mutual recurrences that it tried but failed to find a closed-form solution; and Π the set of remaining recurrences.
- A conditional recurrence solver (CRS, Algorithm (4)) for computing the closed-form solution of several classes of conditional recurrences: given a set Π of recurrences; CRS (Π) returns three sets C, Δ, and a new Π, where C is the set of closed-form solutions computed; Δ the set of conditional recurrences that it tried but failed to find a closed-form solution; and Π the set of remaining recurrences.
- A simplifier (SC) simplify conditional recurrences: given a set Π of recurrences and SC(Π) returns a new simplified Π. SC uses the SMT solver to simplify a condition.
- We have presented the detailed proof of termination of all the algorithms 1 (proposition (11)), 2 (proposition (8)), 2 (proposition (9)) and 4 (proposition (10)) in subsection 12.2.

6.1 The Non-Conditional Recurrence Solver ($NCRS$)

This sub-solver referred to as SymPy's rsolve() function to compute two types of non-conditional recurrences. The pseudo code is given as Algorithm (2) which makes use of the following functions:

- $selectNC(\Pi)$ - it returns a recurrence $\langle \sigma, \sigma_0, n \rangle$ from Π, where σ is the recurence, σ_0 the initial condition, and n the variable involved in the recurrence. The selected σ must be the non-conditional of the form of either

Input: Π – a set of recurrences and constraints
Output: C – the set of closed-form solutions computed; Π – the remaining set of recurrences and constraints simplified using C; Δ – the set of selected but unsolved recurrence equations

$C \leftarrow \emptyset$;
$\Delta \leftarrow \emptyset$;
while *true* **do**
  $C_1, \Delta_1, \Pi \leftarrow NCRS(\Pi)$;
  $C_2, \Delta_2, \Pi \leftarrow MRS(\Pi)$;
  $C_3, \Delta_3, \Pi \leftarrow CRS(\Pi)$;
  $C' \leftarrow C_1 \cup C_2 \cup C_3$;
  $\Delta' \leftarrow \Delta_1 \cup \Delta_2 \cup \Delta_3$;
  if $C' == \emptyset$ *or* $\Delta' == \emptyset$ **then**
    return $C,\ \Pi \cup \Delta$;
  else
    $C \leftarrow C \cup C'$;
    $\Delta \leftarrow \Delta \cup \Delta'$;
  end
end

Algorithm 1: Recurrence Solver (RS)

$$X(n+1) = f(X(n), n),$$

where $f(x, y)$ is a polynomial function of x and y, or,

$$X(n+1) = X(n) + f(n) + A_1 F_1(n) + \cdots + A_k F_k(n),$$

where $f(n)$ is a polynomial function in n, A_i's are constants, and F_i's are function symbols.

- $rsolve(\sigma, \sigma_0, n)$ calls SymPy's recurrence solving function after converting from our language to SmpPy's. SymPy [42] can find the closed-form solution of the recurrence equation of type C-finite and P-finite.
- SR simplifies both Π and Δ using the set computed closed-form solution C. This function is described in more details later.

6.2 The Mutual Recurrence Solver (MRS)

Definition 6 The type of mutually recursive functions that our system can solve is defined by the following selection function:

$$selectMR(\Pi) = \langle \mathbf{œ}, \mathbf{œ_0}, n \rangle,$$

where $\mathbf{œ}$, $\mathbf{œ_0}$ are sets of equations in Π of the form: for some $h > 1$, $\mathbf{œ}$ consists of

$$X_i(n+1) = A * (X_1(n) + \ldots + X_h(n)) + C_i, \quad \text{for } 1 \leq i \leq h, \tag{9}$$

Input: Π – a set of recurrences and constraints
Output: C – the set of closed-form solutions computed; Δ – the set of selected but unsolved recurrence equations; Π – the remaining set of recurrences and constraints simplified using C;

$C \leftarrow \emptyset$;
$\Delta \leftarrow \emptyset$;
$\langle \sigma, \sigma_0, n \rangle = selectNC(\Pi)$;
while $\langle \sigma, \sigma_0, n \rangle \neq \emptyset$ **do**
  $f_\sigma \leftarrow rsolve(\sigma, \sigma_0, n)$;
  if $f_\sigma \neq \emptyset$ **then**
    $C \leftarrow C \cup f_\sigma$;
    $\Pi \leftarrow \Pi - \{\sigma, \sigma_0\}$;
    $\Pi, \Delta \leftarrow SR(f_\sigma, \Pi, \Delta)$;
  else
    $\Delta \leftarrow \Delta \cup \{\sigma, \sigma_0\}$;
  end
  $\langle \sigma, \sigma_0, n \rangle = selectNC(\Pi)$;
end
return C, Δ, Π

Algorithm 2: The Non-Conditional Recurrence Solver ($NCRS$)

where A and C_i are constants, and $\mathbf{œ_0}$ consists of

$$X_i(0) = E_i, \quad \text{for } 1 \leq i \leq h, \tag{10}$$

where each E_i is a constant expression.

The algorithm (3) below outlines our solver for these mutually recursive functions. The function $\mathcal{M}$ in the algorithm works as follows: given the input recurrences (9), it generates the following closed-form solutions for $X_1, ..., X_h$:

$$X_i(n) = ite(n = 0, E_i, ite(n = 1, A(E_1 + .. + E_h) + C_i, A^n h^n (E_1 + ... + E_h) + (C_1 + .. + C_h)\frac{A(1 - A^n h^n)}{1 - A * h} + C_i)) \tag{11}$$

The proof of correctness of the proposed closed-form solution is presented in Proposition (1) in subsection 12.1.

It is possible to compute closed-form solutions of other mutually recursive equations. For example, consider the following:

$$f(n + 1) = 2f(n) + 3g(n), f(0) = 1, g(n + 1) = 5f(n) + 6g(n), \quad g(0) = 2.$$

The recurrences can be manipulated to yield

$$f(n + 2) = 8f(n + 1) + 3f(n), \ f(1) = 8, \ f(0) = 1.$$

These second order recurrences can be solved using SymPy [42] as it has implemented algorithms from Abramov, Bronstein and Petkovsek [1] [35]. We leave this as a future work.

Input: Π – a set of recurrences and other constraints
Output: C – the set of closed-form solutions computed; Δ – the set of selected but unsolved recurrence equations; Π – the remaining set of recurrences and constraints simplified using C

$C \leftarrow \emptyset$;
$\Delta \leftarrow \emptyset$;
$\langle \mathfrak{E}, \mathfrak{E}_0, n \rangle = selectMR(\Pi)$;
while $\langle \mathfrak{E}, \mathfrak{E}_0, n \rangle \neq \emptyset$ **do**
  $\mathbf{f}_{\mathfrak{E}} \leftarrow \mathcal{M}(\mathfrak{E}, \mathfrak{E}_0, n)$;
  if $\mathbf{f}_{\mathfrak{E}} \neq \emptyset$ **then**
    $C \leftarrow C \cup \mathbf{f}_{\mathfrak{E}}$;
    $\Pi \leftarrow \Pi - \mathfrak{E} - \mathfrak{E}_0$;
    $\Pi, \Delta \leftarrow SR(\mathbf{f}_{\mathfrak{E}}, \Pi, \Delta)$;
  else
    $\Delta \leftarrow \Delta \cup \mathfrak{E} \cup \mathfrak{E}_0$;
  end
  $\langle \mathfrak{E}, \mathfrak{E}_0, n \rangle = selectMR(\Pi)$;
end
return C, Δ, Π;

Algorithm 3: Mutual Recurrence Solver (MRS)

6.3 Conditional Recurrence (CRS)

We now consider our conditional recurrence solver $CRS(\Pi)$. A conditional recurrence here is one that is returned by the following seletion function:

$$selectCR(\Pi) = \langle \sigma, X(0) = E, n \rangle,$$

where E is a constant expression and σ is a conditional equation in Π of the following form:

$$X(n+1) = ite(\theta_1, f_1(X(n), n), ite(\theta_2, f_2(X(n), n)..., \\ ite(\theta_h, f_h(X(n), n), f_{h+1}(X(n), n))) \quad (12)$$

where $\theta_1, \theta_2, ..., \theta_h$ are boolean expressions, and $f_1(x, y)$, $f_2(x, y)$, ..., $f_{h+1}(x, y)$ are polynomial functions of x and y.

Algorithm (4) below outlines our solver $CRS(\Pi)$ for conditional recurrence. It relies on the function $C(\sigma, \sigma_0, n)$ which returns f_σ where f_σ is either a closed-form solution for $X(n)$ or when it fails to find the closed solution. $C(\sigma, \sigma_0, n)$ considers following five cases of σ.

Definition 7 (Type 1) A conditional recurrence σ of the form (12) is called of type 1 if all the conditions θ_i, $1 \leq i \leq h$, in it are independent of the recurrence variable n. Given such a type 1 conditional recurrence σ, $C(\sigma, X(0) = E, n)$ works as follows:

For each $i \in [1, h+1]$, introduce a new function $g_i(n)$ and call $rsolve(g_i(n+1) = f_i(g_i(n), n), g_i(0) = E, n)$. If one of them does not return a closed-form solution, then return $\emptyset$ for $C(\sigma, X(0) = E, n)$. Otherwise, suppose $g_i(n) = E_i(n)$ is the returned

Input: Π – a set of recurrences and other constraints.
Output: C – the set of closed-form solutions computed; Δ – the set of selected but unsolved recurrence equations; Π – the remaining set of recurrences and constraints simplified using C, as well as possibly some new constraints added.

$C \leftarrow \emptyset$;
$\Delta \leftarrow \emptyset$;
$\langle \sigma, \sigma_0, n \rangle = selectCR(\Pi)$;
while $\langle \sigma, \sigma_0, n \rangle \neq \emptyset$ **do**
  $f_\sigma, \mathbf{\Phi} \leftarrow C(\sigma, \sigma_0, n)$;
  if $f_\sigma \neq \emptyset$ **then**
    $C \leftarrow C \cup f_\sigma$;
    $\Pi \leftarrow \Pi \cup \mathbf{\Phi}$;
    $\Pi \leftarrow \Pi - \{\sigma, \sigma_0\}$;
    $\Pi, \Delta \leftarrow SR(f_\sigma, \Pi, \Delta)$;
  else
    $\Delta \leftarrow \Delta \cup \{\sigma, \sigma_0\}$;
  end
  $\langle \sigma, \sigma_0, n \rangle = selectCR(\Pi)$;
end
return C, Δ, Π;

Algorithm 4: Conditional Recurrence Solver (CRS)

closed-form solution for g_i, then return f_σ, where f_σ is

$$X(n) = ite(\theta_1, E_1(n), ite(\theta_2, E_2(n), ...ite(\theta_h, E_h(n), E_{h+1}(n))..) \tag{13}$$

The proof of correctness of this closed-form solution is shown in Proposition (2) in subsection 12.1.

Example

$$j_6(n_1 + 1) = ite(nondet\_int_2 > 0, j_6(n_1) + (n_1 + 1) + 1, j_6(n_1) + (n_1) + 1)$$
$$j_6(0) = 0$$

The above conditional recurrence of j_6 satisfies both conditions of type 1. C returns the following closed-form solution for the conditional recurrence of j_6 :

$$j_6(n_1) = ite(nondet\_int_2 > 0, n_1 * (n_1 + 3)/2, n_1 * (n_1 + 2)/2)$$

Definition 8 (Type 2) A conditional recurrence σ of the form (14) is called type 2

$$X(n + 1) = ite(n = c_1, C_1, ite(n = c_2, C_2..., ite(n = c_h, C_h, f(X(n), n))) \tag{14}$$

where $c_1, c_2, ...c_h$ and $C_1, C_2, .., C_h$ are integers constants. Given such a type 2 conditional recurrence σ, $C(\sigma, X(0) = E, n)$ works as follows:

Construct a map $M = \{X(0) : E, X(e_1) : E_1, ..., X(e_h) : E_h)\}$ which contains an entry of the form $X(e_i) = E_i$ for each condition $n = e_i$ and introduce a new function $g(n)$. Then call $rsolve(g(n + 1) = f(g(n), n), M, n)$. If the function call fails to find the

closed form solution of g, then return $\emptyset$ for $C(\sigma, X(0) = E, n)$. Otherwise, suppose $g(n) = E(n)$ is the returned closed-form solution for g, then return f_σ, where f_σ is $g(n) = E(n)$.

Example

$$f_1(0) = 1, f_1(n_1 + 1) = ite(n_1 = 0, 1, ite(n_1 = 1, 1, n_1 * f_1(n_1)))$$

The above conditional recurrence of f_1 satisfies the condition of type 2. C returns the closed-form solution $f_1(n_1) = n_1!$ for the conditional recurrence of f_1.

Definition 9 (Type 3) A conditional recurrence σ of the form (12) is of type 3 if each condition θ_i, $1 \leq i \leq h$, contains only arithmetic comparisons $>$ or $<$, polynomial functions of n, and does not mention $X(n)$.

Given such a type 3 recurrence, our system tries to compute the ranges of θ_i as follows. For each $1 \leq i \leq h$, let

$$\phi_i = \neg\theta_1 \wedge \cdots \wedge \neg\theta_{i-1} \wedge \theta_i.$$

Thus ϕ_i is the condition for $X(n + 1)$ to take the value $f_i(X(n), n)$. Our system then tries to compute h constants $C_1, ...C_h$ such that

$$0 = C_0 < C_1 < \cdots < C_h,$$

and for each $i \leq h$,

$$\forall n.C_{i-1} \leq n < C_i \rightarrow \phi_{\pi(i)}(n), \forall n.n \geq C_i \rightarrow \neg\phi_{\pi(i)}(n).$$

where π is a permutation on $\{1, ..., n\}$. The computation of these constants and the associated permutation π is done using an SMT solver: it first computes k for which $\phi_k(0)$ holds, and then asks the SMT solver to find a model of

$$\forall n.0 \leq n < C \rightarrow \phi_k(n), \forall n.n \geq C \rightarrow \neg\phi_k(n).$$

If it returns a model, then set $C_1 = C$ in the model, and let $\pi(1) = k$, and the process continues until all C_i are computed. If this fails at any point, then the module aborts with $\emptyset$ as the return.

Once the system suceeds in computing these constants, it introduces a new function g_i for each $1 \leq i \leq h + 1$, and calls

$$rsolve(g_i(n + 1) = f_{\pi(i)}(g_i(n), n), g_i(0) = E_{\pi(i)-1}(n/C_{\pi(i)-1}).$$

If any of these calls does not return a closed-form solution, the procedure aborts and returns $\emptyset$. Now suppose for each g_i, $rsolve()$ returns a closed-form solution E_i: $g_i(n) = E_{\pi(i)}(n)$, then the system returns the following closed-form solution:

$$X(n) = ite(C_0 \leq n < C_1, E_1(n), ite(n < C_2, E_2(n), ..., \\ ite(n < C_h, E_h(n), E_{h+1}(n))..) \tag{15}$$

The proof of correctness of this closed-form solution is shown in Proposition (3) in subsection 12.1.

Example

$$X(n+1) = ite(7 \leq n < 10, X(n)+2, ite(n \geq 10, X(n)-2, ite(n < 7, X(n)+1, X(n)+3))))$$
$$X(0) = 0$$

The above conditional recurrence of X satisfies the condition of type 3 conditional recurrence equation. C returns the following closed-form solution for the conditional recurrence of X.

$$X(n) = ite(0 \leq n < 7, n, ite(n < 10, 7 + 2 * (n-7), 13 - 2 * (n-10)))$$

Definition 10 (Type 4) When input conditional recurrence equation σ is either in the form of the equation (16) or (17) is called of type 4

$$X(n+1) = ite(f(n)\%c = d, X(n) \pm A, X(n) \pm B) \tag{16}$$
$$X(n+1) = ite(f(n)\%c \neq d, X(n) \pm A, X(n) \pm B) \tag{17}$$

Such that $c > 0, d \geq 0, d < c$ where c, d, A, B are integer constants and $f(n)$ is polynomial function of n. Then $C(\sigma, X(0) = E, n)$ return f_σ, where $C = d - f(0)\%c$ and C is an integer.

- f_σ is of the following form when input equation σ is in the form of the equation (16)

$$\begin{aligned} X(n) = ite(&0 \leq n \leq C, E + n \times B, \\ &ite(n = C+1, E + A + C \times B, \\ &\quad E + A + C \times B + (\lfloor (n - C - 1)/c \rfloor) \times (\pm A) \\ &\qquad +(n - C - 1 - \lfloor n - C - 1/c \rfloor) \times (\pm B))) \end{aligned} \tag{18}$$

  The proof of correctness of this closed-form solution is presented in Proposition (4) in subsection 12.1.
- f_σ is of the following when input equation σ is in the form of the equation (17)

$$\begin{aligned} X(n) = ite(&n \leq C, E + n \times A, \\ &ite(n = C+1, E + C \times A + B, \\ &\quad E + C \times A + B + (\lfloor (n - C - 1)/c \rfloor) \times (\pm B) \\ &\qquad +(n - C - 1 - \lfloor n - C - 1/c \rfloor) \times (\pm A))) \end{aligned} \tag{19}$$

  The proof of correctness of this closed-form solution is presented in Proposition (5) in subsection 12.1.

Example Lets consider following conditional recurrence of type 4

$$y_3(0) = 0, y_3(n_1 + 1) = ite(n_1\%3 = 0, y_3(n_1) + A, y_3(n_1) + B)$$

The resulting closed-form solution of the conditional recurrence equations are produced by C .

$$y_3(n_1) = \lfloor (n_1/3) \rfloor \times A + (n_1 - \lfloor (n_1/3) \rfloor) \times B$$

Definition 11 (Type 5) A conditional recurrence σ of the form (12) is of type 5 if it is of the following form (20):

$$X(n + 1) = ite(\theta, X(n) \pm A, X(n) \mp B), \tag{20}$$

where θ contains only arithmetic comparisons $>$ or $<$, polynomial functions of $X(n)$ and n. A and B are integer constants such that for some h, either $A = B \times h$ or $B = A \times h$. Then it checks whether $\theta(0)$ holds, and then asks the SMT solver to find a model of

$$\forall n.0 \leq n < C \rightarrow \theta(X(n)/(E + n \times A)) \wedge \neg\phi(X(n)/(E + C \times A))$$

If it returns a model, then $C(\sigma, X(0) = E, n)$ returns f_σ as following:

- If $A = B \times h$ for some h, then f_σ is

$$\begin{aligned} X(n) = ite(&0 \leq n < C, E \pm n \times A, \\ &ite((n - C)\%(h + 1) = 0, E \pm C \times A, \\ &\quad E \pm C \times A \mp (((n - C)\%(h + 1))) \times B)) \end{aligned} \tag{21}$$

  The proof of correctness of this closed-form solution is in Proposition (6) in Subsection 12.1.
- If $B = A \times h$ for some h, then f_σ is

$$\begin{aligned} X(n) = ite(&0 \leq n < C, E \pm n \times A, \\ &ite((n - C)\%(h + 1) = 0, E \pm C \times A, \\ &\quad ite((n - C)\%(h + 1) = 1, E \pm C \times A \mp B, \\ &\quad\quad E \pm C \times A \mp B \pm (((n - C)\%(h + 1)) - 1) \times A))) \end{aligned} \tag{22}$$

  The proof of correctness of this closed-form solution is in Proposition (7) in Subsection 12.1.

Example Consider the following conditional recurrence equation of type 5:

$$y_3(0) = 0, y_3(n + 1) = ite(y_3(n) \leq 50, y_3(n) + 1, y_3(n) - 1)$$

The resulting closed form solution of the conditional recurrence equations are produced by C .

$$y_3(n) = ite(0 \leq n < 50, n, ite((n - 50)\%2 = 0, 50, 49))$$

6.4 Substitute Result (SR)

The substitute function $SR(\mathbf{f}, \mathcal{E})$ takes two arguments. The first argument $\mathbf{f}$ is a set of closed-form solutions:

$$\mathbf{f} = (f_1(n) = e_1(n), \cdots, f_k(n) = e_k(n)),$$

and the second argument $\mathcal{E}$ is a set of expressions. The function returns a new set of expressions obtained by replacing the occurrence of $f_i(t)$ in the expressions in $\mathcal{E}$, for every $1 \leq i \leq k$ and every term t, by $e_i(t)$.

6.5 Simplify Condition

Recurrence Solver uses a systematic technique for simplifying condition of recurrence equation σ to the form present in (12) using Simplify Condition (SC). It use two different rules which are presented in Figure (2). The rule SC_1 produces a non-conditional recurrence equation if identifying $f_1(X(n), n) = f_2(X(n), n) = \ldots = f_{h-1}(X(n), n) = f_h(X(n), n)$ with an SMT query. The algorithm uses the rules SC_2 on input axiom where it processes the conditions by applying the proof strategy sequentially presented in [37]. Upon evaluation of the condition to true, it produces the modified axioms. These rules are applied repeatedly until a fixed point. In essence, they remove any irrelevant and redundant parts of the recurrence equation(s).

$$SC_1 \frac{\overset{\sigma}{f_1(X(n), n) = f_2(X(n), n) = \ldots = f_h(X(n), n) = f_{h+1}(X(n), n)}}{X(n+1) = f_1(X(n), n)}$$

$$SC_2 \frac{\overset{\sigma}{\theta_1}}{X(n+1) = f_1(X(n), n)}$$

Fig. 2: The set of rules for simplifying σ

These rules are applied repeatedly until fixed point. In essence, they remove irrelevant and redundant parts of the recurrence equation(s). How our recurrence solver used simplification technique is explained with an example. The C code snippet P is from HOLA benchmark used in [25] with variables $\mathbf{X} = \{x, y, i, j\}$.

```
int x = 0, y = 0, i = 0, j = 0;
while(x <= LINT  && y <= LINT && nondet_int())
{
```

```
    while(nondet_int())
    {
       if(x==y) { i++;}   else { j++; }
    }
    if(i>=j){ x++; y++;}  else { y++;}
}
assert(i>=j);
```

The translation module of VIAP implements Lin's [28] translation and generates the following set of axioms $\Pi_P^{\mathbf{X}}$ for program P, where $\mathbf{X} = \{x, y, i, j, LINT\}$.

$$
\begin{aligned}
&LINT_1 = LINT, i_1 = i_6(N_2), y_1 = y_6(N_2),\\
&j_1 = 0, x_1 = x_6(N_2),\\
&\forall n_1, n_2.i_2(n_1 + 1, n_2) = ite(x_6(n_2) = y_6(n_2), (i_2(n_1, n_2) + 1), i_2(n_1, n_2)),\\
&\forall n_2.i_2(0, n_2) = i_6(n_2),\\
&\forall n_1, n_2.\neg(nondet\_int_2(n_1, n_2) > 0),\\
&\forall n_1, n_2.n_1 < N_1(n_2) \rightarrow (nondet\_int_2(n_1, n_2) > 0),\\
&\forall n_2.i_6(n_2 + 1) = i_2(N_1(n_2), n_2),\\
&\forall n_2.y_6(n_2 + 1) = ite(i_2(N_1(n_2), n_2) \geq 0, y_6(n_2) + 1, y_6(n_2))\\
&\forall n_2.x_6(n_2 + 1) = ite(i_2(N_1(n_2), n_2) \geq 0, x_6(n_2) + 1, x_6(n_2))\\
&i_6(0) = 0, y_6(0) = 0, x_6(0) = 0\\
&\neg((x_6(N_2) <= LINT) \wedge (y_6(N_2) <= LINT) \wedge (nondet\_int_3(N_2) > 0),\\
&\forall n_2.n_2 < N_2 \rightarrow ((x_6(n_2) <= LINT) \wedge (y_6(n_2) <= LINT))\\
&\qquad\qquad \wedge (nondet\_int_3(n_2) > 0).
\end{aligned}
$$

There are three recurrence relations in the above axioms and all are non-conditional. The set of recurrence relations Π is passed to our desgin system recurrence solver module(RS). It tries to find the closed form solutions of these conditional equations. RS first tries to simplify equations by successfully proving the condition of recurrence $i_2(n_2, n_1)$ always holds using SMT solver (Z3)

$$\forall n_2.x_6(n_2) = y_6(n_2)$$

Then by simplifying the condition of recurrence $i_2(n_1 + 1, n_2) = i_2(n_1, n_2) + 1$ can be solved by RS with respect to the initial value $i_2(0, n_2) = i_6(n_2)$ and RS returns the closed-form solution $i_2(n_1, n_2) = i_6(n_2) + n_1$ which can then be used to simplify the equations for $x_6(n_1)$, $y_6(n_1)$ and $i_6(n_1)$ into

$$\forall n_2.i_6(n_2+1) = i_6(n_2) + N_1(n_2),$$
$$\forall n_2.y_6(n_2+1) = ite(i_6(n_2) + N_1(n_2) \geq 0, y_6(n_2)+1, y_6(n_2))$$
$$\forall n_2.x_6(n_2+1) = ite(i_6(n_2) + N_1(n_2) \geq 0, x_6(n_2)+1, x_6(n_2))$$
$$i_6(0) = 0, y_6(0) = 0, x_6(0) = 0.$$

In the next iteration, the condition of recurrence $x_6(n_1)$,$y_6(n_1)$ can be simplified by successfully proving following the condition $\forall n_2.i_6(n_2) + N_1(n_2) \geq 0$ always holds. Then RS solves the $x_6(n_2+1) = x_6(n_2)+1$ and $y_6(n_2+1) = y_6(n_2)+1$ respect to the initial value $y_6(0) = 0$, $x_6(0) = 0$ respectively. After substituting the closed form solution with $x_6(n_2) = n_2$, $y_6(n_2) = n_2$ the resulting set of axoims $\Pi'^{\mathbf{X}}_P$ are as follows:

$$LINT_1 = LINT, i_1 = i_6(N_2), j_1 = 0, y_1 = N_2, x_1 = N_2,$$
$$\forall n_1, n_2.\neg(nondet\_int_2(n_1, n_2) > 0),$$
$$\forall n_1, n_2.n_1 < N_1(n_2) \rightarrow (nondet\_int_2(n_1, n_2) > 0),$$
$$\forall n_2.i_6(n_2+1) = i_2(N_1(n_2), n_2), i_6(0) = 0,$$
$$\neg((N_2 <= LINT) \wedge (N_2 <= LINT) \wedge (nondet\_int_3(n_2) > 0)),$$
$$\forall n_2.n_2 < N_2 \rightarrow ((n_2 <= LINT) \wedge (n_2 <= LINT) \wedge (nondet\_int_3(n_2) > 0)).$$

For the above program the assertion to be proved is $i_6(N_2) >= 0$. Our system then attempts this by proving the more general one $\forall n_2.i_6(n_2) >= 0$ by induction in the n_2 with respect to $\Pi'^{\mathbf{X}}_P$, which was successful. Tools like ICRA[25] VeriAbs [11], UAutomizer [21], Seahorn [20], and CPAChecker [6] failed prove the assertion. Although, the system is not able to get rid of all the recurrence, but VIAP is able to prove the assertion.

7 Proof strategies

We use Z3 [15] to prove properties about a program with the axioms generated from our translator. Given that Z3 accepts first-order axioms, this seems like a straightforward task. The problem is of course that theorem proving in first-order logic is undecidable, and there are many limitations in what Z3 can prove. We describe below the strategies that we use to make Z3 work for us.

First, we use a natural number sort, but Z3 has only integer sort. So whenever we need to quantify over a natural number variable, we need to make it into a quantification on non-negative integers. For example, to encode in Z3 the following axiom that universally quantifies natural number variable n:

$$0 \leq n < N_2 \rightarrow r_7(n) \geq Y$$

we use an integer type variable n and add the condition $n \geq 0$. Another issue is that many of the benchmark programs that we tried have exponentiation. We found that Z3's builtin module NLSat (nonlinear real arithmetic) was too weak most of the time. Instead, we define our own exponentiation function $power(x, y)$ on integers and include the following axioms:

$$\forall n.n \geq 0 \rightarrow power(0, n) = 0,$$
$$\forall n, m.power(m, n) \rightarrow m = 0,$$
$$\forall n, m.n > 0 \rightarrow power(n, 0) = 1,$$
$$\forall n, m.power(n, m) = 1 \rightarrow n = 1 \vee m = 0,$$
$$\forall n, m.n > 0 \wedge m \geq 0 \rightarrow power(n, m + 1) == power(n, m) * n$$

Of course, these axioms are not complete. But so far they work better for us than Z3's build-in module.

Another experience that we had with Z3 was that it has difficulty with axioms of the form (6), which is crucial for our axiomatization of loops. Our strategy is to add the following consequence of (6) to the axiom set:

$$N = 0 \vee B(N - 1). \tag{23}$$

There are some other preprocessings that we have done while translating the axioms to Z3. Chiefly, we make use of sympy's expand and simplify functions to convert arithmetic terms to a standard form. For example, a term like $(x + 1)^2 + x$ is converted to $x^2 + 3x + 1$, and exponentiation are simplified by grouping those with the same base, for example, by converting $x^y \times x^z$ to x^{y+z}.

Finally, since Z3 does not have a build-in mechanism for induction, we implemented a simple induction scheme on top of it. In the end, we came up with two proof strategies: direct proof and proof by simple induction.

7.1 Strategy 1: direct proof

Input A - a set of first-order axioms; α - precondition (a sentence on the input values of program variables); β - postcondition (a sentence on the input and output values of program variables);

Output "proved", "failed to prove", or "refuted" (with a counter-example);

1. Declare all variables in A, α and β as integer variables;
2. Add binary integer function `power(x,y)` and its associate axioms;
3. Convert α and each axiom in A to Z3 format (after some pre-processing by sympy);
4. For each axiom in A of the form (6), add (23);
5. For each output variable `X1` in β, match it with an equation of the form `X1 = W` in A, and replace it in β by `W`. Let the resulting sentence be β'. Convert $\neg\beta'$ to Z3 format.

6. Query Z3 with the axioms thus generated and output "proved" if it returns "unsat", "refuted" (with a counter-example) if it returns the counter-example, and "failed to prove" otherwise.

7.2 Strategy 2: simple induction

The first four steps of this strategy are the same as the first one:

Input A - a set of first-order axioms; α - precondition (a sentence on the input values of program variables); β - postcondition (a sentence on the input and output values of program variables);

Output "proved", "failed to prove", or "refuted" (with a counter-example);

1. Declare all variables in A, α and β as integer variables;
2. Add binary integer function `power(x,y)` and its associate axioms;
3. Convert α and each axiom in A to Z3 format (after some pre-processing by sympy);
4. For each axiom in A of the form (6), add (23);
5. Let Q be the set of Z3 axioms generated so far.
6. For each output variable `X1` in β, match it with an equation of the form `X1 = W` in A, and replace it in β by `W`. Let the resulting sentence be β'. Find in β' a system generated natural number constant N denoting the number of iterations in a loop. The following steps try to prove $\forall k.\beta'(N/n)$ by induction on n.
7. Replace N in β' by 0. Let β'' be the resulting sentence. Convert $\neg\beta''$ to Z3 axioms and add it to Q.
8. Query Z3 with the axioms thus generated. If it returns "unsat", then continue. Otherwise, return "failed to prove".
9. Choose a new symbol k not occurring in Q or β', and declare it to be a non-negative integer.
10. Replace N in β' by k to get a new sentence β_1. Replace N in β' by $k+1$ to get another sentence β_2.
11. For each term in β_2 of the form $f(k+1)$, match it with an equation in A of the form $\forall n.f(n+1) = W(n)$, and replace $f(k+1)$ in β_2 by $W(k)$. Let the resulting sentence be β_3. Convert $\neg(\beta_1 \rightarrow \beta_3)$ to Z3 format and add it to Q.
12. Query Z3 with the axioms thus generated. If it returns "unsat", then return with "proved". Otherwise, return "failed to prove".

Here, we consider a more substantial example with nested loops to demonstrate how VIAP handle such program. Consider the following algorithm for computing the integer division by Cohen [13] (the one below is from [28] which is adapted from [32]). It has two loops, one nested inside another:

```
// X and Y are two input integers; Y > 0
Q=0; // quotient
R=X; // remainder
assume(X>=0);
assume(Y>0);
```

```
while (R >= Y) {
    A=1;
    B=Y;
    while (R >= 2*B){
        A = 2*A;
        B = 2*B;
    }
    R = R-B;
    Q = Q+A;
}
assert(X==R*Y+Q);
```

Our translator outputs the following set of axioms after solving recurrences $A_2()$ and $B_2()$:

$$
\begin{aligned}
&Y_1 = Y, X_1 = X,\\
&A_1 = A_7(N_2), q_1 = q_7(N_2), r_1 = r_7(N_2), B_1 = B_7(N_2),\\
&r_7(n) < 2 \times 2^{N_1(n)} \times Y,\\
&n_1 < N_1(n_2) \rightarrow r_7(n_2) \geq 2 \times 2^{n_1} \times Y,\\
&A_7(n+1) = 2^{N_1(n)} \times 1, B_7(n+1) = 2^{N_1(n)} \times Y,\\
&q_7(n+1) = q_7(n) + 2^{N_1(n)} \times 1, r_7(n+1) = r_7(n) - 2^{N_1(n)} \times Y,\\
&A_7(0) = A, B_7(0) = B, q_7(0) = 0, r_7(0) = X,\\
&r_7(N_2) < Y,\\
&n < N_2 \rightarrow r_7(n) \geq Y
\end{aligned}
$$

As an illustration, the translator of assertion results β as following :

$$X = q_7(N_2)Y + r_7(N_2).$$

To prove the above assertion β under the precondition α:

$$X \geq 0 \wedge Y > 0$$

It tries to prove it directly using Z3, but fails. So we try to prove it by proving the following more general one

$$\forall n. X = q_7(n)Y + r_7(n) \tag{24}$$

by induction on n, which was successful.

7.3 Array Handling Strategies

In this section, we describe some implementation details of array handling strategy. For arrays, an important module that we added is for instantiation. Our main objective is translating a program to first-order logic axioms with arithmetic. This translation provides the relationship between the input and output values of the program variables. The relationship between the input and output values of the program variables is independent of what one may want to prove about the program. SMT solver tools like Z3 is just an off shelf tool, so we never considered using the built-in array function there.

7.3.1 Instantiation

Instantiation is one of the most important phases of the pre-processing of axioms before the resulting set of formulas is passed on an SMT-solver according to some proof strategies. The objective is to help an SMT solver like Z3 to reason with quantifiers. Whenever an array element assignment occurs inside a loop, our system will generate an axiom like the following:

$$\begin{aligned} &\forall x_1, x_2 ... x_{k+1}, n. d\text{k}array_i(x_1, x_2 ... x_{k+1}, n+1) = \\ &\quad ite(x_1 = A \wedge x_2 = E_2 \wedge ... \wedge x_{k+1} = E_{h+1}, E, \\ &\qquad d\text{k}array_i(x_1, x_2 ... x_{k+1}, n)) \end{aligned} \tag{25}$$

where

- A is a k-dimensional array.
- $d\text{k}array_i$ is a temporary function introduced by translator.
- x_1 is an array name variable introduced by translator, and is universally quantified over arrays of k dimension.
- x_2,....,x_{k+1} are natural number variables representing array indices, and are universally quantified over natural numbers.
- n is the loop counter variable universally quantified over natural numbers.
- $E, E_2, ..., E_{k+1}$ are expressions.

For an axiom like (25), our system performs two types of instantiations:

- **Instantiating Arrays**: this substitutes each occurrence of variable x_1 in the axiom (25) by the array constant A, and generates the following axiom:

$$\begin{aligned} &\forall x_1, x_2 ... x_{k+1}. d\text{k}array_i(A, x_2 ... x_{k+1}, n+1) = \\ &\quad ite(x_2 = E_2 \wedge ... \wedge x_{k+1} = E_{h+1}, E, d\text{k}array_i(A, x_2 ... x_{k+1}, n)) \end{aligned} \tag{26}$$

- **Instantiating Array Indices**: This substitutes each occurrence of variable x_i, $2 \leq i \leq k$, in the axiom (26) by E_i, and generates the following axiom:

$$\forall n. d\text{k}array_i(A, E_2 ... E_{k+1}, n+1) = E \tag{27}$$

Example This example shows the effect of instantiation on a complete example. Consider the following Battery Controller program from the SV-COMP benchmark [36, 10]:

```
1.     int COUNT , MIN ,i=1 ;
2.     int volArray[COUNT];
3.     if( COUNT %4 != 0) return ;
4.     while(i <= COUNT /4) {
5.          if (5 >= MIN ){ volArray [i*4-4]=5; }
6.              else { volArray [i*4-4]=0; }
7.          if (7 >= MIN ){ volArray [i*4-3]=7; }
8.              else { volArray [i*4-3]=0; }
9.         if (3 >= MIN ){ volArray [i*4-2]=3; }
10.             else { volArray [i*4-2]=0; }
11.         if (1 >= MIN ){ volArray [i*4-1]=1; }
12.             else  { volArray [i*4-1]=0; }
13.         assert ( volArray[i]>=MIN ||volArray[i]==0);
14.         i=i+1; }
```

Our system generates the following set of axioms after the recurrences from the loop are solved by SymPy:

1. $COUNT_1 = COUNT$
2. $j_1 = j$
3. $volArray_1 = volArray$
4. $MIN_1 = MIN$
5. $i_1 = ite(((COUNT\%4) == 0), (N_1 + 1), 1)$
6. $$\forall x_1, x_2.d1array_1(x_1, x_2) = ite((COUNT\%4) == 0, d1array_{13}(x_1, x_2, N_1), d1array(x_1, x_2))$$
7.

$$\forall x_1, x_2, n_1.d1array_{13}(x_1, x_2, (n_1 + 1)) = ite(1 \geq MIN,$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 1, 1, d1array_{13}(volArray, x_2, n_1)),$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 1, 0,$$
$$ite(3 \geq MIN,$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 2, 1, d1array_{13}(volArray, x_2, n_1))$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 2, 0,$$
$$ite(7 \geq MIN,$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 3, 1, d1array_{13}(volArray, x_2, n_1)),$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 3, 0,$$
$$ite(5 \geq MIN,$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 4, 1,$$
$$d1array_{13}(volArray, x_2, n_1)),$$
$$ite(x_1 = volArray \wedge x_2 = (n_1 + 1) * 4 - 4, 0,$$
$$d1array_{13}(volArray, x_2, n_1)))))))))$$

8. $\forall x_1, x_2.d1array_{13}(x_1, x_2, 0) = d1array(x_1, x_2)$
9. $(N_1 + 1) > (COUNT/4)$
10. $\forall n_1.(n_1 < N_1) \rightarrow (n_1 + 1) \leq (COUNT/4)$

where $(COUNT\%4) == 0$ is copied directly from the conditional `COUNT%4 !=0` in the program and is converted to $(COUNT\%4) = 0$ in Z3.

The instantiation module will then generate the following new axioms from the one in 7:

1. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1 + 1) * 4 - 1, n_1 + 1) = ite(1 \geq MIN, 1, 0)$
2. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1 + 1) * 4 - 2), n_1 + 1) = ite(3 \geq MIN, 1, 0)$
3. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1 + 1) * 4 - 3, n_1 + 1) = ite(7 \geq MIN, 1, 0)$
4. $\forall n_1.0 \leq n_1 < COUNT \rightarrow$
   $d1array_{13}(volArray, (n_1 + 1) * 4 - 4, n_1 + 1) = ite(5 \geq MIN, 1, 0)$

For the the following assertion to prove:

$$d1array_{13}(volArray, n_1 + 0, N_1) \geq 2 \vee d1array_{13}(volArray, n_1 + 0, N_1) = 0$$

VIAP successfully proved the assertion irrespective of the value of `COUNT`. On the other hand, tools like CBMC [27] and SMACK+Corral [9] which prove this assertion for arrays with small values of `COUNT=100` fail when the COUNT value is non-deterministic or bigger(`COUNT=10000`) and this has been also reported by [10]. Other tools like UAutomizer [21], Seahorn [20], ESBMC [14], Ceagle [43], Booster [2], and Vaphor [31] fail to prove the assertion even for a small value of `COUNT`. To our

knowledge, Vaphor [31] and VeriAbs [11] are the only other systems that can prove this assertion regardless of the value of `COUNT`. □

8 Contributions in SV-COMP

In this section, we discuss VIAP's participation in the seventh and eight editions of the International Competition on Software Verification (SV-COMP), a community initiative to benchmark and compare verification tools.

8.1 SV-COMP 2018

8.1.1 Benchmarks and Results

For SV-COMP 2018, we had participated for the first time and submitted VIAP version 1[1]. VIAP participated only in three subcategories of Reach-Safety category. The brief description of each of the sub-categories in Reach Safety category, where VIAP participated, are as follows:

- **ReachSafety-Arrays**: a set of benchmarks tasks for which treatment of arrays is necessary in order to determine reachability.
- **ReachSafety-Loops**: a set of benchmarks tasks for which loop analysis is necessary.
- **ReachSafety-Recursive**: a set of benchmarks tasks for which recursive analysis is necessary.

VIAP got the second position in the ReachSafety-Arrays sub-category of the ReachSafety category. In *SV-COMP* 2018 competition, there are 167 tasks with a maximum score of 290 in the ReachSafety-Arrays sub-category. VIAP score 209 by successfully proving 123 tasks. Out of 123 tasks, 115 are confirmed result which means VIAP generated correct witness files and 9 are the unconfirm result which means VIAP generated bad witness files. In the ReachSafety-Arrays sub-category, VeriAbs came first with score 235 by successfully proving 142 tasks (confirm correct 126 and unconfirm correct 16). In ReachSafety-Recursive sub-category, VIAP score 91 out of 148 by successfully proving 69 tasks out of 96. CBMC (95), CPA-Seq (108), ESBMC-incr (99), ESBMC-kind (99), Map2Check (98), Symbiotic (103), UAutomizer (116), UTaipan (104) perform better than VIAP . In ReachSafety-Loops sub-category, VIAP score only 44 out of 274 by successfully proving 79 task out of 274 (confirm correct 21 and unconfirm correct 58), it procduce one incorrect result for which 32 point got deduced.

Categories in which VIAP did not participate include ReachSafety-ControlFlow, ReachSafety-BitVectors, ReachSafety-ECA, ReachSafety-Floats , ReachSafety-Heap,

[1] `https://sv-comp.sosy-lab.org/2019/results/results-verified/`

and ReachSafety-Heap, which require translation and reasoning about the pointer, dynamic linked data structures, and floating points, these forms of analysis are currently not supported by VIAP .

8.2 SV-COMP 2019

In the 2019 edition of SV-COMP [5][2], we submitted an updated version of VIAP to that used in SV-COMP 2018. VIAP came first in the ReachSafety-Arrays and ReachSafety-Recursive sub-category of the ReachSafety category. We attribute the relatively better performance of VIAP in the SV-COMP 2019 to two primary factors which are as following:

- Integration of RS solver.
- Effective array handling proof strategy.
- Effective counter example and witness generation strategy.

In *SV-COMP* 2019 competition, there are 231 tasks with a maximum score of 418 in the ReachSafety-Arrays. VIAP score 378 by successfully proving 123 tasks. Out of 206 tasks, 203 are confirmed result which means VIAP generated correct witness files and 3 are the unconfirm result which means VIAP generated bad witness files. In the ReachSafety-Arrays sub-category, VeriAbs came second with score 365 by successfully proving 202 tasks(confirm correct 196 and unconfirm correct 6). In ReachSafety-Recursive sub-category, VIAP also came first with score 124 out of 148 by successfully proving 89 tasks out of 96. UAutomizer came second with score 120. In ReachSafety-Loops sub-category, VIAP score only 178 out of 357 by successfully proving 124 tasks out of 208 (confirm correct 90 and unconfirm correct 34). VeriAbs(275), CPA-Seq (226), PeSCo(224), UKojak (224) UTaipan (222), UAutomizer (212), Skink(190), Symbiotic (180) perform better than VIAP .

9 Strength and Weaknesses

VIAP supports user assertions, including reachability of labels in the C-code. The advantage (strength) of this approach comes with a clean separation between the translation (semantics) and the use of the translation in proving the properties (computation). The translation part is stable. But as more efficient provers become available, the capabilities of the system improve. This is seen in our newer version of VIAP : by having a more powerful system for computing closed-form solutions of recurrences, the new system becomes more efficient and can prove many properties that our previous system were not able to. However, VIAP provides little or no support for translation and reasoning about dynamic linked data structures or programs

[2] `https://sv-comp.sosy-lab.org/2019/results/results-verified/`

with floating points. We are working in the direction to strengthen our front-and backhand to handle all types of the program so that we can participate in all the sub-categories of ReachSafety in the future edition of SV-COMP. The major disadvantage of the method which translates loop body to the recurrence relation is that if they failed to find closed form solution, then they unable to find suitable invariant as a result they failed to complete the proof. When VIAP fails to come up with a closed-form solution, it falls back to simple induction using Z3. There is clearly a need of better way to do induction and we are working on it. In terms of closed-form solution, in general it is undecidable whether a recurrence has a closed-form solution or not.

10 Related Work

10.1 Closed forms for Recurrence Relations

A number of computer algebra systems can evaluate indefinite sums like $sum(n) = \sum_{k=0}^{n} a_k$, where a_k is a sequence depending only on k, due to the pioneering work of [18], [23] and [29]. One area of interest is summing recurrence relations. Most of the computer algebra systems, such as Maple [38], SymPy [42] and Mathematica [44] can find a closed form solution of C-finite and some P-finite recurrences.

C-finite sequences: In this sequence, the recurrence $f(n)$ takes the form $f(n+r) = a_0(n)f(n) + a_1(n)f(n+1) + ... + a_{r1}(n)f(n+r1)$. Greene and Wilf have provided an algorithm to sum a general form of products of C-finite sequences [19]. The study of Kauers and Zimmermann determined whether there exists summation relationships exist between different C-finite sequences [24].

P-finite sequences: In this sequence, the recurrence $f(n)$ takes the form $f(n+r) = p_0(n)f(n) + p_1(n)f(n+1) + ... + p_{r1}(n)f(n+r1)$. Summing P-finite sequences have been discussed by Abramov and van Hoeij in terms of the original coefficients [1]. The work of Gosper [18] and Zeilberger [45] to sum P-finite sequences that are not hypergeometric [12] have been generalized by Chyzak. Karr's approach [23] approach has been extended to P-finite sequences [41] by Schneider.

Petkovsek *et. al* [34] presents a comprehensive account of all these algorithms to find a closed-form solution of C-finite and P-finite sequences recurrence, most of which are supported by SymPy. In this article we present an algorithm which can compute the closed form of certain types of mutual and conditional recurrences which none of the above algorithms supports.

10.2 Techniques that use Recurrences for Program Analysis

There has been much work using recurrence for program analysis. The work of [3] is a template-based approach which uses an abstract interpretation to detect the induction variables of recurrence relations of programs containing loops. The scope of

this approach is very limited as it can only handle a certain type of loops with some restricted syntax. The method of [40], [8] and [39] use recurrence analysis for computing accurate information about syntactically restricted loops. These approaches are also template based which handle a very simple class of recurrence equations. Ancourt et al. [4] is based on an affine derivative closure method where it approximates any loop by the translation. This method is specifically designed to discover linear recurrence in equations. The methods [17, 33] compute the polynomial invariants of programs using abstract interpretation. The other methods only handle assignments which can be described by C-Finite recurrences. The authors in [7] present algorithms that find out the symbolic bounds for nested loops and then compute the closed forms of certain loops to use over-approximation and under-approximation to find out suitable polynomial invariant. The ALIGATOR tool [26, 22] is another tool that infers loop invariants that are polynomial equalities by solving the recurrence equations representing the loop body in closed form. A recently published work [25] is on similar lines. However, it handles subclasses of recurrence relationships as opposed to [26] and [22]. The main advantage of [25] of over [26] and [22] is that it can handle more complex program structures such as nested loop and non-deterministic input. One important difference is the invariant generation technique is assertion guided whereas the ALIGATOR tool generates all possible invariants. The following features make our approach different from other approaches

- Existing works based on recurrences analysis either focus on computing accurate information about syntactically restricted loops or focus on over-approximate analysis of general loops. In contrast, the recurrence generated by translation algorithm presented in this work precisely summarize the semantics of general loops. In other words, we aim to analyze the accurate semantics of general loops.
- A major disadvantage of existing methods is that if they fail to find a closed-form solution, then they are unable to find suitable invariant, they fail to complete the proof. On the other hand, Even if our system is unable to solve recurrence equations, our system can still complete the proof.
- Some existing methods try to use recurrence solving for inferring all invariants they can find. However our approach is property-guided or goal-directed. We only aim to verify the condition.

11 Conclusion

In this chapter, we have described the theory and implementation of the system called VIAP that can prove the safety properties of a structured program with integer assignments with respect to specifications given as assertion(s) and assumption(s). VIAP works by first translating the given program to a set of first-order axioms and then using the off-the-shelf SMT prover Z3 to prove that the assertion(s) holds under the given assumption(s). Integrated into both the translation part and the proving part is the use of our recurrence solver(RS) to solve recurrences and use of Sympy for simplifying algebraic expressions. In this chapter, we have also described

a new approach to finding the closed-form solution of various types of recurrence equations, including mutual and conditional ones. The approach is a compositional framework of computational algebra, symbolic analysis and abstract interpretation which is built on Sympy. To the best of our knowledge, this is the first work that can find the closed-form solution of mutual and conditional recurrence equations. It is an open source tool which can be used by any application which needs recurrence analysis.

For future work, we want to extend the system VIAP with more powerful heuristics for induction, for wide classes of programs, and for proving properties regarding termination. In future, we plan to extend Recurrence Solver to support more complex types of recurrence. We have also demonstrated how the integration of Recurrence Solver increases the capability of our automatic program verification tool VIAP. In future, we will demonstrate how this approach can be extensively used in loop bound analysis, test case generation and loop summarization which includes nested loop as well.

References

1. Abramov, S.A., Bronstein, M., Petkovšek, M.: On polynomial solutions of linear operator equations. In: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation. pp. 290–296. ISSAC '95, ACM, New York, NY, USA (1995), http://doi.acm.org/10.1145/220346.220384
2. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: An acceleration-based verification framework for array programs. In: Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings. pp. 18–23 (2014), https://doi.org/10.1007/978-3-319-11936-6_2
3. Ammarguellat, Z., Harrison, III, W.L.: Automatic recognition of induction variables and recurrence relations by abstract interpretation. SIGPLAN Not. 25(6), 283–295 (Jun 1990), http://doi.acm.org/10.1145/93548.93583
4. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electron. Notes Theor. Comput. Sci. 267(1), 3–16 (Oct 2010), http://dx.doi.org/10.1016/j.entcs.2010.09.002
5. Beyer, D.: Automatic verification of c and java programs: Sv-comp 2019. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 133–155. Springer International Publishing, Cham (2019)
6. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: International Conference on Computer Aided Verification. pp. 184–190. Springer (2011)
7. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: Abc: Algebraic bound computation for loops. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 103–118. LPAR'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1939141.1939148
8. Cachera, D., Jensen, T., Jobin, A., Kirchner, F.: Inference of polynomial invariants for imperative programs: A farewell to gröbner bases. In: Proceedings of the 19th International Conference on Static Analysis. pp. 58–74. SAS'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-33125-1_7
9. Carter, M., He, S., Whitaker, J., Rakamarić, Z., Emmi, M.: Smack software verification toolchain. pp. 589–592. ICSE '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2889160.2889163

10. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: Ranzato, F. (ed.) Static Analysis. pp. 428–449. Springer International Publishing, Cham (2017)
11. Chimdyalwar, B., Darke, P., Chauhan, A., Shah, P., Kumar, S., Venkatesh, R.: Veriabs: Verification by abstraction competition contribution. In: Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206. pp. 404–408. Springer-Verlag New York, Inc., New York, NY, USA (2017), https://doi.org/10.1007/978-3-662-54580-5_32
12. Chyzak, F.: An extension of zeilberger's fast algorithm to general holonomic functions. Discrete Math. 217(1-3), 115–134 (Apr 2000), http://dx.doi.org/10.1016/S0012-365X(99)00259-9
13. Cohen, E.: Programming in the 1990s: An Introduction to the Calculation of Programs. Springer-Verlag, Berlin, Heidelberg (1990)
14. Cordeiro, L., Morse, J., Nicole, D., Fischer, B.: Context-bounded model checking with esbmc 1.17. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 534–537. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
16. Erosa, A.M., Hendren, L.J.: Taming control flow: A structured approach to eliminating goto statements. In: Bal, H.E. (ed.) Proceedings of the IEEE Computer Society ICCLs, Toulouse, France. pp. 229–240 (1994), http://dx.doi.org/10.1109/ICCL.1994.288377
17. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design. pp. 57–64. FMCAD '15, FMCAD Inc, Austin, TX (2015), http://dl.acm.org/citation.cfm?id=2893529.2893544
18. Gosper, R.W.: Decision procedure for indefinite hypergeometric summation. Proceedings of the National Academy of Sciences of the United States of America 75(1), 40–42 (1978), http://www.jstor.org/stable/67570
19. Greene, C., Wilf, H.S.: Closed form summation of c-finite sequences. Transactions of the American Mathematical Society 359(3), 1161–1189 (2007), http://www.jstor.org/stable/20161620
20. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework, pp. 343–361. Springer International Publishing, Cham (2015), https://doi.org/10.1007/978-3-319-21690-4_20
21. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: Ultimate automizer with array interpolation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 455–457. Springer (2015)
22. Humenberger, A., Jaroschek, M., Kovács, L.: Automated generation of non-linear loop invariants utilizing hypergeometric sequences. CoRR abs/1705.02863 (2017), http://arxiv.org/abs/1705.02863
23. Karr, M.: Summation in finite terms. J. ACM 28(2), 305–350 (Apr 1981), http://doi.acm.org/10.1145/322248.322255
24. Kauers, M., Zimmermann, B.: Computing the algebraic relations of c-finite sequences and multisequences. J. Symb. Comput. 43(11), 787–803 (Nov 2008), http://dx.doi.org/10.1016/j.jsc.2008.03.002
25. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-linear reasoning for invariant synthesis. Proceedings of the ACM on Programming Languages 2(POPL), 54 (2017)
26. Kovács, L.: Reasoning algebraically about p-solvable loops. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 249–264. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792758
27. Kroening, D., Tautschnig, M.: CBMC – C Bounded Model Checker, pp. 389–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014), https://doi.org/10.1007/978-3-642-54862-8_26

28. Lin, F.: A formalization of programs in first-order logic with a discrete linear order. Artificial Intelligence 235, 1 – 25 (2016), http://www.sciencedirect.com/science/article/pii/S000437021630011X
29. Man, Y.K.: On computing closed forms for indefinite summations. J. Symb. Comput. 16(4), 355–376 (Oct 1993), http://dx.doi.org/10.1006/jsco.1993.1053
30. Manna, Z.: Introduction to Mathematical Theory of Computation. McGraw-Hill, Inc., New York, NY, USA (1974)
31. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free horn clauses. In: Rival, X. (ed.) Static Analysis. pp. 361–382. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
32. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 683–693. IEEE (2012)
33. de Oliveira, S., Bensalem, S., Prevosto, V.: Polynomial invariants by linear algebra. CoRR abs/1611.07726 (2016), http://arxiv.org/abs/1611.07726
34. Petkovsek, M., Wilf, H.S., Zeilberger, D.: A = B. Wellesley, Mass. : A K Peters (1996)
35. Petkovšek, M.: Hypergeometric solutions of linear recurrences with polynomial coefficients. J. Symb. Comput. 14(2-3), 243–264 (Aug 1992), http://dx.doi.org/10.1016/0747-7171(92)90038-6
36. Program Committee / Jury: SV-COMP:Benchmark Verification Tasks (2018), https://sv-comp.sosy-lab.org/2018/
37. Rajkhowa, P., Lin, F.: VIAP - automated system for verifying integer assignment programs with loops. In: Jebelean, T., Negru, V., Petcu, D., Zaharie, D., Ida, T., Watt, S.M. (eds.) 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017. pp. 137–144. IEEE Computer Society (2017), https://doi.org/10.1109/SYNASC.2017.00032
38. Redfern, D.: The Maple Handbook (Maple V Release 3). Springer-Verlag, Berlin, Heidelberg (1994)
39. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. Journal of Symbolic Computation 42(4), 443–476 (2007)
40. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using gröbner bases. SIGPLAN Not. 39(1), 318–329 (Jan 2004), http://doi.acm.org/10.1145/982962.964028
41. Schneider, C.: A new sigma approach to multi-summation. Advances in Applied Mathematics 34, 740–767 (05 2005)
42. SymPy Development Team: SymPy: Python library for symbolic mathematics (2016), http://www.sympy.org
43. Wang, D., Zhang, C., Chen, G., Gu, M., Sun, J.: C code verification based on the extended labeled transition system model. In: Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016. pp. 48–55 (2016), http://ceur-ws.org/Vol-1725/demo7.pdf
44. Wolfram Research Inc.: Mathematica 8.0 (2010), http://www.wolfram.com
45. Zeilberger, D.: A holonomic systems approach to special functions identities. Journal of computational and applied mathematics 32(3), 321–368 (1990)

12 Appendix

12.1 Proof of Correctness

Proposition 1 *Equation (11) represents closed-form solution of the set of mutually recurrence equations (9).*

Proof We prove that equation (11) represents the closed-form solution of the set of mutual recurrence equations (9) using induction.

- **Base Case:** The base case when $n = 0$ and $n = 1$ can be proved easily.
- **Induction Hypothesis:** Assume that the following equations holds when $n = k$ such that $k > 1$ for all $i \in [1, h]$

  $X_i(k) = (A^k \times h^k \times (E1 + ... + E_h))$

$$+((C_1 + .. + C_h) \times \frac{A(1 - A^k \times h^k)}{1 - A \times h}) + C_i \quad (28)$$

- **Inductive Step:** We prove that the equations also holds when $n = k + 1$ use the Induction Hypothesis above. From the equations (9) we have the following:

$$X_i(k + 1) = A \times (X_1(k) + ... + X_h(k)) + C_i$$

After substituting corresponding equations from (28) Induction Hypothesis.

$$= A \times (((A^k \times h^k \times (E1 + ... + E_h))$$

$$+((C_1 + .. + C_h) \times \frac{A(1 - A^k \times h^k)}{1 - A \times h}) + C_1) + ...$$

$$+((A^k \times h^k \times (E1 + ... + E_h))$$

$$+((C_1 + .. + C_h) \times \frac{A(1 - A^k \times h^k)}{1 - A \times h}) + C_h)) + C_i$$

$$= A \times ((A^k \times h^{k+1} \times (E1 + ... + E_h))$$

$$+((C_1 + .. + C_h) \times \frac{A \times h \times (1 - A^k \times h^k)}{1 - A \times h}) + (C_1 + .. + C_h) + C_i)$$

$$= A \times ((A^k \times h^{k+1} \times (E1 + ... + E_h))$$

$$+((C_1 + .. + C_h) \times (\frac{A \times h \times (1 - A^k \times h^k)}{1 - A \times h} + 1))) + C_i$$

$$= A \times ((A^k \times h^{k+1} \times (E1 + ... + E_h))$$

$$+((C_1 + .. + C_h) \times (\frac{(1 - A^{k+1} \times h^{k+1})}{1 - A \times h}))) + C_i$$

$$= (A^{k+1} \times h^{k+1} \times (E1 + ... + E_h))$$

$$+((C_1 + .. + C_h) \times (\frac{A \times (1 - A^{k+1} \times h^{k+1})}{1 - A \times h})) + C_i$$

Hence, for all value of $n \geq 0$ the closed-form solutions of the equations 9 holds. □

Proposition 2 *Equation (13) represents closed-form solution of the conditional recurrence (12) when its conditional recurrence is type 1.*

Proof Base Case: when $n = 0$ is very trivial. For the induction step you want to assume that $k > 0$,

$$X(k) = ite(\theta_1, \mathbf{E}_1(k), ite(\theta_2, \mathbf{E}_2(k), ...ite(\theta_h, \mathbf{E}_h(k), \mathbf{E}_{h+1}(k))..)$$

Now, we tried to show the following:

$$X(k+1) = ite(k = 0, E, ite(\theta_1, \mathbf{E}_1(k+1), ite(\theta_2, \mathbf{E}_2(k+1), ... \\ ite(\theta_h, \mathbf{E}_h(k+1), \mathbf{E}_{h+1}(k+1))..)$$

using conditional recurrence (12).

- Case1: When θ_1 is true, we have

$$X(k+1) = f_1(X(k)) = f_1(X(k)/\mathbf{E}_1(k)) = \mathbf{E}_1(k+1)$$

  As $\mathbf{E}_1$ is the closed form solution of $X(n+1) = f_1(X(n))$
- Similarly other cases can be derived.

So the assumption is true. □

Proposition 3 *Equation (15) represents closed-form solution of the conditional recurrence (12) when its conditional recurrence is type 3.*

Proof Base case: The base case for conditional equation (12) is $X(0) = E$. We can also have $X(0) = E_1(0)$ from equation(15) when $n = 0$ as then condition $C_0 \leq n < C_1$ is satisfied. As we know that $E_1(n)(= g_i(n))$ is the closed form solution of $g_i(n+1) = f_1(g_i(n), n)$ with respect to the initial value $X(0) = E$. Hence $E_1(0) = E$ and we can conclude that the base case holds.

$$\begin{aligned} X(k) = ite(&C_0 \leq k < C_1, E_1(k), \\ &ite(k < C_2, E_2(k), ..., \\ &\quad ite(k < C_h, E_h(k), E_{h+1}(k))..) \end{aligned} \tag{29}$$

We assume equation (29) is correct where $k \geq 0$. Now using recurrences equation (12) with the initial value $X(0) = E$, we tried to find whether equation (30) also holds using case analysis

$$\begin{aligned} X(k+1) = ite(&C_0 \leq (k+1) < C_1, E_1(k+1),\\ &ite((k+1) < C_2, E_2(k+1), ...,\\ &ite((k+1) < C_h, E_h(k+1), E_{h+1}(k+1))..) \end{aligned} \tag{30}$$

- **Case 1:** When $C_0 \leq (k+1) < C_1$ is true. We can derive $C_0 \leq k < C_1$ is also true from the assumption (29). Then we have $X(k) = E_1(k)$ from assumption (29). Now from the recurrences equation (12), we can also derive condition $\theta_{\pi^{-1}(1)}$ is true using $C_0 \leq (k+1) < C_2$ as it satisfies the following equations.

$$\begin{aligned} &\forall n.C_0 \leq n < C_1 \rightarrow \phi_{\pi^{-1}(1)}(n),\\ &\forall n.n \geq C_1 \rightarrow \neg\phi_{\pi^{-1}(1)}(n). \end{aligned}$$

  As we know that $E_1(n)(= g_{\pi^{-1}(1)}(n))$ is the closed form solution of $g_{\pi^{-1}(1)}(n+1) = f_1(g_{\pi^{-1}(1)}(n), n)$ with respect to initial value $g_{\pi^{-1}(1)}(0) = E$. Hence we get $g_{\pi^{-1}(1)}(n+1) = E_1(n+1)$. Now we have the following from recurrence equation (12)

$$X(k+1) = f_{\pi^{-1}(1)}(X(k), k) = f_{\pi^{-1}(1)}(E_1(k), k) = E_1(k+1) \tag{31}$$

- **Case 2:** When $(k+1) < C_2$ is true. We can derive $k < C_2$ is also true from the assumption (29). Then we have $X(k) = E_1(k)$ from assumption (29). Now from recurrences equation (12), we can also derive condition θ_i is true using $(k+1) < C_2$ as it satisfies the following equations.

$$\begin{aligned} &\forall n.C_1 \leq n < C_2 \rightarrow \phi_{\pi^{-1}(2)}(n),\\ &\forall n.n \geq C_1 \rightarrow \neg\phi_{\pi^{-1}(2)}(n). \end{aligned}$$

  As we know that $E_1(n)(= g_{\pi^{-1}(2)}(n))$ is the closed form solution of $g_{\pi^{-1}(2)}(n+1) = f_1(g_{\pi^{-1}(2)}(n), n)$ with respect to the initial value $g_{\pi^{-1}(2)}(0) = E_1(n/C_1)$. Hence we can $g_{\pi^{-1}(2)}(n+1) = E_1(n+1)$. Now we have from recurrence equation (12)

$$X(k+1) = f_{\pi^{-1}(2)}(X(k), k) = f_{\pi^{-1}(2)}(E_2(k), k) = E_2(k+1) \tag{32}$$

□

  Similarly we can show the other cases $h-2$ to show that equation (30) is correct.

Proposition 4 *Equation (18) the represents the closed-form solution of the conditional recurrence (16) when the conditional recurrence is type 4.*

Proof Base case: The base case for conditional equation (12) is $X(0) = E$. We can also have $X(0) = E \pm 0 \times B$ from equation(16) when $n = 0$ as then condition $0 \leq n \leq C$ is satisfied. Hence we can conclude that base case holds.

$$X(k) = ite(0 \leq n \leq C, E + n \times B, ite(n = C + 1, \\ E + A + C \times B, E + A + C \times B + (\lfloor (k - C - 1)/c \rfloor) \times (\pm A) \\ +(k - C - 1 - \lceil k - C - 1/c \rceil) \times (\pm B))) \tag{33}$$

We assume equation (47) is correct where $k \geq 0$. Now using recurrences equation (20) with the initial value $X(0) = E$, we tried to find whether equation (48) also holds using case analysis

$$X(k + 1) = ite(0 \leq k + 1 \leq C, E + n \times B, ite(k + 1 = C + 1, \\ E + A + C \times B, E + A + C \times B + (\lfloor (k + 1 - C - 1)/c \rfloor) \times (\pm A) \\ +(k + 1 - C - 1 - \lceil k + 1 - C - 1/c \rceil) \times (\pm B))) \tag{34}$$

- When $0 \leq k + 1 \leq C$ is true, we can derive $0 \leq k \leq C$ is also true. So we have $X(k) = E + k \times B$ from inductive assumption equation (33). Let us consider $f(0)\%c = x$ such that $x \neq d$. Similarly $f(1)\%c = x + 1, ...f(y)\%c = x + y = d$. Hence we can have $y = d - x = C$. Now we have the following from recurrence equation (16) as $\neg(f(k)\%c = d)$ holds when $0 \leq k \leq C$.

$$X(k + 1) = X(k) \pm B = E \pm k \times B \pm B = E \pm (k + 1) \times B \tag{35}$$

- When $k + 1 = C + 1$ is true, then we can derived that $k = C$. So we have $X(k) = E + k \times B$ from inductive assumption equation (33). Now we can have the following from recurrence equation (16) as $(f(C)\%c = d)$ holds when $0 \leq k \leq C$.

$$X(k + 1) = X(k) \pm A = E \pm k \times B \pm A = E \pm C \times B \pm A \tag{36}$$

- When $k + 1 > C + 1$ is true. We can also derive the following possible scenario from assumption equation (33).
  - When $k = C + 1$ is true. So we have $X(k) = E \pm C \times B \pm A$ from inductive assumption equation (33). Now we can have the following from recurrence equation (16) as $(f(k)\%c = d)$ holds when $k = C + 1$.

$$\begin{aligned} X(k + 1) &= X(k) \pm B \\ &= E \pm C \times B \pm A \pm B \\ &= E \pm C \times B \pm A \pm 0 \times A \pm 1 \times B \\ &= E \pm C \times B \pm A \pm \lfloor (1/c \rfloor \times A \pm (1 - \lfloor (1/c \rfloor) \times B \\ &= E \pm C \times B \pm A \pm \lfloor (k + 1 - C - 1)/c \rfloor \times A \pm \\ &\qquad ((k + 1 - C - 1) - \lfloor ((k + 1 - C - 1)/c \rfloor) \times B \end{aligned} \tag{37}$$

  - When $k > C + 1$ is true. So we have $X(k) = E + A + C \times B + (\lfloor (k - C - 1)/c \rfloor) \times (\pm A) + (k - \lceil k - C - 1/c \rceil) \times (\pm B)$ from the inductive assumption equation (33). When we expand the recurrence equation (16), we can have the following:

$$\ldots + \underbrace{B + \ldots + B}_{\text{c times}} + A + \underbrace{B \ldots + B}_{\text{c times}} + A + \ldots$$

From that we can conclude that when $f(k)\%c = d$ is true, then $(k-C-1)\%c = c-1$ is true. Similarly, when $f(k)\%c = d$ is true, $(k-C-1)\%c \neq c-1$ is true.

· When $f(k)\%c = d$, then we have

$$\begin{aligned} X(k+1) &= X(k) \pm A \\ &= E + A + C \times B + (\lfloor (k-C-1)/c \rfloor) \times (\pm A) \\ &\quad + ((k-C-1) - \lceil k-C-1/c \rceil) \times (\pm B) \pm A \\ &= E + A + C \times B + (\lfloor (k-C-1)/c \rfloor + 1) \times (\pm A) \\ &\quad + ((k-C-1) - \lceil k-C-1/c \rceil) \times (\pm B) \\ &= E + A + C \times B + (\lfloor (k+1-C-1)/c \rfloor) \times (\pm A) \\ &\quad + ((k+1-C-1) - \lceil k+1-C-1/c \rceil) \times (\pm B) \end{aligned} \quad (38)$$

Hence $\lfloor (k+1-C-1)/c \rfloor = \lfloor (k-C-1)/c \rfloor + 1$ as we have $(k-C-1)\%c = c-1$.

· When $f(k)\%c \neq d$, then we have

$$\begin{aligned} X(k+1) &= X(k) \pm B \\ &= E + A + C \times B + (\lfloor (k-C-1)/c \rfloor) \times (\pm A) \\ &\quad + ((k-C-1) - \lceil k-C-1/c \rceil) \times (\pm B) \pm B \\ &= E + A + C \times B + (\lfloor (k-C-1)/c \rfloor) \times (\pm A) \\ &\quad + ((k+1-C-1) - \lceil k-C-1/c \rceil) \times (\pm B) \\ &= E + A + C \times B + (\lfloor (k+1-C-1)/c \rfloor) \times (\pm A) \\ &\quad + ((k+1-C-1) - \lceil k+1-C-1/c \rceil) \times (\pm B) \end{aligned} \quad (39)$$

□

Hence $\lfloor (k+1-C-1)/c \rfloor = \lfloor (k-C-1)/c \rfloor$ as we have $(k-C-1)\%c \neq c-1$.

From the result of (35), (36), (37), (38) and (39), we concluded that equation (34) is correct.

Proposition 5 *Equation (15) represents closed-form solution of the conditional recurrence (12) when its conditional recurrence is type 3.*

Proof Similarly this also can be proved. □

Proposition 6 *Equation (21) represents closed-form solution of the conditional recurrence (20) when its conditional recurrence is type 5 and $A = B \times h$ for some $h > 0$.*

Proof Base case: The base case for conditional equation(12) is $X(0) = E$. We can also have $X(0) = E \pm 0 \times A$ from equation (22) when $n = 0$ as then condition $0 \leq n < C$ is satisfied. Hence we can conclude that the base case holds.

$$X(k) = ite(0 \leq k < C, E \pm k \times A, ite((k - C)\%(h + 1) = 0, \\ E \pm C \times A, E \pm C \times A \mp (((k - C)\%(h + 1))) \times B)) \tag{40}$$

We assume equation (47) is correct where $k \geq 0$. Now using recurrences equation (20) with an initial value $X(0) = E$, we tried to find equation (48) also holds using case analysis

$$X(k + 1) = ite(0 \leq (k + 1) < C, E \pm (k + 1) * A, ite((k + 1 - C)\%(h + 1) = 0, \\ E \pm C \times A, E \pm C \times A \mp (((k + 1 - C)\%(h + 1))) \times B)) \tag{41}$$

- Case 1: When $0 \leq (k + 1) < C$ is true. We can also derived $0 \leq k < C$ is true from equation (47). Now from recurrences equation (22), we can also derive that condition θ is true using $0 \leq (k + 1) < C$ as it satisfies the following equations.

$$\forall n.0 \leq n < C \rightarrow \theta(X(n)/(E \pm n \times A)) \wedge \neg\phi(X(n)/(E \pm C \times A))$$

$$X(k + 1) = X(k) \pm A = E \pm k \times A \pm A = E \pm (k + 1) \times A \tag{42}$$

- Case 2: When $(k + 1 - C)\%(h + 1) = 0$ is true. We can also derive the following possible scenario from assumption equation (47).
  - When $k < C$ and $(k - C)\%(h + 1) \neq 0$, we can have $(k - C)\%(h + 1) = 1$ and $k = C - 1$. From inductive assumption, we have $X(k) = E \pm k \times A = E \pm (C - 1) \times A$. Hence $\theta(X(k)/E \pm (C - 1) \times A)$ holds, we can have the following from recurrence equation (20).

$$X(n + 1) = X(n) \pm A \\ = E \pm (C - 1) \times A \pm A = E \pm C \times A \tag{43}$$

  - When $k \geq C$ and $(k - C)\%(h + 1) \neq 0$. we can have $(k - C)\%(h + 1) = h$. From the inductive assumption, we have $X(k) = E \pm C \times A \mp (((k - C)\%(h + 1))) \times B))$. We can derive that $(k - C)\%(h + 1) = h$.

$$E \pm C \times A \mp (((k - C)\%(h + 1))) \times B)) \\ = E \pm C \times A \mp h \times B$$

    Hence $\theta(X(k)/(E \pm C \times A \mp h \times B))$ holds, we can have the following from recurrence equation (20).

$$\begin{aligned} X(n+1) &= X(n) \pm A \\ &= E \pm C \times A \mp h \times B \pm A \\ &= E \pm C \times A \end{aligned} \tag{44}$$

- When $k < C$ and $(k - C)\%(h + 1) = 0$. This scenario is not possible because it contradicts the assumption $(k + 1 - C)\%(h + 1) = 0$
- When $k \geq C$ and $(k - C)\%(h + 1) = 0$. This scenario is not possible because it contradicts the assumption $(k + 1 - C)\%(h + 1) = 0$

• Case 3: When $(k + 1) \geq C \wedge (k + 1 - C)\%(h + 1) \neq 0$ is true. We can also derived the following possible scenario from assumption equation (47).

- When $k < C$ and $(k - C)\%(h + 1) \neq 0$, we can have $(k - C)\%(h + 1) = 1$,$k = C - 1$ and $(k + 1 - C)\%(h + 1) = 0$. This scenario is not possible, as it contradicts the assumption.
- When $k < C$ and $(k - C)\%(h + 1) = 0$. This scenario is not possible, as it is contradicts because $(k - C)\%(h + 1) = 0$ holds when $k = -C$.
- When $k > C$ and $(k - C)\%(h + 1) \neq 0$, we can have $X(k) = E \pm C \times A \mp (((k - C)\%(h + 1))) \times B$. Hence $\neg\theta(X(k)/(E \pm C \times A \mp (((k - C)\%(h + 1))) \times B))$ holds, we can have the following from recurrence equation (20).

$$\begin{aligned} X(n+1) &= X(n) \pm B \\ &= E \pm C \times A \mp (((k - C)\%(h + 1))) \times B \mp B \\ &= E \pm C \times A \mp (((k - C)\%(h + 1)) + 1) \times B \\ &= E \pm C \times A \mp (((k + 1 - C)\%(h + 1))) \times B \end{aligned} \tag{45}$$

- When $k > C$ and $(k - C)\%(h + 1) = 0$, we can have $X(k) = E \pm C \times A$. Hence $\neg\theta(X(k)/(E \pm (C - 1) \times A))$ holds, we can have the following from recurrence equation (20).

$$\begin{aligned} X(n+1) &= X(n) \pm B \\ &= E \pm (C - 1) \times A \mp B \\ &= E \pm C \times A \mp (((k + 1 - C)\%(h + 1))) \times B \end{aligned} \tag{46}$$

From $(k - C)\%(h + 1) = 0$ and $(k + 1 - C)\%(h + 1) \neq 0$, we can conclude that $((k + 1 - C)\%(h + 1)) = 1$ □

From the result of (42), (43), (44), (45) and (46), we concluded that equation (41) is correct.

Proposition 7 *Equation (22) represents the closed-form solution of the conditional recurrence (20) when its conditional recurrence is type 5 and $B = A \times h$ for some $h > 0$.*

Proof Base case: The base case for conditional equation (12) is $X(0) = E$. We can also have $X(0) = E \pm 0 \times A$ from equation (22) when $n = 0$ as then condition $0 \leq n < C$ is satisfied. Hence we can conclude that the base case holds.

$$\begin{aligned} X(k) = ite(&0 \leq k < C, E \pm k \times A, \\ &ite((k - C)\%(h + 1) = 0, E \pm C \times A, \\ &ite((k - C)\%(h + 1) = 1, E \pm C \times A \mp B, \\ &, E \pm C \times A \mp B \pm (((k - C)\%(h + 1)) - 1) \times A))) \end{aligned} \tag{47}$$

We assume equation (47) is correct where $k \geq 0$. Now using recurrences equation (20) with an initial value $X(0) = E$, we tried to find whether equation (48) also holds using case analysis

$$\begin{aligned} X(k + 1) = ite(&0 \leq (k + 1) < C, E \pm (k + 1) \times A, \\ &ite(((k + 1) - C)\%(h + 1) = 0, E \pm C \times A, \\ &ite(((k + 1) - C)\%(h + 1) = 1, E \pm C \times A \mp B, \\ &, E \pm C \times A \mp B \pm ((((k + 1) - C)\%(h + 1)) - 1) \times A))) \end{aligned} \tag{48}$$

- Case 1: When $0 \leq (k+1) < C$ is true. We can also derived $0 \leq k < C$ is true from equation (47). Now from recurrences equation (22), we can also derive condition θ is true using $0 \leq (k + 1) < C$ as it satisfies the following equations.

$$\forall n.0 \leq n < C \rightarrow \theta(X(n)/(E + n \times A)) \wedge \neg\theta(X(n)/(E + C \times A))$$

$$X(k + 1) = X(k) \pm A = E \pm k \times A \pm A = E \pm (k + 1) \times A \tag{49}$$

- Case 2: When $(k + 1 - C)\%(h + 1) = 0$ is true. We can also derived following possible scenario from assumption equation (47).
  - When $k < C$ and $(k - C)\%(h + 1) \neq 0$, we can have $(k - C)\%(h + 1) = 1$ and $k = C - 1$. From the inductive assumption, we have $X(k) = E \pm k \times A = E \pm (C-1) \times A$. Hence $\theta(X(k)/E \pm (C-1) \times A)$ holds, we can have the following from recurrence equation (20).

$$\begin{aligned} &X(n + 1) = X(n) \pm A \\ &= E \pm (C - 1) \times A \pm A = E \pm C \times A \end{aligned} \tag{50}$$

  - When $k \geq C$ and $(k-C)\%(h+1) \neq 0$. we can have $(k-C)\%(h+1) = h$. From the inductive assumption, we have $X(k) = E \pm C \times A \mp B \pm (((n-C)\%(h+1))-1) \times A = E \pm C \times A \mp B \pm (h - 1) \times A$. Hence $\theta(X(k)/(E \pm C \times A \mp B \pm (h - 1) \times A)$ holds, we can have the following from recurrence equation (20).

$$\begin{aligned} X(n+1) &= X(n) \pm A \\ &= E \pm C \times A \mp B \pm (h-1) \times A \pm A \\ &= E \pm C \times A \mp B \pm h \times A \\ &= E \pm C \times A \end{aligned} \tag{51}$$

Hence we know that $B = h \times A$.

- When $k < C$ and $(k - C)\%(h + 1) = 0$. This scenario is not possible because its contradict the assumption $(k + 1 - C)\%(h + 1) = 0$
- When $k \geq C$ and $(k - C)\%(h + 1) = 0$. This scenario is not possible because it contradict the assumption $(k + 1 - C)\%(h + 1) = 0$

- Case 3: When $((k+1)-C)\%(h+1) = 1$ is true, we can derive that $(k-C)\%(h+1) = 0$. We can $X(k) = E \pm C \times A$. Hence $\neg\theta(X(k)/E \pm C \times A)$ holds, we can have the following from recurrence equation (20).

$$X(n+1) = X(n) \mp B = E \pm C \times A \mp B = E \pm C \times A \mp B \tag{52}$$

- Case 4: When $(k + 1) \geq C \wedge (k + 1 - C)\%(h + 1) \neq 0 \wedge (k + 1 - C)\%(h + 1) \neq 1$ is true. We can also derived following possible scenario from assumption equation (47).
  - When $k < C$ and $(k - C)\%(h + 1) = 0$. This scenario is not possible, as it contradicts because $(k - C)\%(h + 1) = 0$ holds when $k = -C$.
  - When $k < C$ and , we can have $(k - C)\%(h + 1) = 1$,$k = C - 1$ and $(k + 1 - C)\%(h + 1) = 0$. This scenario is not possible, as its contradict the assumption because we cannot have $(k + 1) \geq C$.
  - When $k < C$, $(k - C)\%(h + 1) \neq 1$ and $(k - C)\%(h + 1) \neq 0$. This scenario is not possible, as it contradict the assumption.
  - When $k \geq C$ and $(k-C)\%(h+1) = 0$, we can conclude that $(k+1-C)\%(h+1) = 1$ which contradict the assumption. This scenario is not possible.
  - When $k \geq C$ and $(k-C)\%(h+1) = 1$, we can have $X(k) = E \pm C \times A \mp B$. Hence $\theta(X(k)/(E \pm C \times A \mp B)$ holds, we can have the following from recurrence equation (20).

$$\begin{aligned} X(n+1) &= X(n) \pm A \\ &= E \pm C \times A \mp B \pm A \\ &= E \pm C \times A \mp B \pm 1 \times A \\ &= E \pm C \times A \mp B \pm (2-1) \times A \\ &= E \pm C \times A \mp B \pm (((n+1-C)\%(h+1)) - 1) \times A \end{aligned} \tag{53}$$

From $(k - C)\%(h + 1) = 1$, we can drive $(k + 1 - C)\%(h + 1) = 2$.

- When $k \geq C$, $(k - C)\%(h + 1) \neq 1$ and $(k - C)\%(h + 1) \neq 0$, we can have $X(k) = E \pm C \times A \mp B \pm (((k - C)\%(h + 1)) - 1) \times A$. Hence $\theta(X(k)/(E \pm C \times$

$A \mp B \pm (((k - C)\%(h + 1)) - 1) \times A)$ holds, we can have the following from recurrence equation (20).

$$\begin{aligned} & X(n + 1) = X(n) \pm A \\ & = E \pm C \times A \mp B \pm (((k - C)\%(h + 1)) - 1) \times A \pm A \\ & = E \pm C \times A \mp B \pm (((k + 1 - C)\%(h + 1)) - 1) \times A \end{aligned} \tag{54}$$

From the results of (49), (50), (51), (52), (53) and (54), we concluded that equation (48) is correct. □

12.2 Termination Proof

Proposition 8 *The algorithm 2 terminates and the total number of iterations in the worst case is* $m(m + 1)/2$ *where m is the input size of recurrence relations.*

Proof **Assumption**: Each iteration of the loop involves a Sympy's *rsolve* call which always terminates.

The termination condition of the algorithm 2 - when Π is empty. We can say that the algorithm always terminates since the sizes of Π and Δ move in a monotone way up and down and one always moves. Thus, eventually Π becomes empty and the algorithm terminates. Now we show in the worst case, the algorithm 2 terminates after $m(m + 1)/2$ iterations.

Let's consider the input set recurrences $\Pi = \{\sigma^0, \sigma_0^0, \sigma^1, \sigma_0^1, ..., \sigma^m, \sigma_0^m\}$ and before start of the while loop $selectNC(\Pi)$ selects a tuple $\langle \sigma^0, \sigma_0^0, n\rangle$ where n is recurrence variable. In the first step of the while loop, it tries to find the closed form solution of σ^0 with respect to initial value σ_0^0 by calling $rsolve(\sigma^0, \sigma_0^0, n)$. This call fails to return closed form solution, then adds σ^0, σ_0^0 in the set Δ. Similarly $selectNC(\Pi)$ selected the corresponding recurrence relations $\sigma^1, \sigma^2, ..., \sigma^{n-1}$. Then corresponding $rsolve$ call failed to find closed form solution, then add $\sigma^1, \sigma_0^1, \sigma^2, \sigma_0^2$ ...,$\sigma^{m-1}, \sigma_0^{m-1}$ in the set Δ. On m^{th} iteration, $selectNC(\Pi)$ selects tuple $\langle \sigma^m, \sigma_0^m, n\rangle$ where n is recurrence variable and finds the closed form solution of σ^0 with respect to the initial value σ_0^m by calling $rsolve(\sigma^m, \sigma_0^m, n)$ which returns the closed form solution f_{σ^m}. Then adds f_{σ^m} in C and get rid of σ^m, σ_0^m from Π. Then it call substitutes result (SR) function which substitute f_{σ^m} in Δ as Π is empty. As we are considering the worst case, we consider that all the equations in Δ after substitution of f_{σ^j} are added in Π. Now Π contains $m - 1$ pairs of recurrence relations and its corresponding inital value equations. In the same it is possible to derive the iterations of finding closed form solution of $\sigma^{m-2}, \sigma^{m-3}, ..., \sigma^0$. This algorthm terminates when Π is empty. The total numbers of iterations on termination can be represented by the following summation

$$m + (m - 1) + (m - 2) + ... + 1 = m(m + 1)/2$$

Proposition 9 *The algorithm 3 terminates and the total number of iterations in the worst case is* $m(m+1)/2$ *where m is the number of set of mutual recurrence relations present in the input set of recurrence relations.*

Proof The termination condition of the algorithm 3 - when Π is empty. If selected equations $\boldsymbol{\alpha}$ satisfies the condition explained in the sub-section 6.2, then it returns the proposed closed form solutions 11 or otherwise returns empty set. The coefficient checking is bounded by the number of coefficients involved in the input equations. Hence $\mathcal{M}$ terminates. Similarly, we can demonstrate that in the worst case, the algorithm 3 terminates after $m(m+1)/2$ iterations where m is the number of the set of mutual recurrence relations present in the input set of recurrence relations. □

Proposition 10 *The algorithm 4 terminates and the total number of iterations in the worst case is* $m(m+1)/2$ *where m is the input size of recurrence relations.*

Proof **Assumption**: Each iteration of the loop in the algorthm 4 involves a call function C which involves determining the type of input equation by checking conditions followed by $rsolve$ call which always terminates.

The termination condition of the algorithm 4 - when Π is empty. If selected equations σ satisfies the condition explained in the sub-section 6.3, then it returns the proposed corresponding closed form solutions or otherwise returns an empty set. The condition checking is bounded by the number of conditions involved in the input equation. Hence C terminates.

Similarly, we can demonstrate that in the worst case, the algorithm 4 terminates after $m(m+1)/2$ iterations where m is the input size of recurrence relations. □

Proposition 11 *The algorithm 1 terminates.*

Proof The termination condition of the algorithm 1 - when C' and Δ are empty. Each iteration of the loop in the algorthm 1 involves function calls $NCRS(2)$, $NCRS(3)$ and $CRS(4)$. The iteration after all the algorithms terminate, results C' and Δ empty because in that call $NCRS(2)$, $MRS(3)$ and $CRS(4)$ return C_1,Δ_1, C_2,Δ_3 and C_3,Δ_3 which are empty as input to all the algorithms is empty Π.