Maximal and Compositional Pattern-Based Loop Invariants

Virginia Aponte[1], Pierre Courtieu[1], Yannick Moy[2], and Marc Sango[2]

[1] CNAM, 292 rue Saint-Martin F-75141 Paris Cedex 03 - FRANCE
{maria-virginia.aponte_garcia,pierre.courtieu}@cnam.fr
[2] AdaCore, 46 rue d'Amsterdam, F-75009 Paris (France)
{moy,sango}@adacore.com

Abstract. We present a novel approach for the automatic generation of inductive loop invariants over non nested loops manipulating arrays. Unlike most existing approaches, it generates invariants containing disjunctions and quantifiers, which are rich enough for proving functional properties over programs which manipulate arrays. Our approach does not require the user to provide initial assertions or postconditions. It proceeds first, by translating body loops into an intermediate representation of parallel assignments, and second, by recognizing through static analysis code patterns that respect stability properties on accessed locations. We associate with each pattern a formula that we prove to be a so-called local invariant, and we give conditions for local invariants to compose an inductive invariant of the complete loop. We also give conditions over invariants to be locally maximal, and we show that some of our pattern invariants are indeed maximal.

Keywords: Loop invariants, compositional reasoning, automatic invariant generation.

1 Introduction

Thanks to the increased capabilities of automatic provers, deductive program verification emerges as a realistic verification technique in industry, with commercially supported toolsets [11, 30], and new certification standards recognizing its use [27]. In deductive program verification, users first annotate their programs with logical specifications; then a tool generates Verification Conditions (VCs), *i.e.* formulas encoding that the program respects its specifications; finally a tool is called to automatically prove those VCs. The problem is that, in many cases, in particular during development, not all VCs are proved automatically. Dealing with those VCs is a non-trivial task. Three cases are possible: (1) the program does not implement the specification; (2) the specification is not provable inductively; (3) the automatic prover does not find the proof. The solution to (1) is to correct the program or the specification. The solution to (3) is to use a better automatic prover. The solution to (2) is certainly the most challenging for the user. The problem occurs when, for a given loop, the user should supply an inductive loop invariant: this invariant should hold when entering the loop; it should be provable for the n+1th iteration by assuming only that it holds at the nth iteration; it should be sufficient to prove subsequent properties of interest after the loop. In practice, the user has to strengthen the loop invariant with additional properties until it can be proved inductively. In general, this requires understanding the details of the generation of VCs and the underlying mathematical theory, which is not typical engineering knowledge.

Generation of loop invariants is a well researched area, for which there exists a rich set of techniques and tools. Most of these techniques focus on the discovery of predicates that express rich arithmetic properties with a simple Boolean structure (typically, linear or non-linear constraints over program variables). In our experience with supporting industrial users of the SPARK [2] technology, these are seldom the problematic loop invariants. Indeed, users are well aware of the arithmetic properties that should be maintained through loops, and thus have no difficulty manually annotating loops with the desired arithmetic invariants. Instead, users very often have difficulties annotating loops with invariants stating *additional* properties, that that they do not recognize as required for inductive reasoning. These properties typically have a complex Boolean structure, with disjunctions and quantifiers, for expressing both the effects of past iterations and the locations not being modified by past iterations. In this paper, we focus on the automatic generation of these richer loop invariants.[3]

We present a novel technique for generating rich inductive loop invariants, possibly containing disjunctions and quantifiers (universal and existential) over loops manipulating scalar and array variables. Our method is compositional, which differentiates it from previous approaches working on entire loops: we consider a loop as a composition of smaller pieces (called reduced loops), on which we can reason separately to generate local invariants, which are then aggregated to generate an invariant of the complete loop. The same technique can be applied both to unannotated loops and to loops already annotated, in which case it uses the existing loop invariant.

Local invariants are generated based on an extensible collection of patterns, corresponding to simple but frequently used loops over scalar and array variables. As our technique relies on pattern matching to infer invariants, the choice and the variety of patterns is crucial. We have identified five categories of patterns, for search, scalar update, scalar integration, array mapping and array exchange, comprising a total of 16 patterns. For each pattern we define, we provide a local invariant, and prove it to be local, and for some of them maximal. An invariant is *local* when it refers only to variables modified locally in the reduced loop, and when it can strengthen an inductive invariant over the complete loop. We give conditions for invariants to be local. A local invariant is *maximal* when it is at least as strong as any invariant on the reduced loop. To our knowledge, this is the first work dealing with compositional reasoning on loop invariants, defining modularity and maximality criteria. We also extend the notion of stable variables introduced by Kovács and Voronkov[19].

[3] For the sake of simplicity we omit array bound constraints in generated invariants.

Our technique, applied to a loop L that iterates over the loop index i, can be summarized as follows:

1. We translate L into an intermediate language of parallel assignments, which facilitates both defining patterns and reasoning on local invariants. The translation consists in transforming a sequence of assignments guarded by conditions (if-statements) into a set of parallel assignments of guarded values (if-expressions). This can be done using techniques for efficient computing of static single assignment variables as described in [7, 26]. Due to lack of space, details of the translation are omitted.
2. Using a simple syntactic static analysis, we detect stable [19] scalar and array variables occurring in L. A scalar variable is stable if it is never modified. An array variable is stable on the range $a..b$ if the value of the array between indexes a and b is not modified in the first i iterations (where a and b may refer to the current value of i). We define a *preexisting* invariant over L, denoted $\wp_L$, to express these stability properties.
3. We match our patterns against the intermediate representation of L. We require stability conditions on matched code, which are resolved based on $\wp_L$. For each match involving pattern P_k, we instantiate the corresponding local invariant ϕ_k with variables and expressions occurring in L.
4. We combine all generated local invariants $\phi_1 \ldots \phi_n$ with $\wp_L$ to obtain an inductive invariant on the complete L given by $\wp_L \wedge \phi_1 \wedge \ldots \wedge \phi_n$.

This article is organized as follows. In the rest of this section we survey related work and introduce a running example. Section 2 presents the intermediate language. In Section 3, we introduce reduced loops and local invariants. In Section 4, we define loop patterns as particular instances of reduced loops restricted to some stable expressions. We present four examples of concrete patterns and we provide their corresponding local invariants. In Section 5, we present sufficient criteria for a local invariant to be maximal, and we state maximality results on two concrete pattern invariants. We finally conclude and discuss perspectives in Section 6. Due to lack of space, proofs are omitted but are available in [1].

1.1 Related Work

Most existing techniques generate loop invariants in the form of conjunctions of (in)equalities between polynomials in the program variables, whether by abstract interpretation [6, 24], predicate abstraction [12], Craig's interpolation [22, 23] or algebraic techniques [5, 28, 18]. Various works have defined disjunctive abstract domains on top of the base abstract domains [20, 15, 29].

A few works have targeted the generation of loop invariants with a richer Boolean structure and quantifiers, based on techniques for quantifier-free invariants. Halbwachs and Péron [14] describe an abstract domain to reason about array contents over *simple programs* that they describe as *"one-dimensional arrays, traversed by simple for loops"*. They are able to represent facts like $(\forall i)(2 \leq i \leq n \Rightarrow A[i] \geq A[i-1]$, in which a point-wise relation is established between elements of array slices, where this relation is supported by a quantifier-free base abstract domain. Gulwani *et al.* [13] describe a general lifting procedure that creates a quantified disjunctive abstract domain from quantifier-free domains. They are able to represent facts like $(\forall i)(0 \leq i < n \Rightarrow a[i] = 0)$, in which the formula is universally quantified over an implication between quantifier-free formulas of the base domains. McMillan [21] describes an instrumentation of a resolution-based prover that generates quantified invariants describing facts over simple loops manipulating arrays. Using a similar technique, Kovács and Voronkov [19] generate invariants containing quantifier alternation. Our technique may find a weaker invariant than the previous approaches in some cases (like insertion sort) and a stronger invariant in other cases. The main benefit of our technique is its simplicity and its extensibility: once the loop is converted to a special form of parallel assignment, the technique consists simply in pattern matching on the loop statements, and patterns can be added easily to adapt the technique to new code bases, much like in [8].

1.2 Running Example

We will use the program of Fig. 1 as a running example throughout the paper. A simpler version of this program appears in previous works [3, 19].

```
b := 1; c := 1; erased := 0;
for i in 1..10 while A[i] ≠ 0 do
   if A[i] < 0 then
         B[b] := A[i]; b := b+1;
   else
         C[c] := A[i]; c := c+1;
   end if
   A[i]:=erased;
end
```

Fig. 1. Array partitioning

The program fills an array B with the negative values of a source array A, an array C with the positive values of A, and it erases the corresponding elements from A. It stops at the first null value found in A. As pointed out in [19], there are many properties relating the values of A, B and C before and after the loop, that one may want to generate automatically for this program. In this paper, we show how the different steps of our technique apply to this loop.

2 A Language of Parallel Assignments

In this section we introduce the intermediate language $\mathcal{L}$ and its formal semantics. $\mathcal{L}$ is a refinement of the language introduced in [19] that allows us to group all the assignements performed on the same location in a single syntactic unit.

Fig. 2.(a) presents the syntax of $\mathcal{L}$. In this language, programs are restricted to a single non nested for-like loop (possibly having an extra exit condition) over scalar and one-dimensional array variables. Assignments in $\mathcal{L}$ are performed in parallel. Note that location expressions (e_l) can be either scalar variables or array cells, and that all statements (s_l) of a group ($\mathcal{G}$) assign to the same variable: either the group (only) contains guarded statements $g_k \rightarrow x := e_k$ assigning to some scalar variable x; or it contains statements $g_p \rightarrow A[a_p] := e_p$ assigning to the possibly different cells $A[a_1], A[a_2] \dots$ of some array variable A. A loop body ($\mathcal{B}$) is an unordered collection of groups for different variables.

$\mathcal{L} ::= \textbf{loop}\ i\ \textbf{in}\ \alpha\ ..\ \omega\ \textbf{exit}\ e_b$ $\textbf{do}\ B\ \textbf{end}$	loop
$\mathcal{B} ::= \textbf{skip} \mid \mathcal{G}(\parallel \mathcal{G})^*$	body
$\mathcal{G} ::= \{s_l(; s_l)^*\}$	group
$s_l ::= e_b \rightarrow e_l := e_a$	assignment
$e_l ::= x \mid A[e_a]$	location expr
$e_a \in \textbf{Aexp},\ e_b \in \textbf{Bexp}$	

```
loop i in 1..10 exit A[i] = 0 do
     {    A[i] < 0   → B[b] := A[i]}
  || {    A[i] < 0   → b    := b+1 }
  || { ¬(A[i] < 0)   → C[c] := A[i]}
  || { ¬(A[i] < 0)   → c    := c+1 }
  || {       true    → A[i] := erased }
end
```

Fig. 2. (a) Formal syntax of loop programs (b) Running example translation (Fig. 1)

Running example [Step 1: Translation into the intermediate language] The translation of the running example loop (Fig. 1) into $\mathcal{L}$ is given in Fig. 2.(b).

Expressions and variables n, k stand for (non negative) constants of the language; lower case letters x, a are scalar variables; upper-case letters A, C are array variables; v is any variable; e_a is an arithmetic expression; ϵ, e_b, g are Boolean expressions; e is any expression. Subscripted variables x_0 and A_0 denote respectively the initial value of variables x and A.

Informal semantics Groups are executed *simultaneously*: expressions and guards are evaluated *before* assignments are executed. We assume groups and bodies to be *write-disjoint*, and loops to be *well-formed*. A group G is write-disjoint if all its assignments update the same variable, and if for any two different guards g_1, g_2 in G, $g_1 \wedge g_2$ is unsatisfiable. A loop body $B = G_1 \parallel \dots \parallel G_n$ is write-disjoint if all G_k update different variables and if they are all write-disjoint. A loop L is well-formed if its body is write-disjoint. Thus, on each iteration, at most one assignment is performed for each variable. Conditions on guarded assignments are essentially the same as in the work of Kovacs and Voronkov [19], with a slightly different formalism. For simplicity, we require here unsatisfiability of $g_1 \wedge g_2$ for two guards within a group assigning to array A, even in the case where the updated cells for those guards are actually different.

Loop conventions L denotes a loop, B a body, and i is always the loop index. The loop index is not a variable, so it cannot be assigned. For simplicity, we assume that i is increased (and not decreased) after each run through the loop, from its initial value α to its final value ω. We use $\ell_{(\alpha,\omega,\epsilon)}\{B\}$ to abbreviate **loop** i **in** $\alpha..\omega$ **exit** ϵ **do** B **end**, and $\ell_{(\alpha,\omega)}\{B\}$ when $\epsilon = \mathit{false}$. $\overrightarrow{G}$ denotes a body $G_1 \parallel \dots \parallel G_n$ (for some n), while $\overrightarrow{G} \parallel B$, is the parallel composition of groups $G_1, \dots G_n$ from G with all groups from B. $\{\overrightarrow{g_k \rightarrow l_k := e_k}\}$ denotes a group made of the guarded assignments $\{g_1 \rightarrow l_1 := e_1; \ldots ; g_n \rightarrow l_n := e_n\}$. $\mathcal{G}(B)$ denotes the set of groups occurring in B.

Loop variables $V(L)$ is the set of variables occurring in L (note that $i \notin V(L)$). $V_w(L)$ is the set of variables assigned in L, referred to as local (to L). $V_{nw}(L)$ is the set of variables occuring in L but not assigned in L, referred to as external (to L): $V_{nw}(L) = V(L) - V_w(L)$. Given a set of variables V, the *initialisation predicate* ι_V is defined as $\iota_V = \bigwedge_{v \in V} v = v_0$ asserting that all variables $v \in V$ have as initial (abstract) value v_0. Sets and formulas defined on the loop L are similarly defined on the loop body B.

Quantifications, substitutions and fresh variables ϕ, ψ, ι and $\wp$ denote formulas. The loop index i may occur in the formula ϕ or in the expression e, respectively denoted $\phi(i)$ or $e(i)$, but it can be omitted when not relevant. Except for logical assertions (*i.e.* invariants, Hoare triples), formulas are implicitly universally quantified on the set of all their free variables, including i. To improve readability, these quantifications are often kept implicit. We denote by $\exists V.\phi$ the formula $\exists v_1 \ldots v_n.\phi$ for all $v_i \in V$, and by $[V_1 \leftarrow V_2]$ the substitution of each variable of the set V_1 by the corresponding variable of the set V_2. Given a set of variables V, V' denotes the set containing a fresh variable v' for each variable $v \in V$. Given an expression e, we denote $e'^V = e[V \leftarrow V']$ and $\phi'^V = \phi[V \leftarrow V']$.

2.1 Strongest Postcondition Semantics

The predicate transformer sp introduced by Dijkstra [9, 10] computes the strongest postcondition holding after the execution of a given statement. We shall use it to obtain the strongest postcondition holding after the execution of an arbitrary iteration of the loop body, which will be useful when comparing loop invariants according to maximality criteria (see Section 5). Thus, we express the semantics of the intermediate language $\mathcal{L}$ through the formal definition of sp. As our goal is the generation of loop invariants, and not the generation of loop postconditions, we only need to describe sp for loop bodies, instead of giving it for entire loops in $\mathcal{L}$. Note that Definition 1 requires replacing a variable v assigned in the loop body with a fresh logical variable v', standing for the value of v prior to the assignment.

Definition 1 (Predicate Transformer sp**).** *Let ϕ be a formula, $\overrightarrow{G_k}$ a loop body, and $V = V_w(\overrightarrow{G_k})$. We define* $\mathrm{sp}(\overrightarrow{G_k}, \phi)$ *as:*

$$\mathrm{sp}(\textbf{\textit{skip}}, \phi) = \phi \qquad\qquad \mathrm{sp}(\overrightarrow{G_k}, \phi) = \exists V'.\ \big(\phi'^V \wedge \bigwedge_k \mathrm{Psp}(G_k, V)\big)$$

$$\mathrm{Psp}(\{\overrightarrow{g_k \rightarrow x := e_k}\}, V) = \bigwedge_k \big(g_k'^V \Rightarrow x = e_k'^V\big) \ \wedge\ \left(\left(\bigwedge_k \neg g_k'^V\right) \Rightarrow x = x'\right)$$

$$\begin{aligned}\mathrm{Psp}(\{\overrightarrow{g_k \rightarrow A[a_k] := e_k}\}, V) = {} & \bigwedge_k \big(g_k'^V \Rightarrow A[a_k'^V] = e_k'^V\big) \\ & \wedge \forall j. \left(\bigwedge_k \neg\big(g_k'^V \wedge j = a_k'^V\big)\right) \Rightarrow A[j] = A'[j].\end{aligned}$$

3 Reduced Loops and Local Invariants

Remember that we seek to infer local properties over code pieces occurring in a loop L. In this section, we introduce *reduced loops*, which are loops built on groups taken from a loop L, and *local loop invariants*, which are inductive properties holding *locally* on reduced loops. We state a compositionality result for locally inferred invariants allowing us to compose them into an inductive invariant that holds on the entire loop. Our notion of local invariant is generic: it is not limited to the stability properties used by patterns in Section 4.

3.1 (Inductive) ι_L-Loop Invariants

To define inductive loop invariants, we rely on the classical relation $\models_{\text{par}}$ of satisfaction under partial correctness for Hoare triples [17, 25]. Invariants are defined relative to a given initialisation predicate ι_L providing initial values to loop variables. We define $\iota_L = \iota_V$, where V is the set of all variables occurring in L. An ι_L-loop invariant is an inductive loop invariant under ι_L initial conditions. Also, we say that ι_L *covers* ϕ when $V(\phi) \subseteq V(\iota_L)$. In the following, we assume that the initialisation predicate ι_L covers all properties stated on L.

Definition 2 ((Inductive) ι_L-Loop Invariant). *Assume ι_L covers a formula ϕ. ϕ is an ι_L-loop invariant on the loop $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$, iff*
(a) $(i = \alpha \wedge \iota_L) \Rightarrow \phi$; and (b) $\models_{\text{par}} \{\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi\}\ B;\ i := i + 1\ \{\phi\}$.

3.2 Local (Reduced) Loop Invariants

A *reduced loop* from a loop $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$, is a loop with the same index range as L but whose body B_r is a collection of groups occuring within B (*i.e.* $\mathcal{G}(B_r) \subseteq \mathcal{G}(B)$). These loops either take the form $L_r = \ell_{(\alpha,\omega,\epsilon)}\{B_r\}$ or $L_r = \ell_{(\alpha,\omega)}\{B_r\}$. Remember that each group brings together all assignements of a unique variable. Quite naturally, we seek inferring properties restricted to the locally modified variables of reduced loops. Thus, we distinguish between variables updated within reduced loops, called *local*, and variables appearing without being assigned within them, called *external*.

To deduce properties holding locally on L_r, we assume given an inductive loop invariant $\wp_L$ holding on the entire loop, that states properties over variables external to L_r. Thus, we use a global pre-established property on external variables in order to deduce local properties over local variables. The notion of relative-inductive invariants, borrowed from [4], captures this style of reasoning: ϕ is inductive relative to another formula $\wp_L$, when the inductive step of the proof of ϕ holds under the assumption of $\wp_L$ (see Example 1 below).

Definition 3 (Relative Inductive Invariant). *Assume ι_L covers a formula ϕ. ϕ is $\wp_L$-inductive on loop L, if*
(1) $(i = \alpha \wedge \iota_L) \Rightarrow \phi$; (2) $\text{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \wp_L(i) \wedge \phi(i)) \Rightarrow \phi(i + 1)$.

ϕ is a $\wp_L$-*local loop invariant* on loop L_r, if ϕ only refers to variables locally modified in L_r, and if ϕ holds inductively on L_r relatively to the property $\wp_L$.

Definition 4 ($\wp_L$-Local Loop Invariant). *ϕ is a $\wp_L$-local loop invariant for loop L_r if (a) $V(\phi) \subseteq V_w(L_r)$; and (b) ϕ is $\wp_L$-inductive on L_r.*

```
a := 0; b := 0;
loop i in 1..10 do
    b := a+1;
    a :=i;
end
```

```
init: $\iota_L = (a = 0 \wedge b = 0)$
previous: $\wp_L = (a = i - 1)$
 loop i in 1..10 do
       { true → a :=i }
  ||   { true → b := a+1 }
 end
```

```
$L_r$ = loop i in 1..10 do
    { true → b := a+1 }
 end
local: $\boxed{\phi_r = (b = i - 1)}$
final global inv: $\boxed{\wp_L \wedge \phi_r}$
```

Fig. 3. (a) Loop L (b) Init, previous, translation (c) Reduced loop L_r, local prop.

Example 1 (A $\wp_L$-local loop invariant). Fig. 3.(a) shows a loop L, whose translation and initialisation ι_L are given in 3.(b). The reduced loop L_r in 3.(c) is built on the group that assigns to b. There are two variables in L_r: a is external, while b is local to it. We take $\wp_L$ shown in 3.(b) as previously known property (over variables external to L_r). Clearly, $\wp_L$ does not hold on the reduced loop L_r, but is does hold as ι_L-loop invariant on the entire loop L. The local property $\phi_r(i)$ from 3.(c) does not hold (inductively) by itself on the reduced loop, yet $\wp_L \wedge \phi_r(i)$ holds as inductive invariant of L_r. Therefore, $\phi_r(i)$ is $\wp_L$-inductive on L_r. Moreover, as $\phi_r(i)$ only contains variables local to L_r, it follows that $\phi_r(i)$ is $\wp_L$-local on L_r. Finally, as $\wp_L$ holds inductively on the entire loop, according to the Theorem 1 below, the composed invariant $\wp_L \wedge \phi_r$ is indeed an ι_L-invariant on the whole loop L.

Informally, the Theorem 1 says that whenever a property $\wp_L$, used to deduce that a local property ϕ holds on a reduced loop, is itself an inductive invariant on the entire loop, then $\wp_L \wedge \phi$ is an inductive invariant of the entire loop.

Theorem 1 (Compositionality of $\wp_L$-Local Invariants). *Assume that loops $L = \ell_{(\alpha,\omega,\epsilon)}\{\overrightarrow{G} \parallel B\}$ and $L_r = \ell_{(\alpha,\omega,\epsilon)}\{B\}$ are well-formed. Assume that (h_1) ϕ_r is a $\wp_L$-local loop invariant on L_r; (h_2) $\wp_L$ is an ι_L-invariant on L. Then, $\wp_L \wedge \phi_r$ is an ι_L-invariant on L.*

4 Stable Loop Patterns

In this section, we introduce the stability property for expressions, and we give sufficient conditions for this property to hold. Stability over expressions generalizes the notion of stablity on variables introduced in [19] (see 4.2). We define $\wp_L$-stable loop patterns, as a particular instance of reduced loops restricted to stable expressions[4]. As examples, we present four concrete patterns and we provide their corresponding local invariants.

[4] More precisely, to expressions whose location expressions defined over external variables are stable.

4.1 Stability on Variables and Expressions

Given an initialisation ι_L , we define the *initial value* of an expression $e(i)$, denoted $e_0(i)$, as the result of replacing any occurrence of a variable x in e, *except* i[5], by its initial value x_0 according to the initialisation ι_L. Informally, an expression e occurring in a loop L, is stable, if on any run through the loop, e is equal to its initial value e_0. Here, we are interested in being able to prove that $e = e_0$ under the assumption of a preexisting inductive loop invariant $\wp_L$.

Definition 5 (Stable Expressions). *An expression $e(i)$ is said to be $\wp_L$-stable in loop L, denoted $\wp_L$-s, if there exists an ι_L-loop invariant $\wp_L$ on L such that:*

$$\wp_L(i) \Rightarrow (e(i) = e_0(i)).$$

The rationale behind stability is that, given a preexisting inductive loop invariant $\wp_L$, a $\wp_L$-stable expression e can be replaced by its initial value e_0 when reasoning on the loop body using the predicate transformer sp.

4.2 Sufficient Conditions for Stability

In this section we generalize the notion of stability over variables introduced in [19], in order to express the following properties:

1. a scalar variable x keeps its initial value x_0 throughout the loop;
2. there exist a constant offset from i, denoted $p(i)$, that corresponds to a valid index for array A, such that every cell value in the array slice $A[p(i) \ldots n]$ is equal to its initial value.

For array A and loop L, these properties are formally expressed by:

$$\begin{aligned} \dashv_x &\equiv \quad x = x_0 && \textsf{Scalar stability} \\ \triangle_{A,p} &\equiv \quad \forall j.(j \geq p(i) \Rightarrow A[j] = A_0[j]) && \textsf{Array p} - \textsf{stability} \end{aligned}$$

If $\triangle_{A,p}$ holds, we say that A is p-stable. When $p(i) = \alpha$ this property is equivalent to $A = A_0$. To increase readability, the latter notation is preferred.

A sufficient condition for a variable to be stable is when this variable is not updated at all in the loop. An array B in this case verifies the property $\triangle_{B,\alpha}$. Finding p-stability on some array A can be done by examining all updates to cells $A[p_k(i)]$ and choosing $p(i)$ as $p(i) = max(\overrightarrow{p_k(i)})$. Assume now that array A is known to be p-stable, and that $A[a]$ occurs in some expression e. If $A[a]$ corresponds to an access in the stable slice of A, then e is stable, which can be verified by checking that $a \geq p$ is a loop invariant.

Running example [Step 2: Extracting a preexisting global invariant] The variable `erased` is never assigned in this loop, so it is stable. Array A is updated only in cell $A[i]$, entailing i-stability for A. Thus, we can extract the following inductive invariant expressing stability properties for our loop: $\wp_L = \dashv_{\texttt{erased}} \wedge \triangle_{A,i}$.

[5] And except occurrences at array index positions.

4.3 $\wp_L$-Loop Patterns

Given a preexisting inductive loop invariant $\wp_L$, we define loop patterns relative to $\wp_L$, or $\wp_L$-*loop patterns*, as triples $P_n = (L_n, C_n, \phi_n)$, where: L_n is a loop scheme given by a valid loop construction in our intermediate language $\mathcal{L}$; C_n is a list of constraints requiring the $\wp_L$-s property on generic sub-expressions $e_1, e_2 \dots$ of L_n; ϕ_n is an invariant scheme referring only to variables local to L_n.

Fig. 4 presents examples of three concrete loop patterns. For each of them, the corresponding loop scheme is given in the upper-left entry, the constraints in the upper-right entry, and the invariant scheme in the bottom entry. To identify the pattern P_n within the source loop L, L_n must match actual constructions occurring in L, and the pattern constraints must be satisfied. In that case, we generate the corresponding local invariant by instantiating ϕ_n with matched constructions from L.

Theorem 2 establishes that each invariant scheme ϕ_n from Fig. 4 is indeed a $\wp_L$-local invariant on its corresponding loop scheme L_n. By the compositional result of Theorem 1, each generated local invariant can be composed with the preexisting ι_L-invariant to obtain a richer ι_L-invariant holding on the entire loop.

1. Search Pattern	
$L_1 = \ell_{(\alpha,\omega,\epsilon)}\{\mathbf{skip}\}$	ϵ is $\wp_L$-s.
$\phi_1(i) = \forall j. \alpha \le j < i \Rightarrow \neg\epsilon_0(j)$	

2. Single Map Pattern	
$L_2 = \ell_{(\alpha,\omega)}\{B_2\}$ $B_2 = \texttt{true} \rightarrow A[i] := e(i)$	$e(i)$ is $\wp_L$-s.
$\phi_2(i, A) = \forall j.(\alpha \le j < i \Rightarrow A[j] = e_0(j)) \wedge \forall j. j \ge i \Rightarrow A[j] = A_0[j]$	

3. Filter Pattern	
$L_3 = \ell_{(\alpha,\omega)}\{B_3\}$ $B_3 = \{g(i) \rightarrow A[v] := e(i)\}$ $\parallel \{g(i) \rightarrow v := v + 1\}$	g, e are $\wp_L$-s.
$\phi_3(i, v, A) = \forall j.(\alpha \le j < i \wedge g_0(j) \Rightarrow \exists k.(v_0 \le k < v \wedge A[k] = e_0(j)))$ $\wedge \forall k_1, k_2. v_0 \le k_1 \le k_2 < v \Rightarrow \exists j_1, j_2. \begin{pmatrix} \alpha \le j_1 \le j_2 < i \wedge A[k_1] = e_0(j_1) \\ \wedge g_0(j_1) \wedge g_0(j_2) \wedge A[k_2] = e_0(j_2) \end{pmatrix}$ $\wedge \forall j.(j \ge v \Rightarrow A[j] = A_0[j])$	

Fig. 4. Three $\wp_L$-Loop Patterns

Theorem 2 (Search, Map and Filter Invariant Schemes are $\wp_L$-local). *For* $n \in [1, 2, 3]$ *assume that* $P_n = (L_n, C_n, \phi_n)$ *corresponds to the patterns given in Fig. 4. Assume having three pairs* $(\iota_{L_n}, \wp_{L_n})$ *satifying each the constraints* C_n *for pattern* P_n*. Then, each* ϕ_n *is a* $\wp_{L_n}$*-local loop invariant on the loop* L_n*.*

Running example [Step 3: Discovering patterns, generating local properties] We take $\wp_L \equiv \exists_{\texttt{erased}} \wedge \triangle_{A,i}$ (see Step 2) as preexisting inductive invariant. By pattern-matching, we can recognize three patterns in L: the Search pattern on line 1; the Single Map pattern on line 6; the Filter pattern, once on lines 2-3, and once again on lines 4-5. We must check that all pattern constraints are respected. First note that $\wp_L$ entails i-stability for A, and therefore the location expression $A[i]$ (occurring in both instances of the Filter pattern) is $\wp_L$-s, as well as expressions $A[i] = 0$ in the Search pattern, and $A[i] < 0$ in the Filter pattern. Finally, $\wp_L$ entails stability of `erased` in the Map pattern. We instantiate the corresponding invariant schemes and obtain the local invariants shown below. Note that $\phi_3(i, b, B)$ and $\phi_3(i, c, C)$ correspond to different instances of the Filter pattern. We unfold only one of them here:

$$\begin{aligned}
\phi_1(i) &= \forall j.\alpha \le j < i \Rightarrow \neg(A_0[i] = 0) \\
\phi_2(i, A) &= \forall j.(\alpha \le j < i \Rightarrow A[j] = \texttt{erased}_0) \ \wedge \ \forall j.(j \ge i) \Rightarrow A[j] = A_0[j] \\
\phi_3(i, c, C) &= \ldots \\
\phi_3(i, b, B) &= \forall j.(\alpha \le j < i \wedge A_0[j] < 0 \Rightarrow \exists k.(b_0 \le k < b \wedge B[k] = A_0[j])) \\
&\quad \wedge \forall k_1, k_2.b_0 \le k_1 \le k_2 < b \Rightarrow \exists j_1, j_2. \begin{pmatrix} \alpha \le j_1 \le j_2 \le i \\ \wedge\ A_0[j_2] < 0 \wedge B[k_1] = A_0[j_1] \\ \wedge\ A_0[j_2] < 0 \wedge B[k_2] = A_0[j_2] \end{pmatrix} \\
&\quad \wedge \forall j.(j \ge b \Rightarrow B[j] = B_0[j])
\end{aligned}$$

Example 2 (A disjunctive/existential pattern example). Fig. 5 provides an example of pattern whose invariant contains disjunctions and existential quantifiers. This pattern typically corresponds to the inner loop in a sorting algorithm. The local invariant obtained for the loop from Fig. 5.(b) is:

$$\begin{aligned}
&((m = m_0) \wedge \forall j.(\alpha \le j < i \Rightarrow \neg(A_0[j] < A_0[m_0]))) \\
\vee\ &(\exists j.(\alpha \le j < i \wedge m = j) \wedge \forall k.(\alpha \le k < i \Rightarrow \neg(A_0[k] < A_0[m]))
\end{aligned}$$

Min Index Pattern	
$L_4 = \ell_{(\alpha,\omega)}\{B_4\}$ $B_4 = \ \{e(i) < e(a) \to a := i\}$	$e(i)$ is $\wp_L$-s.
$\phi_4(i, a) = ((a = a_0) \wedge \forall j.(\alpha \le j < i \Rightarrow \neg(e_0(j) < e_0(a_0))))$ $\vee\ (\exists j.(\alpha \le j < i \wedge a = j \wedge \forall k.(\alpha \le k < i \Rightarrow \neg(e_0(k) < e_0(a))))$	

```
for i in 1..n do
  if A[i] < A[m]
    then m := i;
  end if
end
```

Fig. 5. (a) A pattern with existentials and disjunctions (b) A loop instance

Running example [Step 4: Aggregating local invariants] We know that the pre-existing invariant $\wp_L$ holds as ι_L-invariant on L. By Theorem 2, ϕ_1 is $\wp_L$-local on $L_1 = \ell_{(\alpha,\omega,\epsilon)}\{\textbf{skip}\}$, and ϕ_2 and ϕ_3 are $\wp_L$-local on loops $L_k = \ell_{(\alpha,\omega)}\{B_k\}$ respectively for $k = 1, 2$. It is easy to obtain from these results, that ϕ_2 and ϕ_3

are $\wp_L$-local on loops $L_k = \ell_{(\alpha,\omega,\epsilon)}\{B_k\}$. Therefore, according to Theorem 1, we can compose all these invariants to obtain the following richer ι_L-invariant holding on L: $\wp_L \wedge \phi_1(i) \wedge \phi_2(i, A) \wedge \phi_3(i, b, B) \wedge \phi_3(i, c, C)$.

5 Maximal Loop Invariants

In this section, we present maximality criteria on loop invariants, whether inductive or not. A loop invariant is maximal when it is stronger than any invariant holding on that loop. For consistency, we compare loop invariants only if they are covered by the same initialisation predicate. We adapt this notion to reduced loops by defining *local invariant maximality.* These notions are rather generic and apply to any loop language equipped with a strongest postcondition semantics.

Definition 6 (Maximal ι_L-Loop Invariant). *ϕ is a maximal ι_L-loop invariant of loop L if (1) ϕ is an ι_L-loop invariant for L, and (2) for any other ι_L-loop invariant ψ of L, $\phi \Rightarrow \psi$ is an ι_L-loop invariant of L.*

Theorem 3 (Loop Invariant Maximality). *Let $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$ and assume that ϕ is some formula covered by ι_L. ϕ is a maximal ι_L-invariant of L if*

(a) $i = \alpha \wedge \iota_L \Leftrightarrow i = \alpha \wedge \phi(i)$
(b) $\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon(i) \wedge \phi(i)) \Leftrightarrow \alpha \leq i \leq \omega \wedge \phi(i+1)$

As seen in Section 3, a local invariant ϕ_r refers only to variables locally modified in the reduced loop L_r. Nevertheless, external variables may occur in L_r, for which we are unable to locally infer properties. To ensure consistency when comparing local invariants, we reason on the maximality of ϕ_r, strenghtened by a formula $\dashv$ stating that all variables external to L_r remain constant through the execution of the reduced loop.

Definition 7 (Local Invariant Maximality). *Let $L = \ell_{(\alpha,\omega,\epsilon)}\{\overrightarrow{G} \parallel B\}$ be a well-formed loop, and $L_r = \ell_{(\alpha,\omega,\epsilon)}\{B\}$. Let ι_r be an initialisation restricted to variables occurring in L_r, and $\dashv$ a formula asserting constant values $x = x_0$, $A = A_0$ for all variables x, A external to L_r. We say that ϕ_r is locally maximal on L_r when $\dashv \wedge \phi_r$ is a maximal ι_r-loop invariant of L_r.*

In [1] we provide proofs for the Theorem 4 below. We show that the local loop invariants schemes $\phi_1(i)$ for the Search Pattern, and $\phi_2(i)$ for the Single Map Pattern, as stated in Fig. 4, are indeed locally maximal on their corresponding reduced loop. Notice that $\phi_3(i)$ for the Filter Pattern is not maximal as stated in Fig. 4. For example, it would be possible to state a stronger invariant for this pattern by recursively defining a logic function for counting the number of array elements satisfying the guard g_0 up to the i^{th} element, and using this function in the loop invariant to give the current value of the variable v.

Theorem 4 (Search and Single Map Invariants Local Maximality). *Let ϕ_1, L_1, ϕ_2, L_2 as given in Fig. 4. ϕ_1 is locally maximal on the loop L_1, and ϕ_2 is locally maximal on the loop L_2.*

6 Conclusion and Further Work

We present a novel and compositional approach to generate loop invariants. Our approach complements previous approaches: instead of generating relatively weak invariants on any kind of loop, we focus on generating strong or even maximal invariants on particular loop patterns, in a modular way.

The central idea in our approach is to separately generate local loop invariants on reduced versions of the entire loop. This is supported by the introduction of a preexisting loop invariant, which states external properties (*i.e.* properties which do not necessarily hold locally) on the complete loop. This preexisting invariant is then strengthened by the local loop invariants. Since there is no constraint on the way the external invariant is found, our approach fits in smoothly with other automated invariant generation mechanisms.

We propose a specialized version of reduced loops, for which the external invariant is a stability property of some locally accessed variables. We give loop pattern schemes and syntactic criteria to generate invariants for any loop containing these patterns. Independently, we present conditions on arbitrary loop invariants to be maximal, and state results of local maximality for some of our loop patterns. When not maximal, our inferred invariants are essentially as expressive as those generated by previous approaches, but have the advantage of being pre-proven, and thus are well adapted to integration on full automatic invariant generation of industrial oriented frameworks.

Our method applies to programs in an intermediate language of guarded and parallel assignments, to which source programs should first be translated. We have designed such a translation from a subset of the SPARK language, based on an enriched version of static single assignment form [26]. The idea is to transform a sequence of assignments to variables guarded by conditions (if-statements) into a set of parallel assignments to SSA variables [7], where the value assigned has guard information (if-expressions). In the case of array variables, array index expressions that are literals or constant offsets from the loop index are treated specially, in order to generate array index expressions that can be matched to the patterns we define. This translation is exponential in the number of source code statements in the worst-case, but this does not occur on hand-written code.

We expect to implement the translation and the pattern-based loop invariant generation in the next generation of SPARK tools [16, 30]. We believe that combining this technique with other ones (and with itself) will be very efficient.

Going further we could develop a broader repository of pattern-driven invariants, to address the more frequent and known loop patterns. As proof of patterns (correctness and optionally maximality) are tedious and error-prone, we plan to mechanize them in a proof assistant and design a repository of formally proven patterns. In particular, as the present technology seems perfectly applicable to non terminating loops, we plan to define new patterns for while loops.

The current approach does not handle nested loops, and the patterns we define do not apply to loops with a complex accumulation property, where the effect of the ith iteration depends in a complex way on the cumulative effect of previous iterations. So it does not apply for example to insertion sort, which can be treated by other approaches [14, 13] based on complex abstract domains in abstract interpretation. We are interested in pursuing the approach to treat these more complex examples.

Acknowledgements We would like to thank Laura Kovács for her feedback on an early version of this work, Benjamin Brosgol for his careful review of the manuscript and anonymous referees for their valuable comments.

References

1. V. Aponte, P. Courtieu, Y. Moy, and M. Sango. Maximal and Compositional Pattern-Based Loop Invariants - Definitions and Proofs. Technical Report CEDRIC-12-2555, CEDRIC laboratory, CNAM-Paris, France. `http://cedric.cnam.fr/fichiers/art_2555.pdf`.
2. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming language Design and Implementation*, PLDI '07, pages 300–309, New York, NY, USA, 2007. ACM.
4. A. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Aspects of Computing*, 20:379–405, 2008. 10.1007/s00165-008-0080-9.
5. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Proc. 15th International Conference on Computer Aided Verification*, LNCS, pages 420–432. Springer, 2003.
6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.
7. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.
8. E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 121–130, New York, NY, USA, 2006. ACM.
9. E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.
10. E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics.* Springer, 1990.
11. Escher C Verifier, 2012. `http://www.eschertech.com/products/ecv.php`.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, 1997. Springer-Verlag.
13. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 235–246, New York, NY, USA, 2008. ACM.

14. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 339–348, New York, NY, USA, 2008. ACM.
15. W. R. Harris, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis via satisfiability modulo path programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 71–82, New York, NY, USA, 2010. ACM.
16. Hi-Lite: Simplifying the use of formal methods. `http://www.open-do.org/projects/hi-lite/`.
17. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.
18. L. Kovács. Invariant generation for p-solvable loops with assignments. In *Proceedings of the 3rd International Conference on Computer science: Theory and Applications*, CSR'08, pages 349–359, Berlin, Heidelberg, 2008. Springer-Verlag.
19. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '09, Washington, DC, USA, 2009. IEEE Computer Society.
20. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proc. ESOP'05*, LNCS 3444:5–20.
21. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proceedings of the Theory and Practice of Software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 413–427, Berlin, Heidelberg, 2008. Springer-Verlag.
22. K. L. McMillan. Interpolation and sat-based model checking. In *Proc. 15th International Conference on Computer Aided Verification*, LNCS, pages 1–13. Springer, 2003.
23. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.
24. A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, March 2006.
25. H. R. Nielson and F. Nielson. *Semantics with Applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.
26. K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM.
27. RTCA. Formal methods supplement to DO-178C and DO-278A. Document RTCA DO-333, RTCA, December 2011.
28. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 318–329, New York, NY, USA, 2004. ACM.
29. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV '11, New York, NY, USA, July 2011. ACM.
30. SPARK Pro, 2012. `http://www.adacore.com/home/products/sparkpro/`.