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 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 by recognizing through static analysis simple code patterns that respect stability properties on accessed locations, on an intermediate representation of parallel assignments. 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 [8, 25], and new certification standards recognizing its use [22]. 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 prove automatically 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 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 [1] technology, these are seldom the problematic loop invariants. Indeed, users are well aware of the arithmetic properties that they maintain through loops, and thus have no difficulty manually annotating loops with the desired arithmetic invariants. Instead, users very often have difficulties annotating loops with *obvious* invariants, that they do not recognize as required for inductive reasoning. These invariants 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 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 modular, and for some of them maximal. A local invariant is *modular* when it can strengthen an inductive invariant over the complete loop. We give conditions for local invariants to be modular. A local invariant is *maximal* when it is stronger than 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[15].

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

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

1. We translate L into an intermediate language of parallel assignments, which facilitates both defining patterns and reasoning on local invariants.
2. Using a simple syntactic static analysis, we detect stable [15] 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 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 modular invariants. In section 4, we define loop patterns as particular instances of reduced loops restricted to stable variables. We present three concrete patterns and we provide the corresponding modular invariants. In section 5, we present sufficient criteria for a local invariant to be maximal, and we state maximality results on our three concrete pattern invariants. We finally conclude and discuss perspectives in section 6.

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 [5, 20], predicate abstraction [9], Craig's interpolation [18, 19] or algebraic techniques [4, 23, 14]. Various works have defined disjunctive abstract domains on top of base abstract domains [16, 12, 24].

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 [11] 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.* [10] describes 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 [17] 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[15] generate invariants containing quantifier alternation.

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 [2, 15]. 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 [15], 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. In this case, it even generates the most precise loop invariant.

```
b := 1; c := 1; erased := 0;
for i in 1..10 while A[i] ≠ 0 loop
   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 loop
```

Fig. 1. Array partitioning

2 A Language of Parallel Assignments

In this section we introduce the intermediate language $\mathcal{L}$ and its formal semantics, as well as notation used throughout this article.

2.1 Syntax

Fig. 2.(a) presents the intermediate language $\mathcal{L}$. In this language, programs are restricted to a simple for-like loop (possibly having an extra exit condition) over scalar and one-dimensional array variables. Assignments in $\mathcal{L}$ are performed in parallel.

$\mathcal{L}$ ::= **loop** i **in** α .. ω **exit** e_b **do** B **end**	loop
$\mathcal{B}$ ::= **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
$e_a, e_b \in$ **Aexp**, **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)

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.

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 (abstract) 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 [15], with a slightly different formalism. Note that, 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 on 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 also assume it is increased (and not decreased) after each run through the loop, from its initial value α to its final value ω. We abbreviate the construction **loop** i **in** $\alpha..\omega$ **exit** ϵ **do** B **end** by $\ell_{(\alpha,\omega,\epsilon)}\{B\}$, and by $\ell_{(\alpha,\omega)}\{B\}$ when $\epsilon = \mathit{false}$. We use $\{\overrightarrow{G_k}\}$ to denote the loop body $\{G_1 \parallel \dots \parallel G_n\}$. Similarly, $\{\overrightarrow{g_k \rightarrow l_k := e_k}\}$ denotes a group made of the n guarded assignments $\{g_1 \rightarrow l_1 := e_1; \dots ; g_n \rightarrow l_n := e_n\}$. $\mathcal{G}(B)$ 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 $\iota_V \equiv \bigwedge_{v \in V} v = v_0$ asserting that all variables $v \in V$ have their initial (abstract) value v_0. Sets and formulas defined on loop L are similarly defined on the loop body B.

Quantifications and substitutions ϕ, ψ, ι and $\wp$ denote formulas. If the loop index i occurs in the formula ϕ or the expression e, then they are noted $\phi(i)$ or $e(i)$, but this 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.

2.2 Strongest Postcondition Semantics

The predicate transformer sp introduced by Dijkstra [6, 7] computes the strongest postcondition holding after the execution of a given statement. We shall use it to compute 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 a formal definition of sp. In definition 1, we give a syntactic formulation of sp. As our goal is the generation of loop invariants, and not the generation of loop postcondtions, we only need to describe sp for loop bodies, instead of giving it for entire loops in $\mathcal{L}$.

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

$$\mathrm{sp}(\boldsymbol{skip}, \phi) = \phi \quad \textit{and} \quad \mathrm{sp}(\{\overrightarrow{G_k}\}, \phi) = \exists(V').\left(\phi'^{V} \wedge \left(\bigwedge_k \mathrm{Psp}(G_k, V)\right)\right)$$

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

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

sp definition 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. Given a set of variables V, we denote by V' the set containing a fresh variable v' for each variable $v \in V$. Given an expression e, we denote by $e'^{V} \equiv e[V \leftarrow V']$. A similar substitution is defined on predicate ϕ and denoted ϕ'^{V}. The property SP-MONO taken from [21] and corollary 1 will be used in several proofs.

Corollary 1 (Renaming of External Variables in sp**).** *Let* $L = \ell_{(\alpha,\omega,\epsilon)}\{\overrightarrow{G} \parallel B\}$ *be a well-formed loop. Let* $V_G = V_w(\overrightarrow{G})$ *and* $V_B = V_w(B)$. *Then,*

$$\mathrm{Psp}(B, V_B \cup V_G) = (\mathrm{Psp}(B, V_B))'^{V_G}.$$

Corollary 2 (Monotonicity of sp**).** *Given formulas* P, Q *and statement* C,

$$(P \Rightarrow Q) \Rightarrow (\mathrm{sp}(C, P) \Rightarrow \mathrm{sp}(C, Q)) \qquad [\text{SP-MONO}]$$

3 Reduced Loops and Modular Invariants

In this section, we define reduced loops, which are smaller versions of some loop L, and modular loop invariants. A local invariant over a reduced loop is modular when it can strengthen a preexisting inductive invariant $\wp_L$ over the complete loop. Our notion of modularity is generic with respect to $\wp_L$. In particular, it is not limited to the stability properties that we use in our patterns in section 4.

3.1 (Inductive) ι_L-Loop Invariants

We rely on the classical relation $\vDash_{\mathrm{par}}$ of satisfaction under partial correctness [13, 21] of Hoare triples to define inductive loop invariants. Invariants are defined relative to a given initialisation predicate providing initial (abstract) values to loop variables, defined as $\iota_L \equiv \iota_V$, 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). *ϕ is an ι_L-loop invariant on the loop $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$, iff*
(a) ι_L covers ϕ; (b) $(i = \alpha \wedge \iota_L) \Rightarrow \phi$; and
(c) $\vDash_{\mathrm{par}} \{\alpha \le i \le \omega \wedge \neg\epsilon \wedge \phi\}\ B;\ i := i+1\ \{\phi\}$.

Suppose we want to state that some ψ is an ι_L-loop invariant of $\ell_{(\alpha,\omega,\epsilon)}\{B\}$. We shall use the following lemma, whose proof is omitted due to lack of space.

Lemma 1 (ι_L-Loop Invariant Definition via sp). *ψ is an ι_L-loop invariant on loop $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$ iff:*
(a) ι_L covers ψ; (b) $i = \alpha \wedge \iota_L \Rightarrow \psi(\alpha)$;
(c) $\mathrm{sp}(B, \alpha \le i \le \omega \wedge \neg\epsilon \wedge \psi(i)) \Rightarrow \psi(i+1)$.

3.2 Modular (Reduced) Loop Invariants

A *reduced loop* from a given loop $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$, is a loop with same index range as L but whose body B_r occurs within B (*i.e.* $\mathcal{G}(B_r) \subseteq \mathcal{G}(B)$). These loops take the forms $L_r = \ell_{(\alpha,\omega,\epsilon)}\{B_r\}$ or $L'_r = \ell_{(\alpha,\omega)}\{B_r\}$, and are noted $L_{\downarrow L_r}$ and $L_{\downarrow L'_r}$. *Local variables* are variables updated in reduced loops $L_{\downarrow L_r}$, and *external variables* are variables appearing without being assigned in $L_{\downarrow L_r}$. To deduce properties on $L_{\downarrow L_r}$, we assume that an inductive loop invariant $\wp_L$ states properties over variables external to $L_{\downarrow L_r}$. The notion of relative-inductive invariants, borrowed from [3], captures this style of reasoning: ϕ is inductive relative to another formula $\wp_L$, on loop L, when the inductive step of the proof of ϕ holds under the assumption $\wp_L$.

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

ϕ is a $\wp_L$-*modular 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 ($\wp_L$-Modular Loop Invariant). *ϕ is a $\wp_L$-modular 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**
    a :=i ;
    b := a+1;
end
	init : $\iota_L = (a = 0 \wedge b = 0)$
	external: $\wp_L = (a = i - 1)$
	loop i **in** 1..10 **do**
	    { **true** → a :=i }
	  $\Vert$ { **true** → b := a+1 }
	end
		loop i **in** 1..10 **do**
		    { **true** → b := a+1 }
		end
		local: $\boxed{\phi_b = (b = i - 1)}$
		final global inv: $\boxed{\wp_L \ \wedge \ \phi_b}$

Fig. 3. (a) Loop L (b) Previous properties (c) Reduced loop $L_{\downarrow L_b}$

Example 1 (A $\wp_L$-modular loop invariant). Consider the loop L and formulas given in Fig. 3. ι_L corresponds to initialisation conditions (with concrete values, for illustration). Let us call $L_{\downarrow L_b}$ the loop reduced to the group assigning to b in L, given in Fig. 3.(c). We take $\wp_L$ as a previously known property over variables external to $L_{\downarrow L_b}$. Clearly, $\wp_L$ is an ι_L-loop invariant on L, but it does not hold on the reduced loop $L_{\downarrow L_b}$. $\phi_b(i)$ does not hold inductively by itself on the reduced loop, yet $\wp_L \wedge \phi_b(i)$ is an inductive invariant of $L_{\downarrow L_b}$. Therefore, $\phi_b(i)$ is $\wp_L$-inductive on $L_{\downarrow L_b}$. Moreover, $\phi_b(i)$ only refers to variable b modified locally in $L_{\downarrow L_b}$. It follows that $\phi_b(i)$ is $\wp_L$-modular on $L_{\downarrow L_b}$. Additionally, as $\wp_L$ holds inductively on the entire loop, according to the theorem 1 below, the strengthened invariant $\wp_L \wedge \phi_b$ is indeed an ι_L-invariant on the whole loop L.

Informally, 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 $\wp_L$-Modular Invariants). *Assume the loops $L = \ell_{(\alpha,\omega,\epsilon)}\{\overrightarrow{G_k} \parallel B\}$ and $L_B = \ell_{(\alpha,\omega,\epsilon)}\{B\}$ are well-formed. Assume that (h_1) ϕ is a $\wp_L$-modular loop invariant on L_B; (h_2) $\wp_L$ is an ι_L-invariant on L. Then, $\wp_L \wedge \phi$ is an ι_L-invariant on L.*

Proof. Following lemma 1, $\wp_L \wedge \phi$ is an ι_L-invariant of L, if: (a) ι_L covers $\wp_L \wedge \phi$; (b) $i = \alpha \Rightarrow \wp_L \wedge \phi$; and (c) $A \Rightarrow \phi(i+1) \wedge \wp_L(i+1)$ where $A = \mathrm{sp}(\{\overrightarrow{G_k} \parallel B\}, \alpha \le i \le \omega \wedge \neg\epsilon \wedge \wp_L(i) \wedge \phi(i))$. From ($\mathrm{h}_1$) conditions (a) and (b) hold by definition. By (h_2) and lemma 1 we know that $A \Rightarrow \wp_L(i+1)$. Thus, we only need to prove $A \Rightarrow \phi(i+1)$. We use sp definition to develop A and deduce (by forgetting $\mathrm{Psp}(\overrightarrow{G_k}, V)$):
$A \Rightarrow \exists V'.(\alpha \le i \le \omega \wedge \neg\epsilon(i)^{\prime V} \wedge \wp_L(i)^{\prime V} \wedge \phi(i)^{\prime V} \wedge \mathrm{Psp}(B, V))$,

where $V_G = V_w(\overrightarrow{G_k})$, $V_B = V_w(B)$ and $V = V_G \cup V_B$. By corollary 1, we can replace $\mathrm{Psp}(B, V)$ by $(\mathrm{Psp}(B, V_B))^{\prime V_G}$. Moreover, by hypothesis of well-formedness on L, we know that $V_G \cap V_B = \emptyset$. Therefore, any predicate $P^{\prime V}$ can be written as $(P^{\prime V_B})^{\prime V_G}$ and we obtain (1) below, where $C \equiv \mathrm{sp}(B, \alpha \le i \le \omega \wedge \neg\epsilon \wedge \wp_L(i) \wedge \phi(i))$, which can be expanded to $C \equiv \exists V'_B.\alpha \le i \le \omega \wedge \neg\epsilon^{\prime V_B} \wedge \wp_L^{\prime V_B} \wedge \phi^{\prime V_B} \wedge \mathrm{Psp}(B, V_B)$. On the other hand, by (h_1) we also have (2) below:

$$A \Rightarrow \exists V'_G . C'^{V_G} \quad (1) \qquad\qquad C \Rightarrow \phi(i+1) \quad (2)$$

To conclude, we need to rewrite (1) and (2) with explicit universal quantifications on the free variables $\vec{x}$ of these formulas (see 2.1):

$$\forall \vec{x}, A \Rightarrow \exists V'_G . C'^{V_G} \quad (1') \qquad\qquad \forall \vec{x}, C \Rightarrow \phi(i+1) \quad (2')$$

We now prove that $\forall \vec{x}, A \Rightarrow \phi(i+1)$. Suppose that for some $\vec{a}$, $A[\vec{x} \leftarrow \vec{a}]$ holds, let us prove that $\phi(i+1)[\vec{x} \leftarrow \vec{a}]$ also holds. By $(1')$ we have: $\exists V'_G . (C'^{V_G}[\vec{x} \leftarrow \vec{a}])$. Therefore there exists $v_1 \dots v_n$ such that $C'^{V_G}[\vec{x} \leftarrow \vec{a}][V'_G \leftarrow \vec{v_i}]$ holds, which is identical to $C'^{V_G}[V'_G \leftarrow \vec{v_i}][\vec{x} \leftarrow \vec{a}]$ which is itself identical to $C[V_G \leftarrow \vec{v_i}][\vec{x} \leftarrow \vec{a}]$. We can now apply $(2')$ and deduce $\phi(i+1)[V_G \leftarrow \vec{v}][\vec{x} \leftarrow \vec{a}]$. Since $V_G \cap V(\phi) = \emptyset$ we finally have $\phi(i+1)[\vec{x} \leftarrow \vec{a}]$. □

4 Value Preserving Loop Patterns

In this section, we introduce the value preservation property for expressions, and we give sufficient conditions for expressions to be value preserving. We define $\wp_L$-value preserving loop patterns, as a particular instance of reduced loops restricted to value preserving expressions[4]. We present three concrete patterns and we provide the corresponding modular invariants.

4.1 Value Preservation

Informally, an expression e occurring in loop L is value preserving 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 (Initial Value by ι_l). *The initial value of expression $e(i)$ by initialisation ι_L, noted $e_0(i)$, is the result of replacing any variable x in e,* except *i, by its initial value x_0 according to ι_L:* $\quad e_0(i) \overset{def}{=} e(i)[x \leftarrow \iota_L(x)]$.

Definition 6 (Initial value preservation). *An expression $e(i)$ is said to be $\wp_L$-vp in loop L 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 value preservation is that, given a preexisting inductive loop invariant $\wp_L$, a $\wp_L$-value preserving expression e can be replaced by its initial value e_0 when reasoning on the loop body using sp.

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

4.2 Sufficient Conditions for Value Preservation

In this section, we generalize the notion of stability introduced in [15], 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 position $p(i)$ in array A, which can be expressed as a constant offset from i, such that every cell value in the array slice $A[p(i) \dots n]$ is equal to its initial value.

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

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

If $\triangle_{A,p}$ holds, we say that A is p-stable. When $p = \alpha$ 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 $A[p_k]$ and choosing p as $p = max(\overrightarrow{p_k})$. 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 value preserving, 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 for our loop: $\wp_L = \dashv_{erased} \wedge \triangle_{A,i}$.

4.3 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 $P_n = (L_n, C_n, \phi_n)$. L_n is a loop scheme given by a valid loop construction in $\mathcal{L}$; C_n is a list of constraints requiring $\wp_L$-vp property on generic sub-expressions $e_1, e_2 \dots$ of L_n; ϕ_n is a local invariant referring only to variables local to L_n.

Fig. 4 presents 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. We establish in lemmas 2, 3 and 4 that the local property ϕ_n is indeed a $\wp_L$-modular invariant on the reduced loop $L_{\downarrow L_n}$, for each of the three loop patterns presented here. Thus, according to the compositional result given in theorem 1, each generated local invariant can strengthen the preexisting ι_L-invariant $\wp_L$ to obtain a richer ι_L-invariant for loop L.

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

2. Single Map Pattern	
$L_2 = \ell_{(\alpha,\omega)}\{B_2\}$ $B_2 = \texttt{true} \rightarrow A[i] := e(i)$	$e(i)$ is $\wp_L$-vp.
$\phi_2(i, A) = \begin{array}{l} \forall j.(\alpha \leq j < i \Rightarrow A[j] = e_0(j)) \wedge \\ \forall j.(j \geq i) \Rightarrow A[j] = A_0[j] \end{array}$	

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

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

Running example [Step 3: Discovering patterns, generating local properties] We take $\wp_L \equiv \dashv_{erased} \wedge \triangle_{A,i}$ (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 $\wp_L$-vp, 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 in Fig. 5. 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.

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

Fig. 5. Generated local invariants for the running example

In the following we provide lemmas stating the modularity of the pattern properties given in Fig. 4. The proof of lemma 4 is omitted.

Lemma 2 (Search Pattern Invariant Modularity). *Given $\iota_L, \wp_L$ such that ϵ is $\wp_L$-vp, $\phi_1(i)$ is a $\wp_L$-modular loop invariant on L_1, for $\phi_1(i), L_1$ from fig. 4.*

Proof. By definition 4 we need to prove that $V(\phi_1) \subseteq V_w(L_1)$, which is trivial since $V(\phi_1) = \emptyset$, and that ϕ_1 is $\wp_L$-inductive. By definition 3 this amounts to proving that (1) $i = \alpha \wedge \iota \Rightarrow \phi_1$ and (2) $\mathrm{sp}(\mathbf{skip}, \alpha \le i \le \omega \wedge \neg\epsilon \wedge \wp_L(i) \wedge \phi_1(i)) \Rightarrow \phi_1(i+1)$. Since $i = \alpha \wedge \iota_L$ implies that $\alpha \le j < i$ is false, (1) holds. Let us prove (2) by unfolding the definition of sp:

$$\mathrm{sp}(\mathbf{skip}, \alpha \le i \le \omega \wedge \neg\epsilon(i) \wedge \wp_L(i) \wedge \phi_1(i)) = \alpha \le i \le \omega \wedge \neg\epsilon(i) \wedge \wp_L(i) \wedge \phi_1(i)$$

which implies $\phi(i)_1$ and $\neg\epsilon_0(i)$ because ϵ is $\wp_L$-vp. Therefore $\phi_1(i+1)$ holds. □

Lemma 3 (Single Map Pattern Invariant Modularity). *Given $\iota_L, \wp_L$ such that $e(i)$ is $\wp_L$-vp, $\phi_2(i, A)$ is a $\wp_L$-modular invariant of L_2, for $\phi_2(i, A)$ and L_2 as given in fig. 4.*

Proof. By definition 4 and 3 we have to prove (ϵ is false in this pattern):

- $V(\phi_2) \subseteq V_w(L_2)$ which follows from $V(\phi_2) = \{A\}$;
- $i = \alpha \wedge \iota_L \Rightarrow \phi_2(i, A)$, which follows from $i = \alpha \wedge \iota_L \Rightarrow \phi_2(\alpha, A_0)$ and as $\iota_L \Rightarrow (A = A_0)$;
- $\mathrm{sp}(B_2, \alpha \le i \le \omega \wedge \wp_L(i, A) \wedge \phi_2(i, A)) \Rightarrow \phi_2(i + 1)$ which we prove below.

Suppose that $\mathrm{sp}(B_2, \alpha \le i \le \omega \wedge \wedge \wp_L(i, A) \wedge \phi_2(i, A))$ holds and let us prove that $\phi_2(i + 1, A)$ holds. By definition of sp there exists A' such that:

(a) $\wp_L(i, A')$ (b) $\phi_2(i, A')$ (c) $A[i] = e(i, A')$ (d) $\forall j. j \ne i \Rightarrow A[j] = A'[j]$

Moreover, the pattern constraint ($e(i)$ is $\wp_L$-vp) and (a) entails $e(i, A') = e_0(i, A_0)$. By (c) and (d) we know that A and A' differ only on cell $A[i]$ which contains $e(i, A')$ which allows to prove easily that $\phi_2(i + 1, A)$ also holds. □

Lemma 4 (Filter Pattern Invariant Modularity). *For all $\iota_L, \wp_L$ such that $g(i)$ and $e(i)$ are $\wp_L$-vp, $\phi_3(i, v, A)$ as given in figure 4 is a $\wp_L$-modular loop invariant of the loop L_3 of figure 4.*

Running example [Step 4: Aggregating local modular invariants] We know that the preexisting invariant $\wp_L$ holds as ι_L-invariant on L. Also, by lemma 2, ϕ_1 is $\wp_L$-modular on $L_1 = \ell_{(\alpha,\omega,\epsilon)}\,\{\mathbf{skip}\}$, and by lemmas 3 and 4 ϕ_2 and ϕ_3 are $\wp_L$-modular on loops $L_k = \ell_{(\alpha,\omega)}\,\{B_k\}$ respectively for $k = 1, 2$. It is easy to obtain from these results, and using property SP-MONO, that ϕ_2 and ϕ_3 are $\wp_L$-modular on loops $L_k = \ell_{(\alpha,\omega,\epsilon)}\{B_k\}$. Therefore, we can apply the theorem 1, and 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 local loop invariants. A local invariant is maximal when it is stronger than any invariant on the reduced loop. For consistency, we compare loop invariants only if they are covered by the same initialisation predicate. Our notion of loop invariant maximality is independent of the language chosen to write those loops: it can be applied to any loop language equipped with a strong postcondition semantics. We show that the loop invariants we defined in section 4, for the three concrete patterns we introduced, are indeed maximal.

Definition 7 (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 2 (Loop Invariant Maximality). *Let $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$ and assume that ϕ is some formula. ϕ is a maximal ι_L-invariant of L if*

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

Definition 8 (Local Invariant Maximality). *Let $L = \ell_{(\alpha,\omega,\epsilon)}\{\overrightarrow{G} \parallel B\}$ be a well-formed loop, and $L_{\downarrow L_B}$ a loop reduced to body B. Let ι_B be an initialisation restricted to variables occurring in $L_{\downarrow L_B}$, and ι_V an initialisation restricted to variables external to $L_{\downarrow L_B}$ ($V = V_{nw}(B)$). If $\iota_V \wedge \phi_B$ is a maximal ι_B-loop invariant on $L_{\downarrow L_B}$, then ϕ_B is locally maximal on the reduced loop $L_{\downarrow L_B}$.*

It can be shown that the loop invariants $\phi_1(i), \phi_2(i), \phi_3(i)$ given in Fig. 4 for our patterns, and holding as $\wp_L$-modular loop invariants respectively on reduced loops $L_{\downarrow L_1}$, $L_{\downarrow L_2}$ and $L_{\downarrow L_3}$, are additionally locally maximal on each one of these loops. We claim local maximality for the Single Map pattern invariant.

Lemma 5 (Single Map Local Maximality). *Let L be a well-formed loop. ϕ_2 is locally maximal on the reduced loop $L_{\downarrow L_2}$ for ϕ_2, where L_2 is as defined in Fig. 4.*

Running example [Maximality of the generated loop invariant] When a loop L is totally matched by $\wp_L$-patterns, we can show that the obtained ι_L-invariant is maximal on L. The details of the proof are beyond the scope of this article. As a corollary, the result invariant generated in section 4.3 is ι_L-maximal on L.

6 Conclusion

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 maximal invariants on particular loop patterns, in a modular way. 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. The central idea in our approach is to generate local modular loop invariants on reduced versions of the entire loop. This is supported by the introduction of a preexisting loop invariant strengthened by local modular loop invariants, which states external properties (*i.e.* properties which do not necessarily hold locally) on the complete loop. This gives us the power to reuse and compose invariants obtained locally, as long as the external property used to deduce them is itself an inductive invariant. 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. Going further we could develop a repository of pattern-driven (proven maximal) invariants, to address frequent and known loop patterns. We expect that combining this technique with other ones (and with itself) will be very efficient. Independently, we present conditions on arbitrary loop invariants to be maximal, and state results of local maximality for our loop patterns.

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

References

1. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
2. 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.
3. A. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Aspects of Computing*, 20:379–405, 2008. 10.1007/s00165-008-0080-9.
4. 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.
5. 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.
6. E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.
7. E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics.* Springer, 1990.

8. Escher C Verifier, 2012. `http://www.eschertech.com/products/ecv.php`.
9. 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.
10. 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.
11. 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.
12. 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.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.
14. 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.
15. 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.
16. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proc. ESOP'05*, LNCS 3444:5–20.
17. 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.
18. K. L. McMillan. Interpolation and sat-based model checking. In *Proc. 15th International Conference on Computer Aided Verification*, LNCS, pages 1–13. Springer, 2003.
19. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.
20. A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, March 2006.
21. H. R. Nielson and F. Nielson. *Semantics with Applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.
22. RTCA. Formal methods supplement to DO-178C and DO-278A. Document RTCA DO-333, RTCA, December 2011.
23. 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.
24. 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.
25. SPARK Pro, 2012. `http://www.adacore.com/home/products/sparkpro/`.

Appendix A: Additional Material for Referees

In this section we present some additional patterns and detail the proofs of some of the properties stated in this article.

A.1: Some Additional Patterns

In the following we shall use the following notations:

$$\begin{array}{rcl} \mathrm{btw}(j,k,i) \equiv j < k < i & \qquad & k \text{ is between } j,i \\ \mathrm{iter}(i) \equiv i - \alpha + 1 & \qquad & \text{number of iterations} \end{array}$$

All the patterns given below are defined on loops of the form $L_r = \ell_{(\alpha,\omega)}\{B_r\}$, i.e., where the loop exit condition is false. For simplicity, we detail only its corresponding body pattern B_r. For all patterns we assume the existence of some preexisting inductive loop invariant $\wp_L$, and initialisation ι_L. Two loop examples and their corresponding generated invariants are given in figure 6.

Knwown Single Update Pattern A scalar variable is modified in a single assignment, whose guard is a $\wp_{L_p}$-value-preserving expression, and it receives a $\wp_{L_p}$-value-preserving expression.

$$\begin{array}{lcl} B_r & \overset{def}{=} & \{g(i) \rightarrow a := e(i)\} \qquad \text{where } \ g(i), e(i) \text{ are } \ \wp_L\text{-vp} \\ \phi(i,a) & = & ((a = a_0) \wedge \forall j.(\alpha \leq j < i \Rightarrow \neg g_0(j))) \\ & & \vee (\exists j.(\alpha \leq j < i \wedge g_0(j) \wedge a = e_0(j)) \wedge \forall k.(\mathrm{btw}(j,k,i) \Rightarrow \neg g_0(k))) \end{array}$$

Known Single Add-Up Pattern A scalar variable a is modified at most in a single assignment, whose guard is a value-preserving expression, and it receives its own previous value plus a constant n.

$$\begin{array}{lcl} B_r & \overset{def}{=} & \{g(i) \rightarrow a := a + n\} \qquad \text{where } \ g(i) \text{ is } \ \wp_L\text{-vp} \\ \phi(i,a) & = & ((a = a_0) \wedge \forall j.(\alpha \leq j < i \Rightarrow \neg g_0(j))) \\ & & \vee\ \exists j. \left(\begin{array}{l} \quad \alpha \leq j < i \wedge\ g_0(j) \\ \wedge\ \exists k.(1 \leq k \leq \mathrm{iter}(j) \wedge a = a_0 + n * k) \\ \wedge\ \forall k.(\mathrm{btw}(j,k,i) \Rightarrow \neg g_0(k)) \end{array} \right) \end{array}$$

Single Min/Max Pattern A scalar variable a is modified in a single assignment, whose guard is a comparison between the variable and a value-preserving expression, and it receives that expression.

$$B_r \overset{def}{=} \{a \bullet e(i) \to a{:=}e(i)\} \quad \text{where } \bullet \in \{<, \leq, >, \geq\} \text{ and } e(i) \text{ is } \wp_L\text{-vp}.$$

$$\phi(i,A) \;=\; \begin{array}{l} ((a = a_0) \wedge \forall j.(\alpha \leq j < i \Rightarrow \neg(a \circ e_0(j)))) \\ \vee \begin{pmatrix} \exists j.(\alpha \leq j < i \wedge a = e_0(j)) \wedge \forall k.(\text{btw}(j,k,i) \Rightarrow \neg(a_0 \bullet e_0(k))) \\ \wedge \forall k.(\alpha \leq k < i \Rightarrow a \circ e_0(k)) \end{pmatrix} \end{array}$$

where $\circ$ can be deduced from $\bullet$ as follows:

$\bullet$	$<$	$\leq$	$>$	$\geq$
$\circ$	$\geq$	$\geq$	$\leq$	$\leq$

Known Multiple Map Pattern An array variable A is modified in more than one assignment, all of whose guards are value-preserving expressions, and it receives in each a value-preserving expression.

$$\begin{array}{lll} B_r & \overset{def}{=} \{ & g_1(i) \to A[i] := e_1(i) \qquad \text{where } g(i) \text{ is } \wp_L\text{-vp} \\ & & \quad ... \\ & & g_n(i) \to A[i] := e_n(i)\} \end{array}$$

$$\begin{array}{rl} \phi(i,A) = & \forall j. \bigwedge_{r\in[1...n]}(\alpha \leq j < i \wedge g_{r0}(j) \Rightarrow A[j] = e_{r0}(j)) \\ & \wedge \forall j.(\alpha \leq j < i \wedge \big((\bigwedge_{r\in[1...n]} \neg g_{r0}(j)) \Rightarrow A[j] = A_0[j]\big) \\ & \wedge \forall j.(\neg(\alpha \leq j < i) \Rightarrow A[j] = A_0[j]) \end{array}$$

Program	Description	Invariant

`for i in 0..n loop`
`  if (A[i] >0)`
`     then c:=c+1;`
`end loop`
	Counting positives	$(c = c_0 \wedge \forall j.(\alpha \leq j < i \Rightarrow \neg(A_0[j] > 0)))$
		$\vee \exists j.((\alpha \leq j < i \wedge A_0[j] > 0)$
		$\wedge \exists k.(1 \leq k \leq j \wedge c = c_0 + k)$
		$\wedge \forall k.(j < k < i \Rightarrow \neg(A_0[k] > 0)))$

`m:= A[1];`
`for i in 0..n loop`
`   if (A[i] < m)`
`     then m:= A[i];`
`end loop;`
	Searching for min	$(m = m_0 \wedge \forall j.(\alpha \leq j < i \Rightarrow \neg(A_0[j] < m_0)))$
		$\vee$
		$(\exists j.(\alpha \leq j < i \wedge m = A_0[j]) \wedge$
		$\forall k.(1 \leq k < i \Rightarrow m \geq A_0[k]))$

Fig. 6. Loop examples and generated invariants

A.2 : Additional Proofs

Given a set of locations, and a set of values, we consider states σ defined in the usual way, that is, as a partial function mapping locations to values.We assume given an operational semantics on $\mathcal{L}$ given by the relation. The semantics of loops in $\mathcal{L}$ is given by the following relation $< p, \sigma > \rightsquigarrow \sigma'$. We rely on the following semantic definition [21] of strongest postcondition sp predicate transformer.

Definition 9 (Predicate $\mathrm{sp}(C, P)$). *For any stament C and predicate P we define the predicate $\mathrm{sp}(C, P)$ as being such that:*

$$\sigma' \vDash \mathrm{sp}(C, P) \Leftrightarrow \exists\sigma.(< C, \sigma > \rightsquigarrow \sigma' \wedge \sigma \vDash P)$$

Proof of lemma 1 (Invariant Definition by sp)

Proof. Let assume ψ is an ι_L-invariant. Using ι_L-invariant definition, condition 1 follows immediately, and the Hoare triple $\{\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i)\}\ B; i := i+1\ \{\psi(i)\}$ must hold. As, $i \notin V_w(B)$, we necessarily have: $\{\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i)\}\ B\ \{\psi(i+1)\}\ i := i+1\ \{\psi(i)\}$, as otherwise, ψ would not be an inductive invariant. Using SP-STRG on the triple $\{\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i)\}\ B\ \{\psi(i+1)\}$ we obtain $\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i)) \Rightarrow \psi(i+1)$ as desired.
Assume now conditions 1 and 2, and let σ_1 be a state such that $\sigma_1 \vDash_{par} \mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i))$. Then, by sp definition: $\exists\sigma_0. < C, \sigma_0 > \rightsquigarrow \sigma_1 \wedge \sigma_0 \vDash \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i)$. On the other hand, by 2, we obtain that $\sigma_1 \vDash_{par} \psi(i+1)$ holds. Using definition on partial correctness satisfaction, we obtain that $\{\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i)\}\ B\ \{\psi(i+1)\}$ holds. Clearly, $\{\psi(i+1)\}\ i := i+1\ \{\psi(i)\}$ holds as well and we obtain that $\{\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i)\}\ B; i := i+1\ \{\psi(i)\}$ holds. This, together with 1, shows that ψ is an ι_L-invariant for L. □

Proof of corollary 1 (Renaming of External Variables in sp)

Proof. By definition, $\mathrm{Psp}(B, V_B \cup V_G)$ results in a formula where (a) all variables occurring in V_B are replaced by x' only on read expressions within B, and (b) all variables $x \in V_G$ occurring in B are replaced by x'. As L is well formed, we know that $V_G \cap V_B = \emptyset$ and therefore, we can separate substitutions performed on V_G's variables from those performed on V_B's variables. Substitutions performed by (a) can be obtained from $\mathrm{Psp}(B, V_B)$. From Psp definition, its easy to see that this formula is equal to $\mathrm{Psp}(B, V_B \cup V_G)$ except for all variables in V_G that are renamed by their primed version. □

Proof of lemma 4 (Filter Pattern Invariant Modularity) In the following we denote $\phi_3(i, v, A)$ as the conjunction $P(i, v, A) \wedge Q(i, v, A) \wedge R(i, v, A) \wedge S(i, v, A)$. We also express all program expressions and logical expressions as functions (or predicates) on (i, v, A). For instance, $\wp_L$ and e are denoted $\wp_L(i, v, A)$ and $e(i, v, A)$. We also use the notation $\wp_L'^{\{v,A\}} = \wp_L(i, v', A')$.

Proof. By definition 4 and 3 we have to prove:

- $V(\phi_3) \subseteq V_w(L_3)$ which follows from $V \overset{def}{=} V(\phi_3) = \{v, A\}$;
- $i = \alpha \wedge \iota_L \Rightarrow \phi_3(i, v, A)$, which follows from $i = \alpha \wedge \iota_L \Rightarrow \phi_3(\alpha, v_0, A_0)$ and that $\iota_L \Rightarrow (v = v_0 \wedge A = A_0)$;

– $\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon(i,v,A) \wedge \wp_L(i,v,A) \wedge \phi_3(i,v,A)) \Rightarrow \phi_3(i+1)$ which we prove below.

Suppose that $\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \wp_L(i,v,A) \wedge \phi_3(i,v,A))$ holds and let us prove that $\phi_3(i+1,v,A)$ holds. By definition of sp this implies that there exists v' and A' such that the following properties hold:

(a) $\alpha \leq i \leq \omega$	(e) $g(i,v',A') \Rightarrow v = v'+1$
(b) $\neg\epsilon(i,v',A')$	(f) $\neg g(i,v',A') \Rightarrow v = v'$
(c) $\wp_L(i,v',A')$	(g) $g(i,v',A') \Rightarrow A[v'] = e(i,v',A')$
(d) $\phi_3(i,v',A')$	(h) $\forall j.(\neg(g(i,v',A') \wedge j = v')) \Rightarrow A[j] = A'[j]$

Notice moreover that the constraints and (c) imply the following:(i) $g(i,v',A') = g(i,v_0,A_0)$, (j) $e(i,v',A') = e(i,v_0,A_0)$ and (k) $\epsilon(i,v',A') = \epsilon(i,v_0,A_0)$. We now prove $\phi_3(i+1,v,A)$ by case on the truth of $g(i,v',A')$ (which is equivalent to $g_0(i,v',A')$ by constraints).

– If $g(i,v',A')$ is false, then by (f) and (h) we can replace v' and A' by v and A in (d) and have $\phi_3(i,v,A)$ and $\phi_3(i+1,v,A)$.
– If $g(i,v',A')$ is true, then by (e) we can replace v' by $v-1$ in (d) and have $\phi_3(i,v-1,A')$. Moreover by (h) we know that A and A' differ only on cell $A[v'] = A[v+1]$ which contains $e(i,v',A')$. Together with $g(i,v',A')$ it is also easy to see that $\phi_3(i+1,v,A)$ also holds. □

Proof of theorem 2 (Loop Invariant Maximality)

Proof. 1. Proving that ϕ is an ι_L-invariant on L: We proceed by showing that ϕ fulfills conditions of lemma 1. Condition (a) is a direct consequence of hypothesis (1); condition (b) follows from (2); condition (c) follows from (3). Therefore, by lemma 1, ϕ is a ι_L-invariant on L.

2. Proving that ϕ is ι_L-maximal: Let ψ be an ι_L-invariant on L, let us prove that $\phi \Rightarrow \psi$ is an ι_L-invariant on L. It suffices to show that $\phi \Rightarrow \psi$ fulfills the three conditions of lemma 1. The first two are easy to show:

– verifying condition (a) on $\phi \Rightarrow \psi$ is equivalent to ask $V(\phi \Rightarrow \psi) \subseteq V(\iota_L)$, which holds since it holds for ϕ by (1) and for ψ as condition (a) is verified on ψ.
– $(i = \alpha \wedge \iota_L) \Rightarrow (i = \alpha \wedge \phi(i))$ holds by (2) and $(i = \alpha \wedge \iota_L) \Rightarrow (\psi(i))$ holds as condition (b) of lemma 1 is verified on ψ. Therefore, condition (b) is verified on $i = \alpha \wedge \iota_L) \Rightarrow (\phi(i) \Rightarrow \psi(i))$ yielding that condition (b) holds on $\phi \Rightarrow \psi$.

Let us prove the last condition, (c) on $\phi \Rightarrow \psi$, i.e. for all i:

$$\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge (\phi(i) \Rightarrow \psi(i))) \Rightarrow (\phi(i+1) \Rightarrow \psi(i+1)).$$

First note that by (3), it suffices to show this property when $\alpha \leq i \leq \omega$, as it holds vacuously on other values of i. Assume $\alpha \leq i \leq \omega$, and $\mathrm{sp}(B, \alpha \leq$

$i \leq \omega \wedge \neg\epsilon \wedge (\phi(i) \Rightarrow \psi(i)))$ and $\phi(i+1)$. Let us prove this entails $\psi(i+1)$. By technical lemma 6, we have $\forall i,\ (\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi(i)) \Rightarrow \psi(i)$ and thus, $\forall i,\ (\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi(i)) \Rightarrow (\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i))$. Therefore by SP-MONO, we have for all i:

$$\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi(i)) \Rightarrow \mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \psi(i))$$

Since $\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi(i))$ holds when $\alpha \leq i \leq \omega$ by (3), and having $\phi(i+1)$ by previous assumption, we obtain $\mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon(i) \wedge \psi(i))$. Finally, since ψ is an ι_L-invariant, we obtain from last result and as we know that condition (c) holds on ψ, the desired result $\psi(i+1)$. □

The following technical lemma states that if the conditions (1), (2) and (3) of theorem 2 hold for an ι_L-invariant, then: $\forall i, (\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi(i)) \Rightarrow \psi(i)$.

Lemma 6 (Invariant maximality technical lemma). *Let ι_L be an initialisation on the loop $L = \ell_{(\alpha,\omega,\epsilon)}\{B\}$, where $i \notin V_w(B)$. Let ϕ be a ι_L-loop invariant such that:*

(1) ι_L covers ϕ.
(2) $\forall i,\ i = \alpha \wedge \iota_L \Leftrightarrow i = \alpha \wedge \phi(i)$
(3) $\forall i,\ \mathrm{sp}(B, \alpha \leq i \leq \omega \wedge \neg\epsilon(i) \wedge \phi(i)) \Leftrightarrow \alpha \leq i \leq \omega \wedge \phi(i+1)$

Then for any ι_L-invariant ψ, $\forall i, (\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi(i)) \Rightarrow \psi(i)$

Proof. We prove $i \leq \omega \wedge \neg\epsilon \wedge \phi(i) \Rightarrow \psi(i)$ by induction on $i \geq \alpha$. Without loss of generality we suppose $\omega \geq \alpha$.
1. Case $i = \alpha$
As condition (b) of lemma 1 holds on ψ, we know that $(\iota_L \wedge i = \alpha) \Rightarrow \psi(i)$ holds. By (2) we have therefore: $(\phi(i) \wedge i = \alpha) \Rightarrow \psi(i)$ which implies for the base case $i = \alpha$ that:

$$(\alpha \leq i \leq \omega \wedge \neg\epsilon \wedge \phi(i)) \Rightarrow \psi(i)$$

2. Inductive step
Assume:

$$i \leq \omega \wedge \neg\epsilon \wedge \phi(i) \Rightarrow \psi(i) \tag{Hi}$$

Let us prove $i+1 \leq \omega \wedge \neg\epsilon \wedge \phi(i+1) \Rightarrow \psi(i+1)$. Assume:

$$i+1 \leq \omega \wedge \neg\epsilon \wedge \phi(i+1) \tag{H1}$$

Let us prove that $\psi(i+1)$ holds. We can rewrite (Hi) as:

$$i \leq \omega \wedge \neg\epsilon \wedge \phi(i) \Rightarrow i \leq \omega \wedge \neg\epsilon \wedge \psi(i)$$

Applying SP-MONO on it we have:

$$\mathrm{sp}(B, i \leq \omega \wedge \neg\epsilon \wedge \phi(i)) \Rightarrow \mathrm{sp}(B, i \leq \omega \wedge \neg\epsilon \wedge \psi(i))$$

By (3) on the left hand side and as condition (c) of lemma 1 holds on ψ, on the right hand side we obtain:

$$\alpha \leq i \leq \omega \wedge \phi(i+1) \Rightarrow \mathrm{sp}(B, i \leq \omega \wedge \neg\epsilon \wedge \psi(i)) \Rightarrow \psi(i+1)$$

Since $\alpha \leq i \leq \omega \wedge \phi(i+1)$ holds by (H1), we obtain the desired result $\psi(i+1)$. □

Proof of lemma 5 (Single Map Invariant Local Maximality) Remember that we ignore array bound considerations. Formally, this means we assume in our formulas that every access to array cells $A[j]$ is done for $j \in [\alpha \ldots \omega]$. That is, arrays have exactly the same bounds as those of the oop index.

Proof. We know that $L_2 = \ell_{(\alpha,\omega)}\{B_2\}$, where $B_2 = \texttt{true} \rightarrow A[i] := e(i)$. Let us define $\iota_2 = \iota_{V(L_{\downarrow L_2})}$ and let $V_2 = V_{nw}(L_{\downarrow L_2})$ (h1). First note, that by definition of Single Map Pattern, the expression $e(i)$ must be initial value preserving in the reduced loop $L_{\downarrow L_2}$. Without loss of generality, we assume $e(i)$ such that any access $A[e_a]$ to array A is such that $e_a \geq i$, otherwise, as $A[i]$ is assigned in this loop, $e(i)$ would not be value preserving. Let us take, $\wp = \iota_{V_2} \wedge \triangle_{A,i}$. We show first that $e(i)$ is $\wp$-vp in the reduced loop. This is clearly the case, by our previous hypothesis on $e(i)$, and because any other location expression x or $B[k]$ occurring in e necessarily corresponds to an external variable x or B, which is not assigned within the reduced loop, and which by (h1) is initialised in ι_{V_2}. Thus, we necessarily have $\iota_{V_2} \Rightarrow x = x_0$ or $\iota_{V_2} \Rightarrow B = B_0$, which yields $e(i)$ is $\wp$-vp. Notice now that the invariant $\varPhi_2(i) = \iota_{V_2} \wedge \phi_2(i)$ is actually equivalent to:

$$\begin{aligned}
\varPhi_2(i) &\equiv \iota_{V_2} \wedge \forall j.(\alpha \leq j < i \Rightarrow A[j] = e_0(j)) \wedge \forall j.(j \geq i) \Rightarrow A[j] = A_0[j] \\
&\equiv \iota_{V_2} \wedge \triangle_{A,i} \wedge \forall j.(\alpha \leq j < i \Rightarrow A[j] = e_0(j)) \\
&\equiv \wp \wedge \forall j.(\alpha \leq j < i \Rightarrow A[j] = e_0(j)) \qquad (1)
\end{aligned}$$

According to definition 8, we must show that $\varPhi_2 = \iota_{V_2} \wedge \phi_2$ is a maximal ι_2-invariant on the reduced loop $L_{\downarrow L_2}$. We proceed by showing that $\varPhi_2$ fulfills the conditions of theorem 2, namely:

1. ι_2 covers $\varPhi_2$, which is immediate as $V(\varPhi_2) = V(\iota_{V_2}) \cup V(\phi_2)$, and because by definition $V(\iota_2) = V(L_{\downarrow L_2})$.
2. $\forall i,\ i = \alpha \wedge \iota_2 \Leftrightarrow i = \alpha \wedge \varPhi_2(i)$
3. $\forall i,\ \mathrm{sp}(B_2, \alpha \leq i \leq \omega \wedge \varPhi_2(i)) \Leftrightarrow \alpha \leq i \leq \omega \wedge \varPhi_2(i+1)$

Let us prove condition (2). As $V(L_{\downarrow L_2}) = V_w(L_{\downarrow L_2}) \cup V_{nw}(L_{\downarrow L_2})$, and because A is the only modified variable in this loop, we know by hypothesis, that $\iota_2 \equiv (A = A_0) \wedge \iota_{V_2}$. When $i = \alpha$, we have $\forall j.(j \geq i \Rightarrow A[j] = A_0[j] \equiv A = A_0$ and that $\forall j.(\alpha \leq j < i \Rightarrow A[j] = e_0(j))$ is vacuosly true. Therefore, $\phi_2(\alpha) \equiv A = A_0$. We obtain:

$$i = \alpha \wedge \varPhi_2(i) \equiv i = \alpha \wedge \iota_{V_2} \wedge (A = A_0) \equiv i = \alpha \wedge \iota_2$$

which achieves the proof of (2). Let us prove condition (3). Let us call:

$$L = \mathrm{sp}(B_2, \alpha \le i \le \omega \wedge \iota_{V_2}(i) \wedge \phi_2(i)) \qquad R = \alpha \le i \le \omega \wedge \iota_{V_2}(i+1) \wedge \phi_2(i+1)$$

We must show $L \Leftrightarrow R$. We develop L by unfolding sp definition, and obtain $L \equiv \exists A'.((a) \wedge (b) \wedge (c) \wedge (d))$ where

$$\begin{array}{ll}
(a)\ \forall j.(\omega \ge j \ge i \Rightarrow A'[j] = A_0[j]) \wedge (A[i] = e'^{\{A\}}(i)) \wedge \quad \iota_{V_2} & \\
(b)\ \forall j.(\alpha \le j < i \Rightarrow (A[j] = A'[j] \wedge A'[j] = e_0(j))) & \equiv \forall j.(\alpha \le j < i \Rightarrow A[j] = e_0(j)) \\
(c)\ \forall j.(\omega \ge j \ge i \Rightarrow (A'[j] = A_0[j] \wedge A[j] = A'[j])) & \\
(d)\ A'[i] = A_0[i] \wedge \alpha \le i \le \omega & \wedge \forall j.(\omega \ge j > i \Rightarrow A[j] = A'[j])
\end{array}$$

By (1) and (a) we can replace $e(i)'^{\{A\}}$ by $e_0(i)$ within (a) to obtain $(a) \equiv \forall j.(\omega \ge j \ge i \Rightarrow A'[j] = A_0[j]) \wedge (A[i] = e_0(i)) \wedge \iota_{V_2}$. Combining this result and (b) we have that:
$A[i] = e_0(i) \wedge \forall j.(\alpha \le j < i \Rightarrow A[j] = e_0(j)) \equiv \forall j.(\alpha \le j < i+1 \Rightarrow A[j] = e_0(j))$.
Combining (c) and (d) we obtain
$(c) \wedge (d) \equiv \exists A'.A'[i] = A_0[i] \wedge \alpha \le i \le \omega \wedge\ \forall j.(\omega \ge j \ge i+1 \Rightarrow A[j] = A_0[j])$. On the other hand, as ι_{V_2} does not contain i, we have $\iota_{V_2}(i) \equiv \iota_{V_2}(i+1)$. Combining these results and unfolding R definition we obtain:

$$\begin{aligned}
L \equiv\ & \alpha \le i \le \omega \wedge \iota_{V_2} \wedge \forall j.(\alpha \le j < i+1 \Rightarrow A[j] = e_0(j)) \\
& \wedge \forall j.(\omega \ge j \ge i+1 \Rightarrow A[j] = A_0[j]) \wedge \exists A'.(A'[i] = A_0[i]) \\
R \equiv\ & \alpha \le i \le \omega \wedge \iota_{V_2} \wedge \forall j.(\alpha \le j < i+1 \Rightarrow A[j] = e_0(j)) \\
& \wedge \forall j.(\omega \ge j \ge i+1 \Rightarrow A[j] = A_0[j])
\end{aligned}$$

By hypothesis, $A_0[i]$ is defined as long as $\alpha \le i \le \omega$ holds. Therefore $\exists A'.(A'[i] = A_0[i])$ is true, which ends the proof. $\square$