A Precise Pictorial Language for Array Invariants

Johannes Eriksson, Masoumeh Parsa(✉), and Ralph-Johan Back

Department of Information Technologies, Åbo Akademi University, Vesilinnantie 3, 20500 Turku, Finland
Johannes.Eriksson@fourferries.fi, {mparsa,backrj}@abo.fi

Abstract. Pictorial languages, while intuitive and descriptive, are rarely used as the primary reasoning language in program verification due to lack of precision. In this paper, we introduce a precise pictorial language for specifying array invariants that preserves visual perspicuity. The language extends Reynold's *partition diagrams* with the notion of a *coloring*, allowing assertions over portions of an array to be expressed by color-coding. The semantics of a coloring is given by a *legend*, mapping a colored partition of an array into a universally quantified predicate over the array. The pictorial syntax is an extension to*invariant diagrams*, transition graphs where preconditions, postconditions and invariants, rather than the program code, determine the main program structure. We demonstrate the approach with three examples, verified using the Why3 theorem prover frontend.

1 Introduction

Deductive program verification is the process of establishing correctness by proving verification conditions (VCs) extracted from a program. It relies on a formal pre- and postcondition specification as well as loop invariants being provided by the programmer. This task by itself requires proficiency in mathematical logic. Further challenges include assessing completeness of the specification, whether invariants are sufficiently strong to establish the postcondition while sufficiently weak to be maintained, and using automatic theorem provers. Training in formal methods aims at giving the necessary conceptual and technical skills to address these challenges.

In instructional settings, verification is often taught by examples from tangible and visually perspicuous domains, such as arrays of colored objects in the case of the famous Dutch national flag three-way-partitioning problem [1]. While such examples are a valuable pedagogical device, how to generalize the reasoning to more typical programming problems is often left unexplained. Also, the transition from instructional pictures to a mathematically precise *reasoning language* does not always follow a happy path. Even though influential authors have already several decades ago highlighted the benefits of pictures in formal reasoning [2,3], pictures have by and large been employed as stepping stones towards

C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 151–160, 2018.
https://doi.org/10.1007/978-3-319-98938-9_9

some final, textual, formalization suitable for conventional symbolic reasoning. While reasons therefore (lack of precision, technical limitations, convention) may be legitimate, we suspect that this demoted role of pictures means their full benefit as reasoning tools is not being realized.

A notable exception is Reynold's *interval* and *partition diagrams* [4], which integrate pictures with mathematical notation seamlessly, allowing invariants and even proofs over arrays to be expressed in a way that simultaneously maintains visual perspicuity and mathematical precision. It is on this trajectory that we position the approach described in this paper. A partition diagram, in its base form, is a *precise*, *compact* and *embeddable* diagram stating that a collection of integer indexes subdivides an array into disjoint partitions. Precise means that the language has a well-defined mathematical meaning, compact that it is space-conserving, and embeddable that it can be integrated into another diagram or a textual formula. Associated with a partitioning is some collection of properties, that the elements in the partitions should satisfy. A property can be expressed precisely by a formula universally quantifying over the partition diagram (e.g., [4, p. 94]), or by (less formally) annotating the partition diagrams with the properties (e.g., [3, p. 94]). In line with the second approach, we extend partition diagrams with the notion of *coloring* a partition. Formally, a coloring is a function from array indexes to a small finite set ("palette") of programmer-defined colors. The programmer gives interpretation to the colors through the *legend* construct. Analogously to its cartography namesake, a legend is a mapping from colors to a universally-quantified predicate over the colored partitions. Together, partitionings, colorings and legends provide a precise and expressive pictorial language for array invariants.

As an umbrella framework we use *invariant-based programming* (IBP), a correct-by-construction formal verification approach geared towards teaching [5]. In IBP, preconditions, postconditions and invariants—under the common nomen *situations*—serve as the main organizing structures of a program. The program is represented by an *invariant diagram*, a graph of nested situations connected by *transitions*. The situations represent state predicates, such as pre- and postconditions and invariants, while the transitions constitute the actual executable code. We define the semantics of colorings and legends by translation into predicates over the program state. After translation, the VCs of the diagram are extracted using the proof rules of invariant diagrams. Nesting allows substitutions to inherit constraints from outer situations. In our extension, nesting also allows legends to be shared by multiple situations, as well as to be extended in substitutions with additional color interpretations. We illustrate the approach with examples from the domain of searching and sorting. The examples have been mechanically verified using the Why3 platform [6], a front-end for a number of automatic theorem provers.

We proceed as follows. Section 2 introduces the pictorial language in the context of two search programs. Section 3 describes the verification semantics. A verification of a slightly more complex program is given in Sect. 4. We discuss related work in Sect. 5 and end the paper with conclusions and future work in Sect. 6.

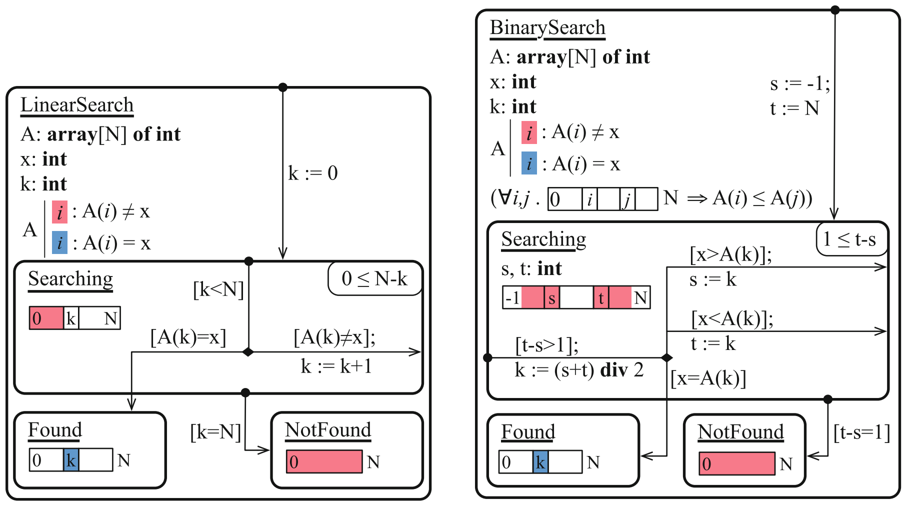


Fig. 1. Linear and binary search

2 Pictorial Invariant Diagrams

Consider the leftmost invariant diagram in Fig. 1. Each rounded rectangle—called a *situation*—identifies a subset of all possible program states. The role of a situation in a program is determined by the *transitions*, guarded program statements, connecting to it: a situation with no incoming transitions corresponds to a precondition (LinearSearch); a situation with no outgoing transitions corresponds to a postcondition (Found and NotFound). A situation with both incoming and outgoing transitions is an *intermediate situation*; an intermediate situation (or collection of intermediate situations) connected through a cycle of transitions corresponds to a *loop* (Searching). There are five types of declarations that can appear inside a situation:

Variable declarations introduce program variables and associate them with types. For example, the declaration "A :**array**[N] **of int**" in the situation LinearSearch types the variable A as an integer array of length N, indexed from 0 to N − 1.

Legends introduce colors and assign them their meanings. For instance, the legend "A| i : A(i) $\neq$ x" states that the red elements in A are different from x. A legend is not a state assertion; rather it introduces an implication, allowing invariants over an array to be expressed visually by "painting" sub-arrays with a relevant property (in this case, that the sub-array is known to not hold the value x). We make this notion more precise in the next section. Legends may introduce any number of new colors, but the color palettes for distinct arrays must be disjoint.

Invariants are assertions over the program variables of the situation. We can express invariants using standard mathematical and logical notation. E.g., “$0 \leq \mathrm{k} \leq \mathrm{N}$” expresses that the value of k is between 0 and N, inclusively. For asserting that a collection of variables form a partitioning we prefer to use Reynolds-style partition diagrams. The basic partition diagram is a rectangular contour:

$$i\boxed{\quad\ j}$$

where i and j are integer expressions over the program variables. It stands for the predicate “$i < j$”. The bounds may be juxtaposed with respect to the adjacent edge to specify whether they are inclusive or exclusive:

$$\begin{aligned} i\boxed{\quad\ }j &= i\boxed{\quad\ j-1} &&= i<j-1\\ \boxed{i\quad\ }j &= i-1\boxed{\quad\ j-1} &&= i-1<j-1\\ \boxed{i\quad j} &= i-1\boxed{\quad\ j} &&= i-1<j \end{aligned}$$

Conjunctions of partition diagrams, when the upper bound of the predecessor coincides with the lower bound of its successor, may be written in chained form:

$$\begin{aligned} i\boxed{\quad j}\!\boxed{\quad k} &= i\boxed{\quad j} \wedge j\boxed{\quad k}\\ i\boxed{\quad}\!\boxed{j}\!\boxed{\quad k} &= i\boxed{\quad j} \wedge \boxed{j\quad k} \end{aligned}$$

The following abbreviations denote singleton intervals:

$$\begin{aligned} \boxed{i} &= \boxed{i\quad i}\\ i\boxed{\,} &= \boxed{i+1\quad i+1}\\ \boxed{\,}i &= \boxed{i-1\quad i-1} \end{aligned}$$

Using partition diagrams, the aforementioned predicate is equivalently expressed as:

$$\boxed{0\quad\ }\!\boxed{\mathrm{k}}\!\boxed{\quad\ \mathrm{N}} = 0 \leq \mathrm{k} \leq \mathrm{N}$$

As mentioned, partition diagrams can be embedded in textual formulas; e.g., the invariant of BinarySearch states that A is sorted.

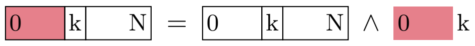
Colorings are pictorial invariants similar to partition diagrams, but appear as colored regions rather than as contours. The basic form is

$$i\colorbox{lightgray}{$\quad /c/ \quad j$}$$

where c is the chosen color of the regions (for contrast, we chiefly pick 🟥, 🟦 and 🟩). It stands for the partial definition of a *coloring function* over the integer interval $(i, j]$. Colorings allow the same syntactic shorthands as partition diagrams (bound juxtaposition, chaining and singleton intervals). For example, the following coloring asserts that the coloring function takes the value 🟥 between 0 (inclusive) and k (exclusive):

$$\colorbox{pink}{$0\quad\ $}\mathrm{k}$$

For compact representation, partitioning and coloring invariants may be drawn overlapping when their bounds coincide. For example, the invariant of situation BinarySearch ▷ Searching is the conjunction of a partitioning and a coloring:



Variants are written in the upper right corner of intermediate situations that are part of a loop. To verify termination, we need to show that the variant $(\mathrm{N} - k)$ is decreased by each transition through the situation and does not decrease below the lower bound (0).

Finally, we note that situations can be nested. Nesting is *conjunctive*: an inner situation inherits all declarations, with the exception of variants, from the enclosing situations.

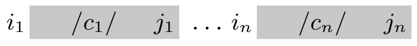
3 Verification of Pictorial Invariant Diagrams

An invariant diagram is *correct* iff it is *consistent*, *terminating* and *live*. A transition t from a situation satisfying predicate p to situation satisfying predicate q is consistent if $p \Rightarrow \mathsf{wp}(t, q)$ is true, where wp is the weakest precondition transformer. For termination, we check that the variant v decreases and that its lower bound is maintained on re-entry to the situation, i.e., $v = v_0 \wedge p \Rightarrow \mathsf{wp}(t, 0 \le v < v_0)$. A situation satisfying p is live if at least one outgoing transition is always enabled, i.e, $p \Rightarrow \mathsf{wp}(t, g_1 \vee \cdots \vee g_n)$, where $g_1, \ldots, g_n$ are the guards of outgoing transitions. Next, we describe how the pictorial elements of a situation (legends and colorings) combine into a predicate onto which these rules can be applied. For a formal treatment of the proof rules themselves, see [7].

For a given situation s, let $\overline{x}$ be the declared variables, $\overline{T}$ their types, and $\overline{a}$ the subset of $\overline{x}$ containing only the variables of array type. The *coloring function* associated with a variable $A \in \overline{a}$ of type **array**[N] in situation s is a total function from the program state and an array index

$$\mathsf{col}_{s,A} : \overline{T} \times [0, \mathrm{N}) \rightarrow C_{s,A} \cup \{⧄\}$$

where the set $C_{s,A}$ is the color palette associated with A in s, and ⧄ is a special value indicating that no coloring has been specified. The coloring function formalizes the mapping between legends and invariants, is fully defined, and is intended to be fully eliminated from the final VC. Given the colorings declared for array A in situation s:



where $c_1, \ldots, c_n \in C_{s,A}$, the coloring function is defined as:

$$\begin{aligned}\mathsf{col}_{s,A}(\overline{x})(i) = {} & \mathsf{if}\ (i_1 < i \le j_1)\ \mathsf{then}\ c_1 \\ & \vdots \\ & \mathsf{else\ if}\ (i_n < i \le j_n)\ \mathsf{then}\ c_n \\ & \mathsf{else}\ ⧄\end{aligned}$$

Disjointness of partitioning means that the if-conditions are mutually exclusive, and the else-clause ensures that the function is total. A legend declaration for variable a in situation s has the general form:

$$A \mid i_1 \quad /c_1/ \quad j_1 \; \ldots \; i_n \quad /c_n/ \quad j_n \quad : \; p(\overline{x}, i_1, \ldots, i_n, j_1, \ldots, j_n)$$

where $c_1, \ldots, c_n \in C_{s,a}$ and p is a predicate on the program state. Semantically, this legend stands for the following predicate:

$$\begin{aligned}\mathsf{lgd}_{s,A}(\overline{x}) = (\forall i_1, \ldots, i_n, j_1, \ldots, j_n \; . \; & (0 \le i_1 < j_1 < \mathrm{N}) \wedge \cdots \wedge (0 \le i_n < j_n < \mathrm{N}) \\ & \wedge \; (\forall k \, . \, i_1 < k \le j_1 \Rightarrow \mathsf{col}_{s,A}(\overline{x})(k) = c_1) \\ & \vdots \\ & \wedge \; (\forall k \, . \, i_n < k \le j_n \Rightarrow \mathsf{col}_{s,A}(\overline{x})(k) = c_n) \\ & \Rightarrow p(\overline{x}, i_1, \ldots, i_n, j_1, \ldots, j_n))\end{aligned}$$

That is, a legend is an assertion that p holds for subintervals of A matching the sequence of colorings given in the legend. Like invariants, legends are conjunctive.

To verify a diagram, we generate a theory including the coloring functions, legend predicates and invariants of each situation, and a lemma to be proved for each transition. For example, the theory of BinarySearch ▷ Searching in Fig. 1 contains the following declarations (for brevity, in $\mathsf{lgd}_{\underline{\text{Searching}},A}$ we have omitted the inner quantifications, as the ranges are singletons in both cases):

$$\begin{aligned}\mathsf{col}_{\underline{\text{Searching}},A}(A,s,t)(i) &= \mathsf{if}\ (-1 \le i \le s)\ \mathsf{then}\ \text{🟥} \\ &\quad\ \mathsf{else\ if}\ (t \le i < N)\ \mathsf{then}\ \text{🟥} \\ &\quad\ \mathsf{else}\ \text{▨} \\ \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s,t) &= \quad (\forall i \, . \, (0 \le i < N) \wedge (\mathsf{col}_{\underline{\text{Searching}},A}(i) = \text{🟥}) \Rightarrow A(i) \neq x) \\ &\quad\ \wedge (\forall i \, . \, (0 \le i < N) \wedge (\mathsf{col}_{\underline{\text{Searching}},A}(i) = \text{🟦}) \Rightarrow A(i) = x) \\ \mathsf{inv}_{\underline{\text{Searching}},A}(A,s,t) &= \quad -1 \le s < t \le N \\ &\quad\ \wedge (\forall i,j . 0 \le i \le j \le N \Rightarrow A(i) \le A(j))\end{aligned}$$

To generate the VCs for situation Searching, we can now apply the proof rules of IBP, taking the conjunction of $\mathsf{lgd}_{\underline{\text{Searching}},A}$ and $\mathsf{inv}_{\underline{\text{Searching}},A}$ as the situation predicate. For example, to prove that the loop transition "$[\mathrm{t} - \mathrm{s} > 1]$; $\mathrm{k} := (\mathrm{s} + \mathrm{t})\ \mathsf{div}\ 2$; $[\mathrm{x} > \mathrm{A(k)}]$; $\mathrm{s} := \mathrm{k}$" is consistent we will need to discharge the following VC:

$$\begin{aligned}(\forall A, s, t \; . \; & \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s,t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A,s,t) \\ & \wedge \; t - s > 1 \; \wedge \; k = (s + t)\ \mathsf{div}\ 2 \; \wedge \; x > A(k) \; \wedge \; s' = k \\ & \Rightarrow \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s',t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A,s',t))\end{aligned}$$

Additionally, to prove that the same transition is decreasing the variant of Searching:

$$\begin{aligned}(\forall A, s, t \; . \; & \mathsf{lgd}_{\underline{\text{Searching}},A}(A,s,t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A,s,t) \\ & \wedge \; t - s > 1 \; \wedge \; k = (s + t)\ \mathsf{div}\ 2 \; \wedge \; x > A(k) \; \wedge \; s' = k \\ & \Rightarrow 1 \le t - s' < t - s)\end{aligned}$$

Note that the antecedents are identical to those of the consistency VC. Finally, the liveness condition for situation Searching is:

$$\begin{aligned}(\forall A, s, t \quad . \quad & \mathsf{lgd}_{\underline{\text{Searching}},A}(A, s, t) \wedge \mathsf{inv}_{\underline{\text{Searching}},A}(A, s, t) \\ \wedge\ & k = (s + t) \ \mathsf{div}\ 2 \\ \Rightarrow\ & (t - s = 1) \vee (t - s > 1 \wedge (x > A(k) \vee x < A(k) \vee x = A(k))))\end{aligned}$$

The VCs can now be discharged using an automatic theorem prover.

4 Example: Insertion Sort

Figure 2 shows an invariant diagram interpretation of insertion sort. It consists of an outer loop (Sorting) maintaining a sorted partition (green), and an inner loop (Inserting) moving the next element from the unsorted partition into its correct position in the sorted partition. The inner loop, as it moves the element back one step per iteration, maintains two sorted partitions (green and blue). The control flow transfers from the inner to the outer loop when the concatenation of the partitions becomes sorted. The outer loop terminates when every element of the array has been processed. Transitions must additionally ensure that A is a permutation of the original A_0.

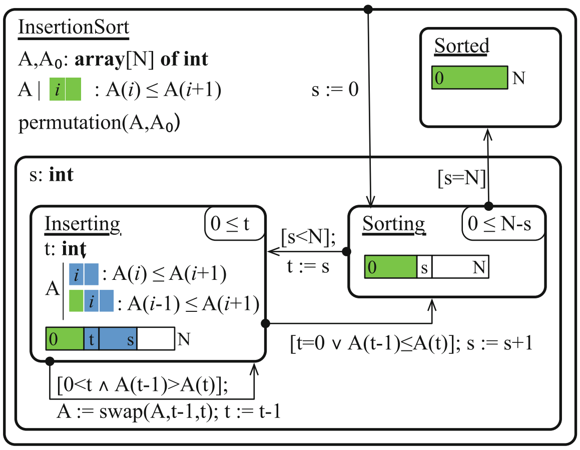


Fig. 2. Insertion sort (Color figure online)

Like invariants, legends are inherited from outer situations. For instance, that any two adjacent green elements are sorted is visible to both Sorting and Inserting. We note that legends may introduce new colors limited in scope to the declaring situation and its nested situations. For example, blue introduced by the legend of Inserting is visible only within Inserting. The coloring functions, legend

$$
\begin{aligned}
\mathsf{col}_{\underline{\text{Sorting}},A}(A,A_0,s)(i) &= \text{if } (0 \le i < s) \text{ then } \text{🟩} \text{ else } \text{⧄} \\
\mathsf{lgd}_{\underline{\text{Sorting}},A}(A,A_0,s) &= (\forall i \;.\; 0 \le i < N \wedge 0 \le i+1 < N \\
&\qquad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = \text{🟩} \\
&\qquad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = \text{🟩} \\
&\qquad \Rightarrow A(i) \le A(i+1)) \\
\mathsf{inv}_{\underline{\text{Sorting}},A}(A,s) &= \mathsf{permutation}(A,A_0) \wedge 0 \le s \le N \\
\mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) &= \text{if } (0 \le i < t) \text{ then } \text{🟩} \\
&\quad\ \text{else if } (t \le i \le s) \text{ then } \text{🟦} \\
&\quad\ \text{else } \text{⧄} \\
\mathsf{lgd}_{\underline{\text{Inserting}},A}(A,A_0,s,t) &= \quad (\forall i \;.\; 0 \le i < N \wedge 0 \le i+1 < N \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = \text{🟩} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = \text{🟩} \\
&\qquad\quad \Rightarrow A(i) \le A(i+1)) \\
&\quad \wedge\ (\forall i \;.\; 0 \le i < N \wedge 0 \le i+1 < N \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = \text{🟦} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = \text{🟦} \\
&\qquad\quad \Rightarrow A(i) \le A(i+1)) \\
&\quad \wedge\ (\forall i \;.\; 0 \le i-1 < N \wedge 0 \le i+1 < N \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i-1) = \text{🟩} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i) = \text{🟦} \\
&\qquad\quad \wedge\ \mathsf{col}_{\underline{\text{Inserting}},A}(A,A_0,s,t)(i+1) = \text{🟦} \\
&\qquad\quad \Rightarrow A(i) \le A(i+1)) \\
\mathsf{inv}_{\underline{\text{Inserting}},A}(A,s,t) &= \quad \mathsf{permutation}(A,A_0) \\
&\quad \wedge\ 0 \le t \le s < N
\end{aligned}
$$

Fig. 3. Coloring function, invariant and legend predicate of situations Sorting and Inserting. (Color figure online)

predicates and invariant predicates for situations Sorting and Inserting are shown in Fig. 3. Given these functions and predicates, the VCs for the transitions are formulated as described in the previous section (omitted here for brevity). The VCs are automatically proved by Why3 and its associated SMT solvers Z3 [8] and CVC4 [9].

5 Related Work

Reynolds [4] introduced interval and partition diagrams to express constraints on arrays. Gries's seminal textbook [3] uses array pictures in several examples. Astrachan [2] suggests pictorial representations of arrays and linked lists. Ginat [10] considers loop invariants as mathematical games, with emphasis on the heuristics of invariant identification. Some recent approaches have explored transforming invariant problems into games [11,12] and crowdsourcing verification to online communities. Partitioning has been employed in static analysis and heuristics-driven loop invariant generation [13,14]. Reasoning on range predicates is the basis of the axiomatic rules on array manipulations for correctness proofs of programs involving arrays in [15]. The converse problem, generating visual representations from textual specifications, has been addressed in the context of the Z language [16], and also with the purpose of visualizing VCs on arrays [17]. While pictures and colors are a staple in algorithm animation, we are not aware of prior work combining partitionings and colorings for formal reasoning.

6 Conclusions and Future Work

In this paper, we have introduced a pictorial language for invariants over arrays. The language extends two existing visual formalisms: the notation for invariants and predicates builds on Reynold's partition diagrams, extending them with colorings to connect partitions with desired properties; the language for specifying the invariant structure and program statements is invariant diagrams, extended with a hierarchical mapping of colorings to predicates. Partition diagrams, colorings and legends seem to be rather expressive visual constructs, allowing many common array invariants to be stated.

This work is in its initial phases with multiple directions to be explored. First and foremost, tool support (in the form of editors and VC generators) would be needed for practical use. Existing tools for IBP [18] do not support the array-specific visual notations introduced here. Secondly, we would like to generalize the approach to more advanced data structures, such as trees and graphs. One challenge here is finding equally expressive and intuitive visual partitioning notations to state invariants over these non-linear data structures. Thirdly, we believe that colorings could serve runtime visualization and animation by overlaying the colors on a data structure instance picture, and analogously, to produce color-coded counterexamples during verification.

References

1. Dijkstra, E.W.: A Discipline of Programming, 1st edn. Prentice Hall PTR, Upper Saddle River (1997)
2. Astrachan, O.L.: Pictures as invariants. In: Dale, N.B. (ed.) Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education 1991, San Antonio, Texas, USA, 7–8 March 1991, pp. 112–118. ACM (1991)
3. Gries, D.: The Science of Programming, 1st edn. Springer, Secaucus (1987). https://doi.org/10.1007/978-1-4612-5983-1
4. Reynolds, J.C.: The Craft of Programming. Prentice Hall PTR, Upper Saddle River (1981)
5. Back, R.J.: Invariant based programming: basic approach and teaching experiences. Form. Asp. Comput. **21**(3), 227–244 (2009)
6. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
7. Back, R.J., Preoteasa, V.: Semantics and proof rules of invariant based programs. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC 2011, pp. 1658–1665. ACM, New York (2011)

8. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
10. Ginat, D.: Loop invariants and mathematical games. SIGCSE Bull. **27**(1), 263–267 (1995)
11. Fava, D., Shapiro, D., Osborn, J., Schäef, M., Whitehead Jr., E.J.: Crowdsourcing program preconditions via a classification game. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, pp. 1086–1096. ACM, New York (2016)
12. Logas, H., Vallejos, R., Osborn, J., Compton, K., Whitehead, J.: Visualizing loops and data structures in Xylem: the code of plants. In: Proceedings of the Fourth International Workshop on Games and Software Engineering, GAS 2015, pp. 50–56. IEEE Press, Piscataway (2015)
13. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 338–350. ACM, New York (2005)
14. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 105–118. ACM, New York (2011)
15. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_23
16. Moremedi, K., van der Poll, J.A.: Transforming formal specification constructs into diagrammatic notations. In: Cuzzocrea, A., Maabout, S. (eds.) MEDI 2013. LNCS, vol. 8216, pp. 212–224. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41366-7_18
17. Jami, M., Ireland, A.: A verification condition visualizer. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 72–86. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_5
18. Eriksson, J.: Tool-supported invariant-based programming. Ph.D. thesis, Turku Centre for Computer Science, Finland (2010). http://urn.fi/URN:ISBN:978-952-12-2446-1