Abstraction of Arrays Based on Non Contiguous Partitions *

Jiangchao Liu and Xavier Rival

INRIA, ENS, CNRS, Paris, France,
jliu@di.ens.fr, rival@di.ens.fr

Abstract. Array partitioning analyses split arrays into contiguous partitions to infer properties of cell sets. Such analyses cannot group together non contiguous cells, even when they have similar properties. In this paper, we propose an abstract domain which utilizes semantic properties to split array cells into groups. Cells with similar properties will be packed into groups and abstracted together. Additionally, groups are not necessarily contiguous. This abstract domain allows to infer complex array invariants in a fully automatic way. Experiments on examples from the Minix 1.1 memory management demonstrate its effectiveness.

1 Introduction

Arrays are ubiquitous, yet their mis-use often causes software defects. Therefore, a large number of works address the automatic verification of array manipulating programs. In particular, partitioning abstractions [5,11,13] split arrays in sets of contiguous groups of cells, in order to, hopefully, infer they enjoy similar properties. A traditional example is that of an initialization loop, with the usual invariant that splits the array in an initialized zone and an uninitialized region.

However, when cells that have similar properties are not contiguous, these approaches cannot infer adequate array partitions. This happens for unsorted arrays of structures, when there is no relation between indexes and cell fields. Then, there are usually relations among cell fields. This phenomenon can be observed in low-level software, such as operating system services and critical embedded systems drivers, which rely on static array zones instead of dynamically allocated blocks [20]. When cells with similar properties are not contiguous, traditional partition based techniques are unlikely to infer relevant partitions / precise array invariants. Figure 1 illustrates the Minix 1.1 Memory Management Process Table (MMPT) main structure. The array of structures `mproc` defined in Figure 1(a) stores the process descriptors. Each descriptor comprises a field `mparent` that stores the index of the parent process in `mproc`, and a field `mpflag` that stores the process status. Figure 1(c) depicts the concrete values stored in `mproc` to describe the process topology shown in Figure 1(b) (we show only 8 processes). An element

* The research leading to these results has received funding from the European Research Council under the FP7 grant agreement 278673, Project MemCAD, and from the ARTEMIS Joint Undertaking no 269335 (see Article II.9 of the JU Grant Agreement).

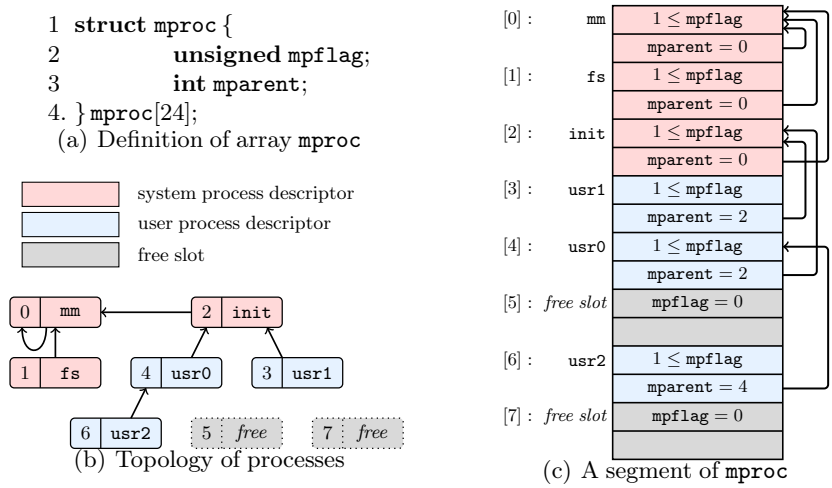


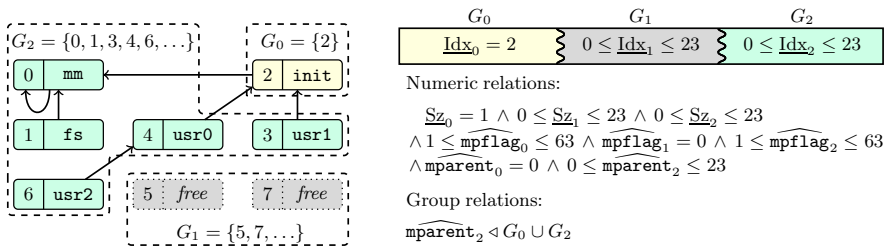
Fig. 1. Minix 1.1 Memory Management Process Table (MMPT) structure

of `mproc` is a process descriptor when its field `mpflag` is strictly positive and a free slot if it is null. Minix 1.1 uses the three initial elements of `mproc` to store the descriptor of the memory management service, the file system service and the init process. Descriptors of other processes appear in a random order. In the example of Figure 1, `init` has two children whose descriptors are in `mproc`[3] and `mproc`[4]; similarly, the process corresponding to `mproc`[4] has a single child the descriptor of which is in `mproc`[6]. Moreover, Minix assumes a parent-child relation between `mm` and `fs`, as `mm` has index 0 and the parent field of `fs` stores 0. To abstract the process table state, valid process descriptors and free slots should be partitioned into *different groups*.

Traditional, contiguous partitioning cannot achieve this for two reasons: (1) the order of process descriptors in `mproc` cannot be predicted, hence is random in practice, and (2) there is no simple description of the boundaries between these regions (or even their sizes) in the program state. The symbolic abstract domain by Dillig, Dillig and Aiken [8] also fails here as it cannot attach arbitrary abstract properties to summarized cells.

In this paper, we set up an abstract domain to partition the array into non contiguous groups for process descriptors and free slot so as to infer this partitioning and precise invariants (Section 2) automatically. Our contributions are:

1. An abstract domain that partitions array elements according to semantic properties, and can represent non contiguous partitions (Section 3).
2. Static analysis algorithms for the computation of abstract post-conditions (Sections 4 and Section 5), widening and inclusion check (Section 6).
3. The implementation and the evaluation of the analysis on the inference of tricky invariants in an excerpt of the Minix 1.1 Memory Management Process Table (MMPT) and other challenging array examples (Section 7).



(a) A concrete state of `mproc` (b) Abstract state with partition

Fig. 2. A partitioning of `mproc` based on non contiguous groups

2 Overview

The Minix MMPT requires `mproc` to permanently satisfy two invariants:

1. Each valid process descriptor has a `mparent` field, that should store a value in $[0, 23]$, hence represents a valid index in `mproc`: this entails the absence of out-of-bound accesses in process table management functions.
2. The `mparent` field of any valid process descriptor should be the index of a valid process descriptor: as a process can only complete its exit phase when its parent calls wait, failure to maintain a parent for each process could cause a terminating process to become dangling and never be eliminated.

To verify these invariants, we propose to check that all system calls preserve them. We design an automatic analysis to verify that, if they are called in a state that satisfies these invariants, they return in a state that also satisfies them. A concrete state is displayed in Figure 2(a), and its abstraction is shown in Figure 2(b). Group 0 contains only the process descriptor of `init`. Group 1 collects all free slots. Group 2 consists of all the valid process descriptors except that of `init`. The reason why we split `init` out into a separate group is that it is often treated in a special manner by OS routines. We let G_i denote the set of indexes of all the elements in group i.

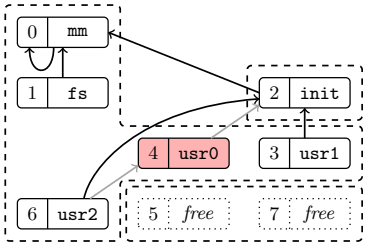
The abstract state shown in Figure 2(b) ties each group to properties of its elements. These will be formally defined in Section 3. By the Minix specification, the elements of group 2 satisfy the following correctness conditions $\mathcal{C}$:

- their indexes are in $[0, 23]$, which we note $0 \leq \underline{\text{Idx}}_2 \leq 23$ in Figure 2(b);
- their flags are in $[1, 63]$ (valid process descriptors have a strictly positive flag), which we note $1 \leq \widehat{\texttt{mpflag}}_2 \leq 63$;
- their parents are valid indexes, which we note $0 \leq \widehat{\texttt{mparent}}_2 \leq 23$;
- their parents are indexes of valid process descriptors, hence are also in group 0 or group 2, which we note $\widehat{\texttt{mparent}}_2 \triangleleft G_0 \cup G_2$;
- the size of group 2 is between 0 and 23, which we note $0 \leq \underline{\text{Sz}}_2 \leq 23$.

Our abstraction relies on *disjoint* groups as other array partitioning abstractions [11,13]. However, our abstraction *does not* assume each group consists of a contiguous set of cells. The *non-contiguousness* of groups is represented by winding separation lines in Figure 2(b). To characterize groups, our abstraction relies

```
void cleanup(int child){
  ...
⓪ int parent = mproc[child].mparent;
① if(parent == 2){
②   mproc[parent].mpflag = 1;
③   mproc[child].mpflag = 0;
④   for(i = 0; i < 24; i++){
⑤     if(mproc[i].mpflag > 0)
        if(i! = parent)
          if(mproc[i].mparent == child)
            mproc[i].mparent = 2;
    }
⑥ }else{...}...}
```

(a) A simplified excerpt of `cleanup`



(b) Effect of `cleanup`

Fig. 3. Minix 1.1 process table management, system function `cleanup`

not only on constraints on indexes, but also on semantic properties of the cell contents: while groups 1 and 2 correspond to a similar range, the `mpflag` values of their elements are different (0 in group 1 and any value in $[1, 63]$ in group 2). Therefore our abstraction can express both contiguous and non contiguous partitions. In this example, we believe the abstract state of Figure 2(b) is close to the programmer's intent, where the array is a collection of unsorted elements.

We now consider the verification of Minix MMPT management procedures. We focus on `cleanup`, which turns elements of `mproc` that describe hanging processes into free slots. Figure 3(a) displays an excerpt of a simplified, recursion free version of `cleanup`, which is chosen to highlight the analysis difficulties. The call `cleanup`(4) in the state of Figure 2(a) will remove process `usr0` and falls in that case; the result is shown in Figure 3(b): process `usr2` becomes a child of `init`, while the record formerly associated to `usr0` turns into a free slot.

Function `cleanup` should be called in a correct Minix process table state and be applied to a child process in group 2, which we note $\texttt{child} \triangleleft G_2$. Figure 4 overviews the steps of the automatic static analysis of the excerpt of `cleanup`. The analysis proceeds by computing abstract post-conditions and loop invariants [3]. In this section, we focus on (1) cell materialization, (2) termination of the loop analysis and (3) removal of unnecessary groups.

From the precondition, fields `mparent` of all elements in group 2 are indexes in groups G_0 or G_2 (abstract state at point ⓪). The test entails `mparent` is 2 at point ② (corresponding to process `init`). Combining this, with the fact that group 0 has exactly one element ($\underline{\text{Sz}}_0 = 1$) at index 2 ($\underline{\text{Idx}}_0 = 2$), the analysis infers that `parent` can only be in group 0 (point ②). Therefore, the update at point ② affects a group with a single element, hence, is a *strong update*, and produces predicate at point ③. However, at that point, the next update is not strong, since `mproc`[`child`] may be any element of group 2, which may have more than one element (it has at least one element since $\texttt{child} \triangleleft G_2$, thus $\underline{\text{Sz}}_2 \geq 1$). Therefore, our domain *materializes* the array element being assigned by splitting group 2 into two groups, labeled 2 and 3. Both groups inherit predicates from former group 2. Additionally,

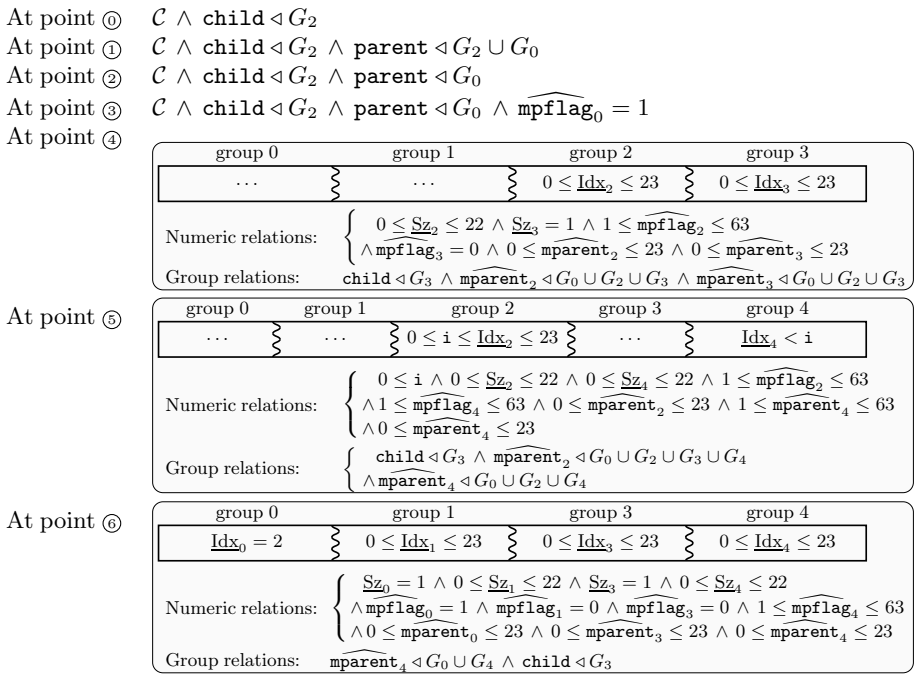


Fig. 4. Overview of the analysis of `cleanup`

group 3 has a single element ($\underline{\text{Sz}}_3 = 1$), thus the analysis performs a strong update and generates the abstract state of ④.

The analysis of all the statements in the program follows similar principles. We only discuss the termination of the analysis here, as our abstract domain has infinite chains (the number of groups is not bounded), hence the analysis of loops requires a terminating binary widening operator [3]. Widening associates groups of its inputs with groups of its result (ensuring the number of groups can only decrease to guarantee termination), and over-approximates group properties. After two widening iterations, our analysis produces abstract post-fixpoint ⑤, where group 1 describes free slots, group 0 describes `init`, group 3 consists of `child` (just cleaned up) and groups 2 (resp., 4) represent valid process descriptors with indexes greater (resp., lower) than `i`. Our analysis can also decrease the number of groups, when some become redundant, e.g., when the analysis proves a group empty. For instance, the loop fixpoint ⑤ shows that indexes of elements in group 2 are greater than `i`. Thus, after the loop exit, any element of group 2 should have an index greater than 24, which implies this group is empty. Hence, this group is removed, and the analysis produces post-condition ⑥, which entails correctness condition $\mathcal{C}$ (note that group 3, corresponding to `child` now describes a free slot).

3 Abstract domain and abstraction relation

In this section, we formalize abstract elements and their concretization. We describe the abstraction of the contents of arrays, using numeric constraints, in Section 3.1. Then, we extend it with relations between groups in Section 3.2.

3.1 The non-contiguous array partition domain

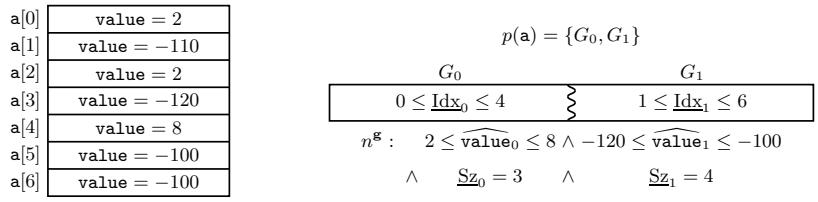
Concrete States. Our domain abstracts arrays of complex data structures. To highlight its core principle and simplify the formalization, we make two assumptions on the programs to analyze. First, there is no array access through pointer dereference (handling them would only require a product with a pointer domain), thus all array index expressions are of the form $\mathtt{a}[\mathtt{ex}]$. Secondly, all variables are either base type (e.g., scalar) variables (denoted by $\mathbb{X}$) or arrays of structures (denoted by $\mathbb{A}$). Structures are considered arrays of length 1, and arrays of scalars are considered arrays of structures made of a single field. A concrete state σ is a partial function mapping basic cells (base variables and fields of array elements) into values (which are denoted by $\mathbb{V}$). We let $\mathbb{N}$ denote non-negative integers and $\mathbb{F}$ denote the set of fields. Thus, the set $\mathbb{S}$ of concrete states is defined by $\mathbb{S} = (\mathbb{A} \times \mathbb{N} \times \mathbb{F} \cup \mathbb{X}) \to \mathbb{V}$. More specifically, the set of all fields of elements of array $\mathtt{a}$ are denoted by $\mathbb{F}_{\mathtt{a}}$, and the set of valid indexes in $\mathtt{a}$ is denoted by $\mathbb{N}_{\mathtt{a}}$.

Non-contiguous array partition. Our analysis partitions each array into one or several *groups* of cells. A group is represented by an abstraction G_i of the set of indexes of its elements, where subscript i identifies the group. We let $\mathbb{G}$ denote the set of group names $\{G_i \mid i \geq 0\}$. An *array partition* is a function $p : \mathbb{A} \to \mathcal{P}(\mathbb{G})$ which maps each array variable to a set of groups. We always enforce the constraint that groups of distinct arrays should have distinct names, to avoid confusion ($\forall a_1, a_2 \in \mathbb{A}, a_1 \neq a_2 \Rightarrow p(a_1) \cap p(a_2) = \emptyset$). To express properties of group contents, sizes, and indexes, we adjoin numeric abstract values to partition p. This numeric information is split into a conjunction made of two parts.

First, a global component $n^{\mathbf{g}}$ constrains base type variables, group sizes and group fields. Group fields are marked as *summary dimensions* [10] in $n^{\mathbf{g}}$, that is as numeric abstract domain dimensions that account for one or more concrete cell(s), whereas base type variables and group sizes are non-summary dimension, i.e., each of them represents exactly one concrete cell.

Second, for each group G_i, the index $\underline{\mathrm{Idx}}_i$ is constrained by a numeric abstract value n^i. This second component is needed because our abstract domain allows empty groups, and when group G_i is empty, $\underline{\mathrm{Idx}}_i$ has no value, which is expressed by $n^i = \bot$. Intuitively, in the concrete level, $\underline{\mathrm{Idx}}_i$ denotes a possibly empty set of values (an empty group example will be provided in Section 7.2).

To sum up, an abstract element is a pair $(p, \overrightarrow{n})$ where $\overrightarrow{n}$ is a tuple $(n^{\mathbf{g}}, n^0, \ldots, n^{k-1})$, and p defines k array partitions. Our abstract domain is parameterized by the choice of a numeric abstract domain $\mathbb{N}^\sharp$, so as to tune the analysis precision and cost. In this paper, we use the octagon abstract domain [18].



(a) Concrete array X (b) Abstract state $a^\sharp$

Fig. 5. An abstraction in our domain

Example 1. Figure 5(a) displays a concrete state, with an array of integers `a` of length 7 (each cell is viewed as a structure with a single field `value`). Figure 5(b) shows an abstraction $a^\sharp = (p, \overrightarrow{n})$ into two groups G_0, G_1, where G_0 (resp., G_1) contains all cells storing a positive (resp., negative) values. This abstraction reveals the array stores no value in $[-99, 1]$.

Concretization. A *concrete numeric mapping* is a function ν, mapping each base type variable to one value, each structure field to a non empty set of values and each index to a possibly empty set of values. We write $\gamma_{\mathbb{N}^\sharp}$ for the concretization of numeric elements, which maps a set of numeric constraints $\overrightarrow{n}$ into a set of functions ν as defined above. The concretization $\gamma_{\mathbb{N}^\sharp}(n^i)$ of constraints over group G_i is such that, when $n^i = \bot$ and $\nu \in \gamma_{\mathbb{N}^\sharp}(n^i)$, then $\nu(\underline{\mathrm{Idx}}_i) = \emptyset$. Then, $\gamma_{\mathbb{N}^\sharp}(n^{\mathbf{g}}, n^0, \ldots, n^{k-1}) = \gamma_{\mathbb{N}^\sharp}(n^{\mathbf{g}}) \cap \gamma_{\mathbb{N}^\sharp}(n^0) \ldots \gamma_{\mathbb{N}^\sharp}(n^{k-1})$. A *valuation* is a function $\psi \in \Psi = \mathbb{G} \to \mathcal{P}(\mathbb{Z})$, and interprets each group by the set of indexes it represents in a given concrete state.

Additionally, we use the following four predicates to break up the definition of concretization:

$$
\begin{aligned}
\mathrm{P_v}(\psi) &\overset{\text{def.}}{\iff} \forall \mathtt{a} \in \mathbb{A},\ \textstyle\bigcup_{G_i \in p(\mathtt{a})} \psi(G_i) = \mathbb{N}_{\mathtt{a}} \\
&\qquad \wedge\ (\forall G_i, G_j \in p(\mathtt{a}), i \neq j \Rightarrow \psi(G_i) \cap \psi(G_j) = \emptyset) \\
\mathrm{P_b}(\sigma, \nu) &\overset{\text{def.}}{\iff} \forall \mathtt{v} \in \mathbb{X},\ \nu(\mathtt{v}) = \sigma(\mathtt{v}) \\
\mathrm{P_i}(\nu, \psi) &\overset{\text{def.}}{\iff} \forall \mathtt{a} \in \mathbb{A}, G_i \in p(\mathtt{a}),\ \psi(G_i) = \nu(\underline{\mathrm{Idx}}_i) \wedge |\psi(G_i)| = \nu(\underline{\mathrm{Sz}}_i) \\
\mathrm{P_c}(\sigma, \psi, \nu) &\overset{\text{def.}}{\iff} \forall \mathtt{a} \in \mathbb{A}, \mathtt{f} \in \mathbb{F}_{\mathtt{a}},\ G_i \in p(\mathtt{a}), j \in \psi(G_i), \sigma(\mathtt{a}, j, \mathtt{f}) \in \nu(\widehat{\mathtt{f}}_i)
\end{aligned}
$$

Predicate $\mathrm{P_v}(\psi)$ states that each array element belongs to *exactly one group* (equivalently, groups form a partition of the array indexes). Predicate $\mathrm{P_b}(\sigma, \nu)$ expresses that ν and σ consistently abstract base type variables. Predicate $\mathrm{P_i}(\nu, \psi)$ expresses that ν and ψ consistently abstract group indexes. Last, predicate $\mathrm{P_c}(\sigma, \psi, \nu)$ states σ and ν define compatible abstractions of groups contents.

Definition 1 (Concretization). *Concretization $\gamma_{\mathbb{P}}$ is defined by:*

$$
\gamma_{\mathbb{P}}(p, \overrightarrow{n}) \overset{def.}{::=} \{(\sigma, \psi, \nu) \mid \nu \in \gamma_{\mathbb{N}^\sharp}(\overrightarrow{n}) \wedge \mathrm{P_v}(\psi) \wedge \mathrm{P_b}(\sigma, \nu) \wedge \mathrm{P_i}(\nu, \psi) \wedge \mathrm{P_c}(\sigma, \psi, \nu)\}
$$

3.2 Relation predicates

The abstraction we have defined so far can describe non-contiguous groups of cells, yet lacks important predicates, that are necessary for the analysis. Let us consider assignment `parent` = `mproc`[`child`].`mparent` in `cleanup` (Figure 3(a)). Numeric constraints localize `child` in $[0, 23]$, but this information does not determine precisely which group does cell `mproc`[`child`] belong to. In particular, the analysis will ignore from that point whether `parent` is the index of a valid process descriptor or not. To avoid this imprecision, we extend abstract states with *relation predicates*, that express properties such as the membership of the value of a variable in a group. They are defined by the grammar below:

Definition 2 (Relation predicates).

$$
\begin{array}{rlll}
r & ::= r \wedge r & & \textit{a conjunction of predicates} \\
 & \mid \textbf{true} & & \textit{empty} \\
 & \mid \mathtt{v} \lhd G^{\mathtt{a}} & \textit{where } \mathtt{v} \in \mathbb{X} & \textit{var-index predicate} \\
 & \mid \widehat{\mathtt{f}}_i \lhd G^{\mathtt{a}} & \textit{where } \mathtt{f} \in \mathbb{F}_{\mathtt{a}}, G_i \in p(\mathtt{a}) & \textit{content-index predicate} \\
G^{\mathtt{a}} & ::= G^{\mathtt{a}} \cup G_i & \textit{where } G_i \in p(\mathtt{a}) & \textit{a disjunction of groups in } \mathtt{a} \\
 & \mid G_i & \textit{where } G_i \in p(\mathtt{a}) &
\end{array}
$$

A relation predicate r is a conjunction of atomic predicates. Predicate $\mathtt{v} \lhd G^{\mathtt{a}}$ means the value of variable `v` is an index in G^a, where G^a is a disjunction of a set of groups of array `a`. Similarly, predicate $\widehat{\mathtt{f}}_i \lhd G^{\mathtt{a}}$ means that all fields `f` of cells in group i are indexes of elements of $G^{\mathtt{a}}$. As an example, if $G^{\mathtt{a}} = G_1 \cup G_3$, then $\mathtt{v} \lhd G^{\mathtt{a}}$ expresses that the value of `v` is either the index of a cell in G_1 or the index of a cell in G_3.

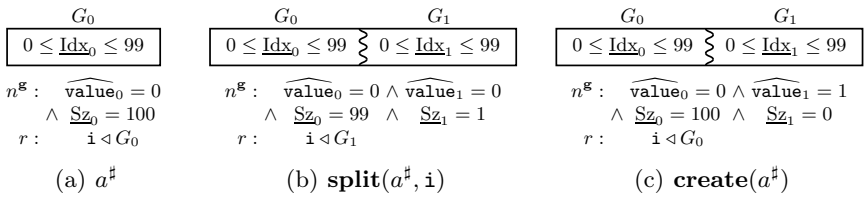
Example 2. We consider function `cleanup` of Figure 3(a). The pre-condition for the analysis of Figure 4 is based on correctness property $\mathcal{C}$, hence partitions `mproc` in three groups, thus $p(\mathtt{mproc}) = \{G_0, G_1, G_2\}$. Additionally, `cleanup` should be called on a valid process descriptor different from that of `init`, hence `child` should be in group G_2, which corresponds to predicate $\mathtt{child} \lhd G_2$. Then `parent` is initialized as the parent of `child`. Since $\widehat{\mathtt{mparent}}_2 \lhd G_0 \cup G_2$, `parent` is a valid process descriptor index, and the analysis derives $\mathtt{parent} \lhd G_0 \cup G_2$. Hence, at point ①, the analysis will derive relations $r = \mathtt{child} \lhd G_2 \wedge \mathtt{parent} \lhd G_0 \cup G_2 \wedge \ldots$.

Similarly, in the **else** branch of condition **if**(`parent` == 2), the analysis derives that $\mathtt{parent} \lhd G_2$.

Concretization. We now extend the concretization to account for relations. First, we let ψ be defined on disjunction of groups, and let $\psi(G_0 \cup \ldots \cup G_i) = \psi(G_0) \cup \ldots \cup \psi(G_i)$. We write $\mathbb{D}^\sharp$ for the set of triples $(p, \overrightarrow{n}, r)$.

Definition 3 (Abstract states and their concretization). *An* abstract state $a^\sharp$ *is a triple* $(p, \overrightarrow{n}, r) \in \mathbb{D}^\sharp$. *The* concretization $\gamma_{\mathbb{D}^\sharp}$ *is defined by:*

$$
\begin{array}{rl}
\gamma_{\mathbb{D}^\sharp}(p, \overrightarrow{n}, r) & ::= \{\sigma \mid \exists \psi, \nu,\ (\sigma, \psi, \nu) \in \gamma_{\text{aux}}(p, \overrightarrow{n}, r)\} \\
\gamma_{\text{aux}}(p, \overrightarrow{n}, \textbf{true}) & ::= \gamma_{\mathbb{P}}(p, \overrightarrow{n}) \\
\gamma_{\text{aux}}(p, \overrightarrow{n}, \mathtt{v} \lhd G^{\mathtt{a}}) & ::= \{(\sigma, \psi, \nu) \in \gamma_{\mathbb{P}}(p, \overrightarrow{n}) \mid \sigma(\mathtt{v}) \in \psi(G^{\mathtt{a}})\} \\
\gamma_{\text{aux}}(p, \overrightarrow{n}, \mathtt{f}_i \lhd G^{\mathtt{a}}) & ::= \{(\sigma, \psi, \nu) \in \gamma_{\mathbb{P}}(p, \overrightarrow{n}) \mid \forall k \in \psi(G_i),\ \sigma(\mathtt{a}, k, \mathtt{f}) \in \psi(G^{\mathtt{a}})\} \\
\gamma_{\text{aux}}(p, \overrightarrow{n}, r_0 \wedge r_1) & ::= \gamma_{\text{aux}}(p, \overrightarrow{n}, r_0) \cap \gamma_{\text{aux}}(p, \overrightarrow{n}, r_1)
\end{array}
$$



(a) $a^\sharp$ (b) $\mathbf{split}(a^\sharp, \texttt{i})$ (c) $\mathbf{create}(a^\sharp)$

Fig. 6. Partition splitting and creation in array `a` from abstract state $a^\sharp$

4 Basic operators on partitions

In this section, we define basic operations on partitions (such as creation and merge), that abstract transfer functions and operators rely on.

Splitting and creation. Unless specified otherwise, our analysis initially partitions each array into a single group, with no contents constraint. Additional groups get introduced during the analysis, by two basic operations:

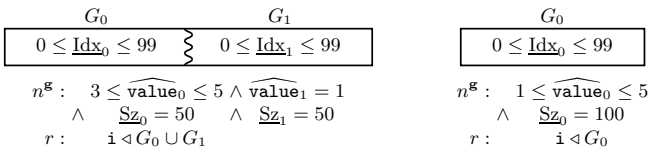
1. Operator **split** replaces a group with two groups, that inherit the properties of the group they replace (also, membership in the old group turns into membership in the join of the new groups). It is typically applied to materialize a cell of a given index (in the group bounds) and enable a strong update.
2. Operator **create** introduces an empty group and is used to generalize abstract states in join and widening (note any field property is satisfied by the empty group; the analysis selects properties depending on the context).

Both operators preserve concretization.

Example 3. Figure 6(a) defines an abstract state $(p, \overrightarrow{n}, r)$ with a single array, fully initialized to 0, and represented by a single group. Applying operator **split** to that abstract state and to index `i` produces the abstract state of Figure 6(b), where G_1 is a group with exactly one element, with the same constraints $\underline{\text{Idx}}$ and $\widehat{\texttt{value}}$ as in the previous state. Similarly, Figure 6(c) shows a possible result for **create**.

Merging groups. Fine partitions with many groups can provide great precision but may incur increased analysis cost. Therefore, the analysis can also force the fusion of several groups into one by calling operation **merge** on a set of groups. This is performed either as part of join and widening or when transfer functions detect some groups get assigned similar values.

Example 4. Figure 7(a) defines an abstract state $a^\sharp$ which describes an array with two groups. Applying **merge** to $a^\sharp$ and set $\{0, 1\}$ produces the state shown in Figure 7(b), with a single group and coarser predicates, obtained by joining the constraints over the contents of the initial groups.



(a) $a^\sharp$ (b) $\mathbf{merge}(a^\sharp, \{0, 1\})$

Fig. 7. Merging in abstract state $a^\sharp$

Reduction. Our abstract domain can be viewed as a product abstraction and can benefit from *reduction* [4]. In $a^\sharp = (p, \overrightarrow{n}, r)$, components $\overrightarrow{n}$ and r may help refining each other. For instance, in Figure 4, the analysis infers at point ① that $\mathtt{parent} \triangleleft G_0 \cup G_2$ and $\underline{\mathrm{Idx}}_0 = 2$. Combining this with the numerical information derived from test $\mathtt{parent} == 2$, the analysis should derive that $\mathtt{parent} \triangleleft G_0$ (i.e., `parent` is the index of `init`). Conversely, r may refine the information on $\overrightarrow{n}$: if $\mathtt{child} \triangleleft G_2$, then group G_2 has at least one element, thus $\underline{\mathrm{Sz}}_2 \geq 1$.

Such steps are performed by a *partial reduction* operator **reduce**, which strengthens the numeric and relation predicates, without changing the global concretization [4] (the optimal reduction would be overly costly to compute). This reduction is done lazily: for instance, the analysis will attempt to generate relations between `i` and $\underline{\mathrm{Idx}}_i$ only when `i` is used as an index to access the array G_i corresponds to.

Basic operations **split**, **create**, **merge** and **reduce** are sound:

Theorem 1 (Soundness). *If $a^\sharp$ is an abstract state, $\mathtt{t}$ an array, G_i a group, then $\gamma_{\mathbb{D}^\sharp}(a^\sharp) \subseteq \gamma_{\mathbb{D}^\sharp}(\mathbf{split}(a^\sharp, \mathtt{t}, G_i))$ and $\gamma_{\mathbb{D}^\sharp}(\mathbf{create}(a^\sharp, \mathtt{t})) = \gamma_{\mathbb{D}^\sharp}(a^\sharp)$. Moreover, if S is a set of groups, $\gamma_{\mathbb{D}^\sharp}(a^\sharp) \subseteq \gamma_{\mathbb{D}^\sharp}(\mathbf{merge}(a^\sharp, \mathtt{t}, S))$. Similarly,* **reduce** *does not change concretization.*

5 Transfer functions

Our analysis of C programs proceeds by forward abstract interpretation [3]. In this section, we study the abstract transfer functions for tests and assignments.

5.1 Analysis of conditions

In the concrete level, if `ex` is an expression, test `ex`? filters out states that do not let `ex` evaluate into `TRUE`. Its concrete semantics can thus be defined as a function over sets of states, by $\forall \mathcal{S} \subseteq \mathbb{S},\ [\![\mathtt{ex}?]\!](\mathcal{S}) = \{\sigma \in \mathcal{S} \mid [\![\mathtt{ex}]\!](\sigma) = \mathtt{TRUE}\}$.

Intuitively, the abstract interpretation of a test from abstract state $a^\sharp = (p, \overrightarrow{n}, r)$ can directly improve the constraints in the numeric component $\overrightarrow{n}$, which can then be propagated into r by **reduce**. The numeric test will derive new constraints only over non summary dimensions, thus tests over fields of groups that contain more than one element will not refine abstract values.

When a test involves an array cell as in $\texttt{a[i]} == 0?$, and if the group that cell belongs to cannot be known precisely, a more precise post-condition can be derived by performing a *locally disjunctive analysis*, that applies numeric test to each possible group, and then joins the abstract states. For instance, if $\texttt{i} \triangleleft G_0 \cup G_1$, the analysis will analyze test $\texttt{a[i]} == 0?$ for both $\texttt{i} \triangleleft G_0$ and $\texttt{i} \triangleleft G_1$, join the results of both tests, and apply operator **reduce** afterwards. Note that the abstract test operator does not change the partition, thus this join boils down to applying the abstract join $\mathbf{join}_{\mathbb{N}^\sharp}$ of numeric abstract domain $\mathbb{N}^\sharp$ and set intersection to relations viewed as sets of atomic relations. The resulting join operator, limited to cases where both arguments have the same partitioning is defined by $\mathbf{join}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1)) = (p, \mathbf{join}_{\mathbb{N}^\sharp}(\overrightarrow{n}_0, \overrightarrow{n}_1), r_0 \cap r_1)$. It is sound: $\forall i \in \{0, 1\}, \gamma_{\mathbb{D}^\sharp}(p, \overrightarrow{n}_i, r_i) \subseteq \gamma_{\mathbb{D}^\sharp}(\mathbf{join}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1)))$.

Abstract transfer function **test** is *sound* in the sense that:

$$\forall \sigma \in \gamma_{\mathbb{D}^\sharp}(a^\sharp),\ [\![\texttt{ex}]\!] = \texttt{TRUE} \Longrightarrow \sigma \in \gamma_{\mathbb{D}^\sharp}(\mathbf{test}(\texttt{ex}, a^\sharp))$$

Example 5. We consider the analysis of the code studied in Section 2. At the beginning of the first iteration of the loop, `i` is equal to 0, so $\texttt{mproc[i]}$ may be in G_1 or in G_2. Then, the analysis of test $\texttt{mproc[i].mpflag} > 0$ will locally create two disjuncts corresponding to each of these groups. However, in the case of G_1, $\widehat{\texttt{mpflag}}_1 = 0$, thus the numeric test $\widehat{\texttt{mpflag}}_1 > 0$ will produce abstract value $\bot$ denoting the empty set of states. Therefore, only the second disjunct contributes to the abstract post-condition. Thus, the analysis derives $\texttt{i} \triangleleft G_2$.

5.2 Assignment

Given l-value `lv` and expression `ex`, the concrete semantics of assignment $\texttt{lv} = \texttt{ex}$ writes the value of `ex` into the cell `lv` evaluates to. It can thus be defined as a function over states, by $[\![\texttt{lv} = \texttt{ex}]\!](\sigma) = \sigma[[\![\texttt{lv}]\!](\sigma) \leftarrow [\![\texttt{ex}]\!](\sigma)]$.

In the abstract level, given abstract pre-condition $a^\sharp = (p, \overrightarrow{n}, r)$, an abstract post-condition for $\texttt{lv} = \texttt{ex}$ can be done in three steps: (1) materialization of the memory cell that gets updated, (2) call to $\mathbf{assign}_{\mathbb{N}^\sharp}$ in $\mathbb{N}^\sharp$ [14], and update of the relations, and (3) reduction of the resulting abstract state.

Materialization. When `lv` denotes an array cell, it should get *materialized* into a group consisting of a single cell, before strong updates can be performed on $\overrightarrow{n}$ and r. To achieve this, the analysis computes which group(s) `lv` may evaluate into in abstract state $a^\sharp$. If there is a single such group G_i, that contains a single cell (i.e., $\underline{\text{Sz}}_i = 1$), then materialization is already achieved. If there is a single such group G_i, but $\underline{\text{Sz}}_i$ may be greater than 1, then the analysis calls **split** in order to divide G_i into a group of size 1 and a group containing the other elements. Last, when there are several such groups (e.g., when `lv` is $\texttt{a[i]}$ and $\texttt{i} \triangleleft G_0 \cup G_1$), the analysis first calls **merge** to merge all such groups and then falls back to the case where `lv` can only evaluate into a single group.

Note that in the last case, the merge of several groups may incur a loss in precision since the properties of several groups get merged before the abstract

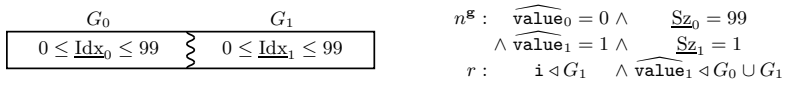


Fig. 8. Post-condition of assignment $\mathtt{a}[\mathtt{i}] = 1$

assignment takes place. We believe this loss in precision is acceptable here. The other option would be to produce a *disjunction* of abstract states, yet it would increase significantly the analysis cost and the gain in precision would be unclear, as programmers typically view those disjunctions of groups of cells as having similar roles. Our experiments (Section 7) did confirm this observation.

Constraints. New relations can be inferred after assignment operations in two ways. First, when both sides are base variables, they get propagated: for instance, if $\mathtt{u} \triangleleft G_i$, then after assignment $\mathtt{v} = \mathtt{u}$, we get $\mathtt{v} \triangleleft G_i$. Second, when the right hand side is an array cell as in $\mathtt{parent} = \mathtt{mproc}[\mathtt{child}].\mathtt{mparent}$ in the example of Section 2, the analysis first looks for relations between fields and indexes such as $\widehat{\mathtt{mparent}}_2 \triangleleft G_0 \cup G_2$, and propagate them to the l-value. In this phase, the numeric assignment relies on local disjuncts that are merged right after the abstract assignment, as we have shown in the case of condition tests (Section 5.1).

The abstract transfer function for assignment is sound in the sense that:

$$\forall \sigma \in \gamma_{\mathbb{D}^\sharp}(a^\sharp),\ \sigma[[\![\mathtt{lv}]\!](\sigma) \leftarrow [\![\mathtt{ex}]\!](\sigma)] \in \gamma_{\mathbb{D}^\sharp}(\mathbf{assign}(a^\sharp, \mathtt{lv}, \mathtt{ex}))$$

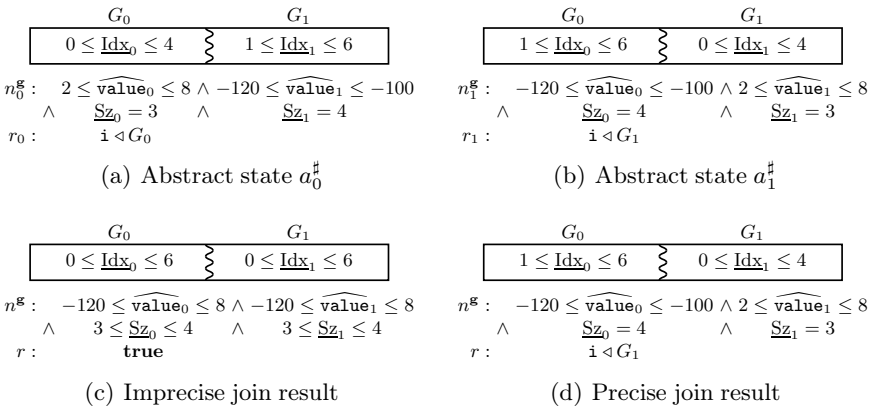
Example 6. We consider $\mathtt{a}[\mathtt{i}] = 1$ and abstract the pre-condition shown in Figure 6(a). The l-value evaluates into an index in G_0, but this group has several elements, thus it is split, as shown in Figure 6(b). Then, the assignment boils down to a strong update in G_1, and produces the post-condition shown in Figure 8. Note that reduction strengthens relations with $\widehat{\mathtt{value}}_1 \triangleleft G_0 \cup G_1$.

6 Join, widening and inclusion check

Our analysis proceeds by standard abstract interpretation, and uses widening and inclusion tests to compute abstract post-fixpoints for loops and abstract join for control flow union (e.g., after an **if** statement). All these operators face the same difficulties: when their inputs do not have a similar of clearly "matching" groups they have to re-partition the arrays so that precise information can be computed. We discuss this issue in detail in the case of join.

6.1 Join and the group matching problem

Let us consider two abstract states $a_0^\sharp, a_1^\sharp$ with the same number of groups for each array, that we assume to have the same names. Then, the operator $\mathbf{join}_\equiv$ introduced in Section 5.1 computes an over-approximation for $a_0^\sharp, a_1^\sharp$, by joining



(a) Abstract state $a_0^\sharp$ (b) Abstract state $a_1^\sharp$

(c) Imprecise join result (d) Precise join result

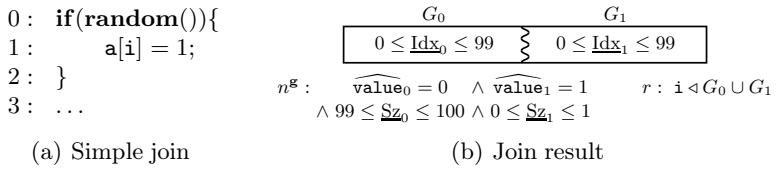
Fig. 9. Impact of the group matching on the abstract join

predicates for each group name, the global numeric invariants and the side relations. However, this straightforward approach may produce very imprecise results if applied directly. As an example, we show two abstract states $a_0^\sharp$ and $a_1^\sharp$ in Figure 9(a) and Figure 9(b), that are similar up to a group name permutation. The direct join is shown in Figure 9(c). We note that the exact size of groups and the tight constraints over `value` were lost. Conversely, if the same operation is done after a permutation of group names, an optimal result is found, as shown in Figure 9(d). This *group matching problem* is actually even more complicated in general as $a_0^\sharp, a_1^\sharp$ usually do not have the same number of groups.

To properly associate G_0 in Figure 9(a) with G_1 in Figure 9(b), the analysis should take into account the *group field properties*. This is achieved with the help of a ranking function $\mathbf{rank} : \mathbb{G} \times \mathbb{G} \to \mathbb{N}$, which computes a distance between groups in different abstract states by comparing their properties: $\mathbf{rank}(G_i, G_j)$ returns a monotone function of the number of common constraints over the fields and indexes of G_i and G_j in $\overrightarrow{n}_0$ and $\overrightarrow{n}_1$. A high value of $\mathbf{rank}(G_i, G_j)$ indicates G_i of $a_0^\sharp$ and G_j of $a_1^\sharp$ are likely to describe sets of cells with similar properties.

Using the set of $\mathbf{rank}(G_i, G_j)$ values, the analysis computes a *pairing* $\leftrightarrow$, that is a relation between groups of $a_0^\sharp$ and groups of $a_1^\sharp$ (this step relies on heuristics; a non optimal pairing will impact only precision, but not soundness) and then apply a group matching which transforms both arguments into "compatible" abstract states using the following (symmetric) principles:

- if there is no G_j such that $G_i \leftrightarrow G_j$, then an empty such group is created with **create**;
- if $G_i \leftrightarrow G_j$ and $G_i \leftrightarrow G_k$, then G_i is split into two groups, respectively paired with G_j and G_k;
- if G_i is mapped only to G_j, G_j is mapped only to G_i, and $i \neq j$, then one of them is renamed accordingly.



(a) Simple join (b) Join result

Fig. 10. Join of a one group state with a two groups state

After this process has completed, a pair of abstract states are produced that have the same number of groups, and $\mathbf{join}_{\equiv}$ can be applied. This defines abstract join operator **join**. The soundness of **join** follows from the soundness of $\mathbf{join}_{\equiv}$ (trivial), and the soundness of **split** and **create**:

Theorem 2 (Soundness).

$$\forall a_0^\sharp, a_1^\sharp,\ \gamma_{\mathbb{D}^\sharp}(a_0^\sharp) \subseteq \gamma_{\mathbb{D}^\sharp}(\mathbf{join}(a_0^\sharp, a_1^\sharp)) \wedge \gamma_{\mathbb{D}^\sharp}(a_1^\sharp) \subseteq \gamma_{\mathbb{D}^\sharp}(\mathbf{join}(a_0^\sharp, a_1^\sharp))$$

Example 7. We assume `a` is an integer array of length 100 and `i` is an integer variable storing a value in $[0, 99]$, and consider the program of Figure 10(a). At the exit of the **if** statement, the analysis needs to join the abstract states shown in Figure 6(a) (that has a single group) and in Figure 8 (that has two groups). We note that G_0 in Figure 6(a) has similar properties as G_0 in Figure 8, thus they get paired. Moreover, G_1 in Figure 8 is paired to no group, so a new group is created (as in Figure 6(c), and paired to it. At that stage $\mathbf{join}_{\equiv}$ applies, and returns the abstract state shown in Figure 10(b).

6.2 Widening

The widening algorithm is similar to that of join. The restriction of widening to *compatible* abstract states is defined by $\mathbf{widen}_{\equiv}((p, \overrightarrow{n}_0, r_0), (p, \overrightarrow{n}_1, r_1)) = (p, \mathbf{widen}_{\mathbb{N}^\sharp}(\overrightarrow{n}_0, \overrightarrow{n}_1), r_0 \cap r_1)$ (note that r_0, r_1 are finite sets of relations, and intersections of finite sets of relations naturally terminates).

The group matching algorithm of Section 6.1 does not ensure termination, as it could create more and more groups. Therefore **widen** relies on a slightly modified group matching algorithm, which will never call **split** and **create**. Instead, it will always match each group of an argument to at least one group of the other argument, and call **merge** when two (or more) groups of one argument are paired with a group of the other. This group matching ensures termination. Therefore, the resulting **widen** operator is a sound and terminating widening operator [3]. For better precision, the analysis always uses **join** for the first abstract iteration for a loop, and uses widening afterwards.

6.3 Inclusion check

To check the termination of sequences of abstract iterates over loops, and the entailment of post-conditions, the analysis uses a sound inclusion check operator **is_le**: when $\mathbf{is\_le}(a_0^\sharp, a_1^\sharp)$ returns `TRUE`, then $\gamma_{\mathbb{D}^\sharp}(a_0^\sharp) \subseteq \gamma_{\mathbb{D}^\sharp}(a_1^\sharp)$.

Like **join**, such an operator is easy to define on compatible abstract states, using an inclusion check operator $\mathbf{is\_le}_{\mathbb{N}^\sharp}$ for $\mathbb{N}^\sharp$: if $\mathbf{is\_le}_{\mathbb{N}^\sharp}(\overrightarrow{n}_0, \overrightarrow{n}_1) = \texttt{TRUE}$ and r_1 is included in r_0 (as a set of constraints), then $\gamma_{\mathbb{D}^\sharp}(p, \overrightarrow{n}_0, r_0) \subseteq \gamma_{\mathbb{D}^\sharp}(p, \overrightarrow{n}_1, r_1)$, hence we let $\mathbf{is\_le}_{\equiv}$ return `TRUE` in that case.

The group matching algorithm for **is_le** is different, although it is based on similar principles. Indeed, it modifies the groups in the left argument so as to construct an abstract state with the same groups as the right argument, using **create**, **split** and **merge**.

7 Verification of the Minix Memory Management Process Table and experimental evaluation

We have implemented our analysis and evaluated how it copes with two classes of programs: (1) the Minix Memory Management Process Table, and (2) academic examples used in related works, where contiguity of groups is sometimes unnecessary for the verification. Our analyzer uses the MemCAD analyzer front-end, and the Apron [14] implementation of octagons [18].

7.1 Verification of memory management part in Minix

The main data-structure of the Memory Management operating system service of Minix 1.1 is the MMPT `mproc`, which contains memory management information for each process. At start up, it is initialized by function `mm_init`, which creates process descriptors for `mm`, `fs` and `init`. After that, `mproc` should satisfy property $\mathcal{C}$ (Section 2). Then, it gets updated by system calls `fork`, `wait` and `exit`, which respectively create a process, wait for terminated children process descriptors be removed, and terminate a process. Each of these functions should be called only in a state that satisfies $\mathcal{C}$, and should return a state that also satisfies $\mathcal{C}$. System calls `wait` and `exit` call the complex utility function `cleanup` discussed in Section 2, to reclaim descriptors of terminated processes.

If property $\mathcal{C}$ was violated, several critical issues could occur. First, system calls could crash due to out-of-bound accesses, e.g., when accessing `mproc` through field `mparent`. Moreover, higher level, hard to debug issues could occur, such as the persistence of dangling processes, that would never be eliminated.

Therefore, we verified (1) that `mm_init` properly establishes $\mathcal{C}$ (with no pre-condition), and (2) that `fork`, `wait` and `exit` preserve $\mathcal{C}$ using our analysis (i.e., the analysis of each of these functions from pre-condition $\mathcal{C}$ returns a post-condition that also satisfies $\mathcal{C}$). Note that function `cleanup` was inlined in `wait` and `fork` in a recursion free form (currently not supported by our analyzer), as well as statements irrelevant to `mproc`.

Program	LOCs	Verified property	Time(s)	Max. groups	Description
`mm_init`	26	establishes $\mathcal{C}$	0.092	4	Minix MMPT: `mproc` init
`fork`	22	preserves $\mathcal{C}$	0.109	3	Minix MMPT sys call
`exit`	68	preserves $\mathcal{C}$	5.41	7	Minix MMPT sys call
`wait`	70	preserves $\mathcal{C}$	5.41	7	Minix MMPT sys call
`complex`	21	$\forall i \in [0, 54], a[i] \geq -1$	0.296	4	Example from [5]
`int_init`	8	$\forall i \in [0, N], a[i] = 0$	0.025	3	Array initialization

Fig. 11. Analysis results (timings measured on Ubuntu 12.04.4, with 16 Gb of RAM, on an Intel Xeon E3 desktop, running at 3.2 GHz)

Our tool achieves the verification of all these four functions. The results are shown in the first four lines of the table in Figure 11, including analysis time and peak number of groups for array `mproc`.

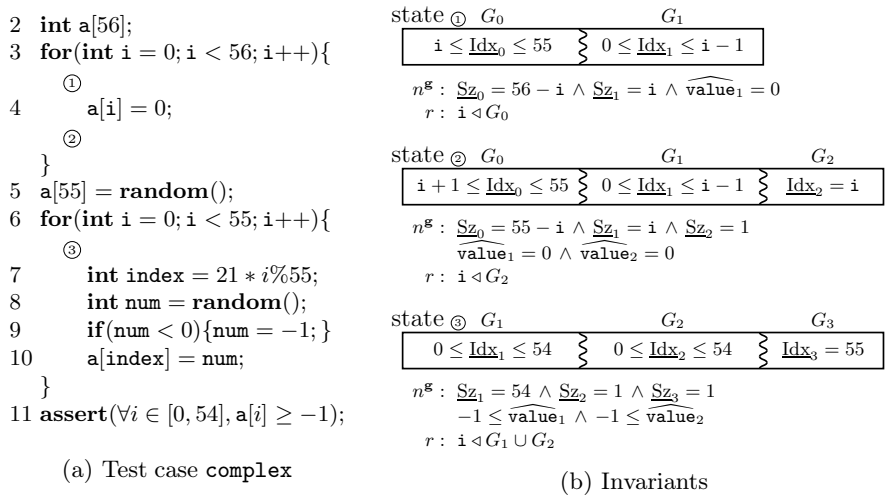
The analysis of `mm_init` and `fork` is very fast. The analysis of `exit` and `wait` also succeeds, although it is more complex due to the intricate structure of `cleanup` (which consists of five loops and many conditions) which requires 194 joins. Despite this, the maximum number of groups remains reasonable (seven in the worst case).

7.2 Application on other cases

We now consider a couple of examples from the literature, where arrays are used as containers, i.e., where the relative order of groups does not matter for the program's correctness. The purpose of this study is to examplify other examples of cases our abstract domain is adequate for. Program `int_init` consists of a simple initialization loop. Our analysis succeeds here, and can handle other cases relying on basic segments, although our algorithms are not specific to segments (and are geared towards the abstraction of non contiguous partitions).

Moreover, Figure 12 shows `complex`, an excerpt of an example from [5]. The second example is challenging for most existing techniques, as observed in [5] since resolving `a[index]` at line 10 is tricky. As shown in Figure 11, our analysis handles these two loops well, with respectively 4 and 3 groups.

The invariant of the first initialization loop in Figure 12 is abstract state ① (at line 4): group G_1 accounts for initialized cells, whereas cells of G_0 remain to be initialized. The analysis of `a[i]` $= 0$; from ① materializes a single uninitialized cell, so that a strong update produces abstract state ②. At the next iteration, and after increment operation `i++`, widening merges G_2 with G_1, which produces abstract state ① again. At loop exit, the analysis derives G_0 is empty as $56 \leq \underline{\text{Idx}}_0 \leq 55$. At this stage, this group is eliminated. The analysis of the second loop converges after two widening iterations, and produces abstract state ③. We note that group G_3 is kept separate, while groups G_1 and G_2 get merged when the assignment at line 10 is analyzed (Section 5.2). This allows to prove the assertion at line 11.



(a) Test case `complex` (b) Invariants

Fig. 12. Array random accesses

8 Related work and conclusion

In this paper, we have presented a novel abstract domain that is tailored for arrays, and that relies on partitioning, without imposing the constraint that the cells of a given group be contiguous.

Most array analyses require each group be a contiguous array *segment*. This view is used both in abstract interpretation based static analysis tools [5,11,13] and in tools based on invariant generation, model checking and theorem proving [1,15,16,17,19]. We believe that both approaches are adequate for different sets of problems: segment based approaches are adequate to verify algorithms that use array to order elements, such as sorting algorithms, while our segment-less approach works better to verify programs that use arrays as dictionaries.

Other works target dictionary structures and summarize non contiguous sets of cells, that are not necessarily part of arrays. In particular, [8,9] seeks for a unified way to reason about pointers, scalars and arrays. These works are orthogonal to our approach, as we strive to use properties specific to arrays in order to reason about the structure of groups. Therefore, [8,9] cannot express the invariants presented in Section 2 for two reasons: (1) the *access paths* cannot describe the contents of array elements as an interval or with other numeric constraints; (2) they cannot express *content-index* predicates. Similarly, HOO [6] is an effective abstract domain for containers and JavaScript open objects. As it uses a set abstract domain [7], it has a very general scope but does not exploit the structure of arrays, hence would sacrifice efficiency in such cases.

Last, template-base methods [2,12] are very powerful invariant generation techniques, yet require user supplied templates and can be quite costly.

Our approach has several key distinguishing factors. First, it not only relies on index relation, but also exploits semantic properties of array elements, to select groups. Second, relation predicates track lightweight properties, that would not be captured in a numerical domain. Last, it allows empty groups, which eliminated the need for any global disjunction in our examples (a few assignments and tests benefit from cheap, local disjunctions). Finally, experiments show it is effective at inferring non trivial array invariants with non contiguous groups, and verify a challenging operating system data-structure.

References

1. F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. In *TACAS*, 2014.
2. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, 2007.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
5. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.
6. A. Cox, E. Chang, and X. Rival. Automatic analysis of open objects in dynamic language programs. In *SAS*, 2014.
7. A. Cox, E. Chang, and S. Sankaranarayanan. Quic graphs: Relational invariant generation for containers. In *ECOOP*, 2013.
8. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, 2010.
9. I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.
10. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimension. In *TACAS*, 2004.
11. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, 2005.
12. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, 2008.
13. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, 2008.
14. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.
15. R. Jhala and K. McMillan. Array abstraction from proofs. In *CAV*, 2007.
16. L. Kovac and A. Voronkov. Finding loop invariants for programs over array using a theorem prover. In *FASE*, 2009.
17. K. McMillan. Quantified invariant generation using an interpolation saturation prover. In *TACAS*, 2008.
18. A. Miné. The octagon abstract domain. In *HOSC*, 2006.
19. M. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS*, 2009.
20. P. Sotin and X. Rival. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*, 2012.