Automatic Verification of Embedded System Code Manipulating Dynamic Structures Stored in Contiguous Regions

Jiangchao Liu, Liqian Chen, Xavier Rival

Automatic Verification of Embedded System Code Manipulating Dynamic Structures Stored in Contiguous Regions

Jiangchao Liu
Laboratory of Software Engineering of Complex Systems, College of Computer, National University of Defense Technology
Changhsha, China
jliu@di.ens.fr

Liqian Chen
Laboratory of Software Engineering of Complex Systems, College of Computer, National University of Defense Technology
Changhsha, China
lqchen@nudt.edu.cn

Xavier Rival
ENS, INRIA, CNRS and PSL*
Paris, France
rival@di.ens.fr

ABSTRACT

User-space programs rely on memory allocation primitives when they need to construct dynamic structures such as lists or trees. However, low-level OS kernel services and embedded device drivers typically avoid resorting to an external memory allocator in such cases, and store structure elements in contiguous arrays instead. This programming pattern leads to very complex code, based on data-structures that can be viewed and accessed either as arrays or as chained dynamic structures. The code correctness then depends on intricate invariants mixing both aspects. We propose a static analysis that is able to verify such programs. It relies on the combination of abstractions of the allocator array and of the dynamic structures built inside it. This approach allows to integrate program reasoning steps inherent in the array and in the chained structure into a single abstract interpretation. We report on the successful verification of several embedded OS kernel services and drivers.

1 INTRODUCTION

While user-space programs usually rely on memory allocation primitives provided by the OS to manage dynamic memory, low-level codes such as embedded device drivers or low-level OS services typically manage their own memory using a custom allocation scheme. The most common way to achieve this is to create a static array, and use it as a *pool of memory cells*, which can be used directly in order to create dynamic structures, like lists or trees. This pattern is much more complex and harder to get correct than using a regular memory allocator, due to the intricacy of the underlying invariants. In essence, it embeds the memory manager into the user code.

We show an instance of this pattern in Figure 1, that consists of a task manager taken from a proprietary real-time embedded OS designed for aerospace (that we later refer to as AOS). This task manager maintains three disjoint sets of tasks, that are respectively *ready*, *sleeping*, and *suspended*. Each group of tasks corresponds to a singly linked list, and the three corresponding lists are stored in a single array, which serves as a memory cells pool. Three variables `ready`, `sleep`, and `suspend` store the index of the first element of each list. Moreover, each list element stores a reference to the next element defined as its index in the array in field `next`. Tasks in state ready are ordered by their order of priority, which is stored in field `prio`. The declaration is shown in Figure 1(a), and an example state is depicted in Figure 1(b). Moreover, the task manager implements system calls (not shown), that operate on this structure, including `init` (initialization of the array and variables), `create` (search of a free slot in the array, and insertion in the list of tasks that are ready), `stop` (removal of a task —the corresponding cell becomes free), and `schedule` (move of a task from one list to another). Similar code can be found in many OS services that need to manage tasks, or in device drivers that need to manage resources. It is also common in low-level embedded codes, as it alleviates the need for a separate memory allocator. On the other hand, it makes the code of the operations on the table (that we later refer as *primitive operations*, or for short, *primitives*) very complex, and hard to get right. Indeed, the operations over the pool of cells mix direct array cell accesses, or accesses following chains of pointers to list elements encoded as indexes. They also involve tricky side conditions, such as cases where any of the lists is or becomes empty. Moreover, they need to preserve sophisticated invariants, such as the well-formedness and the disjointness of the lists of tasks. In the context of embedded systems or critical softwares, such programming patterns induce serious safety concerns. To guarantee the correct behavior of components such as the task manager described above, we need to verify not only memory safety but also the preservation of complex structural invariants by all the primitive operations. For example, the process table of Figure 1 should be consistent at all times, which means the three lists should be well-formed, acyclic and disjoint, and variables should point to the head of each list. If either of these conditions ever gets broken, the task manager will not be able to operate correctly anymore, and will lose or ignore some tasks.

The verification of the task manager boils down to checking that all calls preserve the structural invariant of the cells pool: if any of these is called in a state that satisfies the structural invariants, it

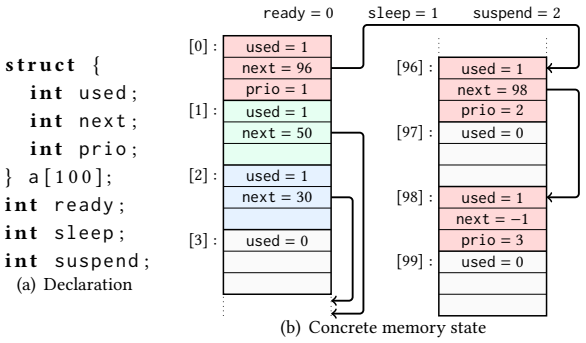


Figure 1: Process tables in a proprietary embedded OS

should return into a state that satisfies it as well. Due to the numbers of possible memory states and of execution paths in each primitive, verification by exhaustive testing does not appear feasible. Static analysis by abstract interpretation [7] aims at computing automatically sound program invariants, and avoids this path explosion issue. Applying this approach to the AOS case of Figure 1 requires the choice of an *abstract domain*, that is a set of abstract predicates supporting automatic analysis operations, and which can express all the properties that a manual proof would manipulate.

In the case of structural properties such as the internal state of Figure 1, a static analysis would need to describe both the lists and the array structure, and the relation between them. The structure of the task lists is highly dynamic since their size, topology and order vary. Thus describing these structures requires to *summarize* them, that is, to abstract them in a way independent from their size, topology and order. Shape analysis tools [5, 6, 23, 27] utilize abstract domains to compactly summarize inductive data-structures such as lists or trees, and can compute precise structural invariants. However, the structure shown in Figure 1 is beyond the scope of these tools, as it also heavily relies on the array structure, and primitives manipulate both array indexes and pointers. Array abstract domains [8, 12, 21] are designed to dynamically segment arrays into groups of cells with similar properties, and that can be abstracted together. While they could deal with pure value properties, they are unable to handle inductive dynamic structures like the lists of tasks of Figure 1. Thus, neither of these two classes of analyses can cope both with the array structure and with the nested lists.

In this paper, we propose a novel memory abstraction technique, which ties summary predicates of two very different forms throughout the analysis. It partitions the array into groups of cells that respectively correspond to each of the three lists of tasks, and to the free cells. Moreover, it ties to each group a composite summary predicate, that describes it both as a set of array cells, and as an inductive structure. An abstract state can be seen as a separating conjunction [22] of summary predicates. Furthermore, it allows to combine in a systematic manner the automatic analysis operations that handle both the array and the nested structures. It makes it possible to re-utilize most components of existing array and shape analyses, and turn them into a single, automatic analysis, able to cope with structures as shown in Figure 1. Due to this tight combination, we call this abstraction a *coalescing* of the underlying array and shape abstractions.

We make the following contributions: (1) we introduce (Section 2) and formalize (Section 3) memory abstraction coalescing, and construct a parametric abstraction for memory cells pools, (2) we present automatic static analysis algorithms to verify programs using pools of memory cells (Section 4), and (3) we report on the verification of a series of programs that manipulate pools of memory cells in OSes, drivers and embedded components (Section 5).

2 OVERVIEW

In this section, we describe the main principles of the coalescing abstraction, and show how it supports the verification of primitives manipulating memory pools. We focus on the structure displayed in Figure 1, and study the verification of the primitive in charge of the task creation system call, that is shown in Figure 2.

```
1  void create( int priority ){
2    int i = 0;
3    while( i < 100 ){
4      if( a[i].used == 0 ){
5        a[i].used = 1;
6        a[i].prio = priority;
7        break;
8      }
9      i++;
10   }
11   /* ... */ // insert a[i] into the "ready" list
12 }
```

Figure 2: Excerpt from the task creation system call (`create`)

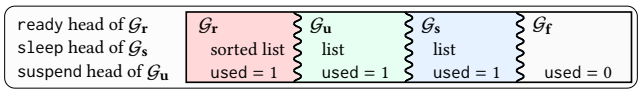
Structural correctness property and verification. Before we discuss the verification itself, we summarize the *structural consistency* that should hold at all times, between calls to primitives. We note C the conjunction of the three properties:

- (C_0) variables `ready`, `sleep` and `suspend` should point to the heads of three disjoint, well-formed acyclic singly-linked lists stored in the array, and such that the next element reference is stored as an index, in the `next` field of each cell, and that the end-of-list is encoded by index -1; the `used` field of all these elements should store value 1;
- (C_1) the set of array cells that appear in none of these three lists form the set of free slots; the `used` field of all these elements should store value 0;
- (C_2) the list with head `ready` is sorted with respect to the values in field `prio`.

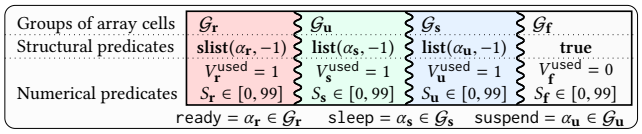
Thus the array should be divided into four groups (three lists of current tasks and a set of free slots). We note that the regions that correspond to each of these four groups are non-contiguous in general. The overall layout of this structure is shown in Figure 3(a), with the convention that jagged lines delimit the boarder of non necessarily contiguous groups of cells. Each group of tasks is described by a set of array indexes, and is tied to specific content properties. For instance, ready tasks are described by $\mathcal{G}_{\mathbf{r}}$, which stores a sorted list, and each cell in this group has a field `used` equal to 1, and `ready` stores the index of the first element of this list.

It is expected that the consistency property C gets temporarily broken in the middle of a primitive call, however, upon primitive completion, the property should hold again. Besides preserving C, primitives also typically have additional input/output requirements. As an example, when `create` is called in a state where there is at least one free slot, it will return in a state where a new task was created. Using a Hoare triple notation, the correctness of this primitive writes down as $\{C \wedge P\}\texttt{create(n)}\{C \wedge Q\}$. The verification of this triple by static analysis boils down to: (1) specifying an abstract state that soundly over-approximates the pre-condition $C \wedge P$; (2) letting the analysis compute a sound abstract post-condition, and letting it check that it entails the post-condition $C \wedge Q$.

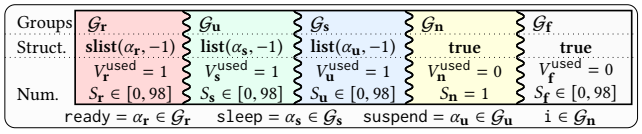
Coalescing abstraction. To carry out this automatic verification, we first need to identify an abstraction that is able to express the consistency property C, and all the properties that the analysis should manipulate. To make the specification of C simple, the abstraction



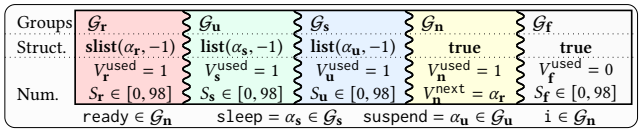
(a) High-level structure layout



(b) Abstract state



(c) Analysis intermediate state



(d) Analysis exit state

Figure 3: Abstraction of a memory pool state and analysis

should follow the high-level layout of Figure 3(a). Moreover, to be effective, it should build on top of the existing memory abstractions used in shape analysis and array analyses, so as to reuse existing predicates and algorithms.

First, the array can be divided into four disjoint areas, which naturally invites to use separation logic [22]. Thus, we describe the state with a formula of the form $\mathcal{G}_\mathbf{r} * \mathcal{G}_\mathbf{s} * \mathcal{G}_\mathbf{u} * \mathcal{G}_\mathbf{f}$, where $\mathcal{G}_\mathbf{r}, \mathcal{G}_\mathbf{s}, \mathcal{G}_\mathbf{u}$ and $\mathcal{G}_\mathbf{f}$ denote memory predicates representing regions in the array and $*$ is the *separating conjunction* and asserts that these four terms describe pairwise disjoint memory regions. This abstract state is represented graphically in Figure 3(b). Array analyses such as [8, 14] build upon abstract states that partition arrays in this manner, and [21] proposed an abstraction able to deal with non-contiguous regions, as found in Figure 3(a). These analyses describe the value of numerical fields using summary numerical dimensions [12], that is, abstract variables, which may describe *several* concrete cells. We follow this convention here, and let $V_\mathbf{r}^{\mathsf{used}}$ denote a numerical dimension describing the values stored in the field used of the elements of $\mathcal{G}_\mathbf{r}$. In our case, $V_\mathbf{r}^{\mathsf{used}}$ is equal to 1, as shown in Figure 3(b). Similarly, $V_\mathbf{s}^{\mathsf{used}}$, $V_\mathbf{u}^{\mathsf{used}}$ and $V_\mathbf{f}^{\mathsf{used}}$ describe the value of this field in the other groups. We also note the number of elements of each group can be bounded. For instance, we have $S_\mathbf{r} \in [0, 99]$, where $S_\mathbf{r}$ is the size of group $\mathcal{G}_\mathbf{r}$.

Secondly, the structures of the task lists can be described using inductive predicates in separation logic. As an example, if α, α' denote symbolic addresses, a segment of a singly linked list starting at address α and ending at address α' can be described as follows (for the sake of concision, we do not expand all the fields):

$$\mathbf{list}(\alpha, \alpha') ::= \mathbf{emp} \wedge \alpha = \alpha' \vee \exists \alpha'',\ \alpha \cdot \mathtt{next} \mapsto \alpha'' * \mathbf{list}(\alpha'', \alpha')$$

Therefore, the formula $\mathbf{list}(\alpha_\mathbf{s}, -1)$ (where sleep = $\alpha_\mathbf{s}$) describes the structure of the list of sleeping tasks in $\mathcal{G}_\mathbf{s}$. Similar formulas describe the structure of $\mathcal{G}_\mathbf{r}$ and of $\mathcal{G}_\mathbf{u}$ (note that $\mathcal{G}_\mathbf{r}$ requires a different inductive predicate **slist** since it consists of a sorted list).

The *coalescing abstraction* aims at tying together the array and inductive summaries one by one. Thus the coalesced abstract state writes down $\mathcal{G}_\mathbf{r} * \mathcal{G}_\mathbf{s} * \mathcal{G}_\mathbf{u} * \mathcal{G}_\mathbf{f}$ where $\mathcal{G}_k = \mathcal{A}_k \wedge \mathcal{I}_k$ and:

- $\mathcal{A}_k$ summarizes a group of cells of the array, and ties numerical predicates to summary dimensions representing the fields;
- $\mathcal{I}_k$ summarizes an inductive structure.

As an example, in the case of $\mathcal{G}_\mathbf{s}$, $\mathcal{A}_\mathbf{s}$ expresses that $V_\mathbf{s}^{\mathsf{used}} = 1$, sleep = $\alpha_\mathbf{s}$ and $S_\mathbf{s} \in [0, 99]$, and $\mathcal{I}_\mathbf{s}$ is $\mathbf{list}(\alpha_\mathbf{s}, -1)$. In the case of $\mathcal{G}_\mathbf{f}$, $\mathcal{I}_\mathbf{f}$ is **true**, as the next fields define no structure over the elements of this group. We formalize this abstraction in Section 3.

Static analysis and verification. We now overview the principles of the automatic static analysis to verify functions like create (Figure 2), starting from the pre-condition defined by C and $S_\mathbf{f} \geq 1$. The post-condition computation proceeds by forward abstract interpretation [7], which means the analysis implements functions to compute an over-approximation of each construction of the language, in terms of coalesced abstract states. In particular, a sound loop invariant is obtained as the widening of a sequence of abstract iterates. Moreover, the verification that a post-condition is satisfied boils down to a conservative implication checking among abstract states. In the following of this section, we discuss two analysis steps that are representative of the whole analysis, namely the analysis of an assignment statement, and the generalization process underlying widening and abstract states implication checking.

We first consider the assignment a[i].used = 1 at line 5 in Figure 2. To compute a post-condition for such a statement, the analysis needs to (1) localize the cell designated by a[i].used, that is, to identify to which group(s) it belongs, and (2) update the predicates of that group. Due to the condition at line 4, the constraints over used in all groups entail that a[i] may only belong to $\mathcal{G}_\mathbf{f}$. However, $\mathcal{G}_\mathbf{f}$ may contain several elements, thus the update step needs to account for the case where most elements of $\mathcal{G}_\mathbf{f}$ are unchanged. This situation is called a *weak update*, and it reduces the accuracy of analysis results. To avoid this issue, the analysis should split group $\mathcal{G}_\mathbf{f}$ before it performs the update. This step is called *materialization*, and isolates a[i] into a group of length 1, which supports a strong update. The corresponding state is shown in Figure 3(c). In that abstract state, not only the assignment at line 5 but also the subsequent assignments can be analyzed very precisely. More generally, the analysis of statements such as condition tests and assignments will also perform materialization. In Figure 2, materialization only operates on groups of array cells and numerical predicates ($\mathcal{A}_k$), but in general, it also needs to refine ($\mathcal{I}_k$) simultaneously, as we will show in Section 4.3.

Conversely, the analysis of loops and the verification of abstract post-conditions require to generalize abstract states. We consider the abstract state observed after create locates a free slot (Figure 3(c)) and inserts it into the ready list. In Figure 3(d), we show the abstract state obtained over the branch where the new task is inserted at the head of the list of ready tasks. Intuitively, that state corresponds to a particular configuration of the property C shown in Figure 3(b), and where $\mathcal{G}_\mathbf{r}$ has at least one element, up to

the merging of $\mathcal{G}_{\mathbf{n}}$ and $\mathcal{G}_{\mathbf{r}}$ in Figure 3(d). To establish this abstract state inclusion, the analysis needs to reconstruct a summary for the group of ready tasks. This *folding* process is at the basis of abstract states join, widening and implication checking (Section 4.4).

3 THE COALESCING ABSTRACTION

In this section, we formalize the *coalescing abstraction*: we define the abstract states (the predicates that the analysis manipulates), and their concretization (the concrete states that they represent). The coalescing construction that we present is generic, and agnostic with regard to the underlying abstractions even though we focus on the case where they describe arrays and inductive structures.

Notations. To simplify notations, and without loss of generality, we assume that programs use a single array $\mathtt{a}$, the elements of which are C structures. We let $\mathbb{F}$ denote the set of fields of these structures. We let $\mathbb{I}$ stand for the set of indexes. We write $\mathbb{V}$ (resp., $\mathbb{X}$) for the set of values (resp., variables). We also assume all array accesses are of the form $\mathtt{a}[\mathtt{v}]$, where $\mathtt{v}$ is an integer variable. A concrete memory state is a partial function mapping basic cells (variables and fields of array cells) into values, denoted as σ. The set of concrete states is defined by $\sigma \in \mathbb{S} = (\mathbb{I} \times \mathbb{F} \cup \mathbb{X}) \to \mathbb{V}$.

To represent abstract constraints, we assume a set of symbolic abstract variables $\bar{\mathbb{A}}$ (typically noted α_i) that denote values or sets of values. We also assume a numerical abstract domain [7] with summary dimensions [12]. We write $\bar{\mathbb{N}}$ for this abstract domain, and $\gamma_{\bar{\mathbb{N}}} : \bar{\mathbb{N}} \to \mathcal{P}(\bar{\mathbb{A}} \to \mathbb{V} \uplus \mathcal{P}(\mathbb{V}))$ for its concretization function where $\uplus$ denotes disjoint union.

Summarizing memory abstractions. As observed in Section 2, coalescing combines two memory abstractions to produce a new memory abstraction, thus we first set up a general definition of this concept. In this paper, we always consider abstract memory states that write down as a separating conjunction of terms, so we follow this layout here. As we noted, the analysis of the examples of Figure 2 requires to describe either individual cells, or summary areas, where individual cells can be either materialized out of a summary predicate for materialization, or folded into a summary predicate for generalization. The definition below formalizes this, while keeping the precise structure of summaries abstract, as it will be instantiated later with several different kinds of summaries:

Definition 1 (Memory abstraction). *A* memory abstraction *consists of a triple* $(\bar{\mathbb{M}}, \gamma_{\bar{\mathbb{M}}}, \leftrightarrow)$ *made of:*

- *a* set of abstract memories $\bar{\mathbb{M}}$ *defined by the grammar shown in Figure 4; an element* $\bar{t} \in \bar{\mathbb{T}}$ *(resp.,* $\bar{m} \in \bar{\mathbb{M}}$*) is called an* abstract term *(resp., an* abstract state*);*
- *a* concretization function $\gamma_{\bar{\mathbb{M}}} : \bar{\mathbb{M}} \to \mathcal{P}(\mathbb{S})$ *defined following the principles of separation logic, as shown in Figure 4;*
- *a* summarization relation $\leftrightarrow$ *between summaries and finite sets of abstract memories, such that, for all summary* $\mathbf{sum}(\alpha_i)$*, numerical constraints* $\bar{n}$ *and finite set* $\bar{M} \in \mathcal{P}_{\text{fin}}(\bar{\mathbb{M}})$*, we have:*

$$(\mathbf{sum}(\alpha_i) \wedge \bar{n}) \leftrightarrow \bar{M} \Longrightarrow \gamma_{\bar{\mathbb{M}}}(\mathbf{sum}(\alpha_i) \wedge \bar{n}) = \cup\{\gamma_{\bar{\mathbb{M}}}(\bar{m}) \mid \bar{m} \in \bar{M}\}$$

Intuitively, an abstract state is a separating conjunction of formulas, and each of these formulas is the conjunction of a basic memory predicate (that we refer to as a term) and a collection of numerical predicates. A basic memory predicate describes either a single memory cell (variable, or array cell, with one or several fields), or some sort of inductive predicate. Moreover, the summarization relation describes how summaries may be turned into more concrete memory descriptions, as part of materialization.

$$\begin{array}{rcll}
\bar{m}\ (\in \bar{\mathbb{M}}) & ::= & \exists \alpha_0, \ldots, \alpha_m,\ (\bar{t}_0 \wedge \bar{n}_0) * \ldots * (\bar{t}_p \wedge \bar{n}_p) & \\
\bar{t}\ (\in \bar{\mathbb{T}}) & ::= & \mathbf{emp} & \text{(empty region)} \\
& | & \mathtt{v} \mapsto \alpha_i & \text{(variable)} \\
& | & \alpha_i \cdot \mathtt{f} \mapsto \alpha_j & \text{(array cell)} \\
& | & \mathbf{sum}(\alpha_i) & \text{(summary predicate)} \\
\bar{n}_i & \in & \bar{\mathbb{N}} & \text{(numerical abstract predicates)} \\
\alpha_i & \in & \bar{\mathbb{A}} & \text{(symbolic abstract variables)}
\end{array}$$

$$\sigma \in \gamma_{\bar{\mathbb{M}}}(\exists \alpha_0, \ldots, \alpha_m,\ (\bar{t}_0 \wedge \bar{n}_0) * \ldots * (\bar{t}_p \wedge \bar{n}_p))$$
$$\iff \exists \nu \in \bar{\mathbb{A}} \to \mathcal{P}(\mathbb{V}),\ \left\{ \begin{array}{l} \forall i,\ \nu \in \gamma_{\bar{\mathbb{N}}}(\bar{n}_i) \\ \sigma, \nu \vdash \bar{t}_0 * \ldots * \bar{t}_p \end{array} \right.$$
$$\sigma, \nu \vdash \mathbf{emp} \iff \sigma = \emptyset$$
$$\sigma, \nu \vdash \bar{t}_0 * \ldots * \bar{t}_p \iff \exists \sigma_0, \ldots, \sigma_p,\ \left\{ \begin{array}{l} \forall i,\ \sigma_i, \nu \vdash \bar{t}_i \\ \sigma = \sigma_0 \uplus \ldots \uplus \sigma_p \end{array} \right.$$
$$\sigma, \nu \vdash \mathtt{v} \mapsto \alpha_i \iff \sigma = \{\mathtt{v} \mapsto \nu(\alpha_i)\}$$
$$\sigma, \nu \vdash \alpha_i \cdot \mathtt{f} \mapsto \alpha_j \iff \sigma = \{(\nu(\alpha_i), \mathtt{f}) \mapsto \nu(\alpha_j)\}$$

Figure 4: Summarizing memory abstractions syntax and concretization (parameterized by the definition of sum)

In this section, we assume that $\bar{n}$ expresses conjunctions of linear inequalities and set constraints. As this paper is not specifically focusing on value abstract domains, we represent these constraints as logical formulas, although our implementation relies on a proper abstract domain, and static analysis algorithms perform sound approximation of value constraints whenever required.

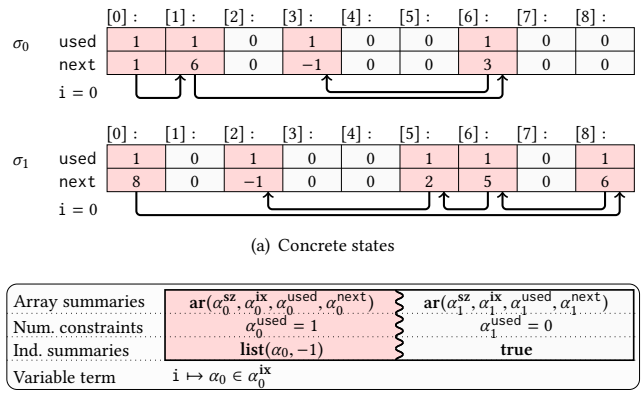
Abstraction of arrays. The description of the array properties considered in Section 2 requires an array abstraction that can tie numerical properties to possibly non-contiguous groups of cells. It boils down to a memory abstraction in the sense of Definition 1, with the appropriate notion of summary.

In this paragraph, we assume $\mathbb{F} = \{\mathtt{f}_0, \mathtt{f}_1\}$. An *array summary* is a summary $\mathbf{ar}(\alpha^{\mathbf{sz}}, \alpha^{\mathbf{ix}}, \alpha_0, \alpha_1)$, that describes a group of array cells, where $\alpha^{\mathbf{sz}}$ describes the number of cells in the group, $\alpha^{\mathbf{ix}}$ the set of their indexes, and α_0, α_1 the sets of values stored in their fields. Such a summary predicate may either be turned into the memory predicate **emp** that stands for the empty region when it is empty, or to the disjoint union of a single cell and of a smaller group when it is not empty. This actually defines a summarization relation $\leftrightarrow^{\mathbf{a}}$:

$$(\mathbf{ar}(\alpha^{\mathbf{sz}}, \alpha^{\mathbf{ix}}, \alpha_0, \alpha_1) \wedge \alpha^{\mathbf{ix}} = \emptyset \wedge \bar{n}) \leftrightarrow^{\mathbf{a}} \{\mathbf{emp}\}$$
$$(\mathbf{ar}(\alpha^{\mathbf{sz}}, \alpha^{\mathbf{ix}}, \alpha_0, \alpha_1) \wedge \alpha^{\mathbf{sz}} > 0 \wedge \alpha_\upsilon \in \alpha^{\mathbf{ix}} \wedge \bar{n}) \leftrightarrow^{\mathbf{a}}$$
$$\left\{ \begin{array}{l} \exists \alpha_0', \alpha_1',\ (\alpha_\upsilon \cdot \mathtt{f}_0 \mapsto \alpha_0' * \alpha_\upsilon \cdot \mathtt{f}_1 \mapsto \alpha_1' * \mathbf{ar}(\alpha_u^{\mathbf{sz}}, \alpha_u^{\mathbf{ix}}, \alpha_0, \alpha_1)) \\ \wedge\ (\alpha^{\mathbf{sz}} = \alpha_u^{\mathbf{sz}} + 1 \wedge \alpha^{\mathbf{ix}} = \alpha_u^{\mathbf{ix}} \uplus \{\alpha_\upsilon\} \wedge \bar{n} \wedge \bar{n}[\alpha_0/\alpha_0', \alpha_1/\alpha_1']) \end{array} \right\}$$

The second case duplicates α_0, α_1 and creates α_0', α_1' to account for the value of the fields of the materialized cell. This array predicate generalizes to any number of fields, and leads to a memory abstraction that can summarize array regions:

Definition 2 (Non-contiguous array abstraction). *The* non-contiguous array abstraction *(or for short,* array abstraction*) is the triple* $(\bar{\mathbb{M}}^{\mathbf{a}}, \gamma^{\mathbf{a}}_{\bar{\mathbb{M}}}, \leftrightarrow^{\mathbf{a}})$ *where* $\bar{\mathbb{M}}^{\mathbf{a}}$ *is defined as the instance of* $\bar{\mathbb{M}}$



(a) Concrete states

(b) Coalesced abstract state

Figure 5: Coalescing abstraction example

where summary predicates are occurrences of $\mathbf{ar}$*, where* $\leftrightarrow^{\mathbf{a}}$ *is the above summarization relation, and where* $\gamma^{\mathbf{a}}_{\bar{\mathbb{M}}}$ *is defined by* $\leftrightarrow^{\mathbf{a}}$*.*

We note that the concretization $\gamma^{\mathbf{a}}_{\bar{\mathbb{M}}}$ is defined by the summarization relation $\leftrightarrow^{\mathbf{a}}$, which is expected since summary predicates aim at supporting abstract predicates folding / unfolding.

Example 1. *As an example, we consider a simplified memory cells pool, which is based on a single list of active elements, and on two fields* `used` *and* `next`*. An element of index* `i` *is in the list if and only if* `a[i].used` *is equal to 1; then,* `a[i].next` *denotes the index of the next element; otherwise,* `a[i].used` *is equal to 0. Two example concrete states are shown in Figure 5(a). Elements in the active list are shown in red. This set of states can be described by an abstract state* $(\bar{t}_0 \wedge \bar{n}_0) * (\bar{t}_1 \wedge \bar{n}_1) * (\bar{t}_2 \wedge \bar{n}_2)$*, where:*

- $\bar{t}_0 = \mathbf{ar}(\alpha_0^{\mathbf{sz}}, \alpha_0^{\mathbf{ix}}, \alpha_0^{\mathtt{used}}, \alpha_0^{\mathtt{next}})$ *and* $\bar{n}_0 = (\alpha_0^{\mathtt{used}} = 1)$ *describe the group of active elements;*
- $\bar{t}_1 = \mathbf{ar}(\alpha_1^{\mathbf{sz}}, \alpha_1^{\mathbf{ix}}, \alpha_1^{\mathtt{used}}, \alpha_1^{\mathtt{next}})$ *and* $\bar{n}_1 = (\alpha_1^{\mathtt{used}} = 0)$ *describe the group of non-active elements;*
- $\bar{t}_2 = \mathtt{i} \mapsto \alpha_0$ *and* $\bar{n}_2 = \alpha_0 \in \alpha_0^{\mathbf{ix}}$ *describe variable* `i`*.*

Abstraction of inductive structures. So far, we have considered only the description of the properties relative to the array structure, so we now turn our attention to the inductive structures stored in each region. In Section 2, we have observed that these structures can be represented using inductive summaries, we defined a summary **list** for singly linked lists, and we noted sorted singly linked lists can also be described using an inductive summary predicate. More generally, an inductive summary predicate $\mathbf{i}$ is defined by a summarization relation of the form $\mathbf{i}(\alpha_0, \ldots, \alpha_k) \leftrightarrow^{\mathbf{i}} \{\bar{m}_0, \ldots, \bar{m}_p\}$ where $\bar{m}_0, \ldots, \bar{m}_p$ are made of terms the memory part of which consists either in individual memory cells or in other instances of the summary predicate $\mathbf{i}$ itself. Each of the terms $\bar{m}_0, \ldots, \bar{m}_p$ accounts for one of the ways to construct a structure; as an example the predicate **list** introduced in Section 2 comprises two such cases. We write $\bar{\mathbb{M}}^{\mathbf{i}}$ for the set of memory predicates where all summary predicates are either of the above form or the **true** predicate that describes any memory region.

Definition 3 (Inductive abstraction). *The* inductive memory abstraction *is the triple* $(\bar{\mathbb{M}}^{\mathbf{i}}, \gamma^{\mathbf{i}}_{\bar{\mathbb{M}}}, \leftrightarrow^{\mathbf{i}})$ *where* $\bar{\mathbb{M}}^{\mathbf{i}}$ *and* $\leftrightarrow^{\mathbf{i}}$ *are defined as above, and* $\gamma^{\mathbf{i}}_{\bar{\mathbb{M}}}$ *is the concretization function defined by* $\leftrightarrow^{\mathbf{i}}$*.*

Note that the summarization relation defines the concretization function as in the case of the array abstraction (Definition 2).

Example 2. *We consider the same structure as in Example 1. The region formed by the active elements stores a singly linked list whereas the other array cells satisfy no particular inductive property. As a consequence, this set of states can be described by an abstract state* $(\bar{t}_0 \wedge \bar{n}_0) * (\bar{t}_1 \wedge \bar{n}_1) * (\bar{t}_2 \wedge \bar{n}_2)$*, where:* $\bar{t}_0 = \mathbf{list}(\alpha_0, -1)$ *and* $\bar{n}_0 = \mathbf{true}$ *describe the group of active elements,* $\bar{t}_1 = \mathbf{true}$ *and* $\bar{n}_1 = \mathbf{true}$ *describe the group of non-active elements, and* $\bar{t}_2 = \mathtt{i} \mapsto \alpha_0$ *and* $\bar{n}_2 = \mathbf{true}$ *describe the state of variable* `i`*. In particular, we note that these constraints convey the fact that* `i` *points to the head of a singly linked list.*

Coalescing abstraction. We remarked in Section 2 that the analysis of programs like the `create` function of Figure 2 requires to reason simultaneously about arrays and inductive structures in a same region. To achieve this, the *coalescing abstraction* combines summaries locally:

Definition 4 (Coalescing abstraction). *Let* $(\bar{\mathbb{M}}^0, \gamma^0_{\bar{\mathbb{M}}}, \leftrightarrow^0)$ *and* $(\bar{\mathbb{M}}^1, \gamma^1_{\bar{\mathbb{M}}}, \leftrightarrow^1)$ *be two memory abstractions in the sense of Definition 1, with different sets of summary predicates. We call* coalescing abstraction *the memory abstraction* $(\bar{\mathbb{M}}^{\bowtie}, \gamma^{\bowtie}_{\bar{\mathbb{M}}}, \leftrightarrow^{\bowtie})$ *such that:*

- *the summary predicates in* $\bar{\mathbb{M}}^{\bowtie}$ *are of the form* $\mathbf{sum}^0 \wedge \mathbf{sum}^1$ *where* $\mathbf{sum}^0 \in \bar{\mathbb{M}}^0$ *and* $\mathbf{sum}^1 \in \bar{\mathbb{M}}^1$ *are summary predicates;*
- *the summarization relation* $\leftrightarrow^{\bowtie}$ *is defined by:*

$$(\mathbf{sum}^0 \wedge \mathbf{sum}^1 \wedge \bar{n}) \leftrightarrow^{\bowtie} \bar{M} \iff \begin{cases} & (\mathbf{sum}^0 \wedge \bar{n}) \leftrightarrow^0 \bar{M} \\ \wedge & (\mathbf{sum}^1 \wedge \bar{n}) \leftrightarrow^1 \bar{M} \end{cases}$$

- *the concretization function* $\gamma^{\bowtie}_{\bar{\mathbb{M}}}$ *is defined by* $\gamma^{\bowtie}_{\bar{\mathbb{M}}}(\mathbf{sum}^0 \wedge \mathbf{sum}^1 \wedge \bar{n}) = \gamma^0_{\bar{\mathbb{M}}}(\mathbf{sum}^0 \wedge \bar{n}) \cap \gamma^1_{\bar{\mathbb{M}}}(\mathbf{sum}^1 \wedge \bar{n})$*.*

In the rest of the paper, we focus on the case where $(\bar{\mathbb{M}}^0, \gamma^0_{\bar{\mathbb{M}}}, \leftrightarrow^0)$ is the array abstraction $(\bar{\mathbb{M}}^{\mathbf{a}}, \gamma^{\mathbf{a}}_{\bar{\mathbb{M}}}, \leftrightarrow^{\mathbf{a}})$ and $(\bar{\mathbb{M}}^1, \gamma^1_{\bar{\mathbb{M}}}, \leftrightarrow^1)$ is the inductive abstraction $(\bar{\mathbb{M}}^{\mathbf{i}}, \gamma^{\mathbf{i}}_{\bar{\mathbb{M}}}, \leftrightarrow^{\mathbf{i}})$ although the Definition 4 sets up a general notion of coalescing abstraction. We remark an important characteristic of coalescing: the structure of a coalesced summary $\mathbf{sum}^0 \wedge \mathbf{sum}^1$ deeply ties the structures of summary predicates $\mathbf{sum}^0$ and $\mathbf{sum}^1$. Indeed, when we consider $\mathbf{sum}^0 = \mathbf{ar}(\ldots)$ and $\mathbf{sum}^1 = \mathbf{list}(\ldots)$, then a memory region described by $\mathbf{sum}^0 \wedge \mathbf{sum}^1$ is either empty or non empty, thus the materialization of $\mathbf{ar}(\ldots) \wedge \mathbf{list}(\ldots)$ produces a disjunction of two elements which correspond to the case where both summaries unfold to the empty (resp., non empty) region.

Example 3. *We consider the structure of Example 1. Two abstractions of this structure were presented in Example 1 and Example 2, which respectively account for the array view and for the inductive view of the structure. Each abstraction consists of the separating conjunction of two summary predicates and one points-to predicate over a variable. The terms of these two abstractions describe regions that coincide, thus the whole structure can be accurately represented in the coalescing abstraction. For instance, the first group can be represented by* $\mathbf{ar}(\alpha_0^{\mathbf{sz}}, \alpha_0^{\mathbf{ix}}, \alpha_0^{\mathtt{used}}, \alpha_0^{\mathtt{next}}) \wedge \mathbf{list}(\alpha_0, -1) \wedge (\alpha_0^{\mathtt{used}} = 1)$*.*

cond :	$\bar{\mathbb{M}} \times \mathtt{Cond}$	$\longrightarrow$	$\bar{\mathbb{M}}$	local condition test
upd :	$\bar{\mathbb{M}} \times \mathtt{Lv} \times \mathtt{Expr}$	$\longrightarrow$	$\bar{\mathbb{M}}$	local update
mat :	$\bar{\mathbb{T}} \times \bar{\mathbb{A}}$	$\longrightarrow$	$\mathcal{P}_{\text{fin}}(\bar{\mathbb{M}})$	materialization
abs :	$\bar{\mathbb{M}}$	$\longrightarrow$	$\bar{\mathbb{T}}$	summarization
sel :	$\bar{\mathbb{T}} \times \bar{\mathbb{T}}$	$\longrightarrow$	$\mathbb{N}$	selection
$\sqcup, \triangledown$:	$\bar{\mathbb{M}} \times \bar{\mathbb{M}}$	$\longrightarrow$	$\bar{\mathbb{M}}$	join, widening
$\sqsubseteq$:	$\bar{\mathbb{M}} \times \bar{\mathbb{M}}$	$\longrightarrow$	**Bool**	inclusion check

Figure 6: Abstract operations of a memory domain $\bar{\mathbb{M}}$

The whole abstract state is shown in Figure 5(b) and highlights the coalescing of terms.

The structure shown in Figure 3(b) is similar, and requires four coalesced summaries, and three atomic terms for the variables.

4 STATIC ANALYSIS BY ABSTRACT INTERPRETATION

In this section, we elaborate automatic static analysis algorithms for the coalescing abstraction set up in Section 3. Static analyses based on the components of the coalescing abstraction rely on complex algorithms, and reinventing a novel static analysis directly from Definition 4 would duplicate a large part of this work. Instead, we aim at integrating existing analysis algorithms.

4.1 Language and concrete semantics

For the sake of concision, we focus on the smallest set of program constructions that require all the important analysis operations:

`s`	::=	`lv = expr`	assignment to cell field or variable
	|	**if**`( cond ) s`	condition
	|	**while**`( cond ) s`	loop
	|	`s0; s1`	sequence

Moreover, we let the semantics of a statement `s` be the function $[\![\mathtt{s}]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ that maps a set of states observed before executing `s` into the set of states that are observed after executing it (if `s` does not terminate, it returns the empty set). This semantics is adequate to tackle the verification problem stated in Section 2, as we are interested in proving properties of the form "any call starting in a state that satisfies $C \wedge P$ exits in a state that satisfies $C \wedge Q$". Therefore, the remaining of the section aims at computing an over-approximation for $[\![\mathtt{s}]\!]$.

4.2 Abstract domain and analysis

In this section, we use the abstract domain $\bar{\mathbb{M}}^{\bowtie}$ obtained by coalescing $\bar{\mathbb{M}}^{\mathbf{a}}$ and $\bar{\mathbb{M}}^{\mathbf{i}}$. The over-approximation of $[\![\mathtt{s}]\!]$ takes the form of a function $[\![\mathtt{s}]\!]^\sharp : \bar{\mathbb{M}}^{\bowtie} \to \bar{\mathbb{M}}^{\bowtie}$, and such that $[\![\mathtt{s}]\!] \circ \gamma^{\bowtie}_{\bar{\mathbb{M}}} \subseteq \gamma^{\bowtie}_{\bar{\mathbb{M}}} \circ [\![\mathtt{s}]\!]^\sharp$. As an example, $[\![\mathtt{s0; s1}]\!] = [\![\mathtt{s1}]\!] \circ [\![\mathtt{s0}]\!]$ so that $[\![\mathtt{s0; s1}]\!]^\sharp = [\![\mathtt{s1}]\!]^\sharp \circ [\![\mathtt{s0}]\!]^\sharp$ satisfies this soundness property. To construct this analysis, we define abstract operations in the coalescing domain from operations in $\bar{\mathbb{M}}^{\mathbf{a}}$ and $\bar{\mathbb{M}}^{\mathbf{i}}$. Figure 6 lists the operations that we use in the rest of this section. Each operation satisfies a soundness condition. For instance, **cond** computes a sound post-condition for a condition test statement: given abstract pre-condition $\bar{m}$ and condition `cond`, $\mathbf{cond}(\bar{m}, \mathtt{cond})$ returns an over-approximation of the set of states in $\gamma_{\bar{\mathbb{M}}}(\bar{m})$ that satisfy `cond`. To define the coalescing analysis, we assume that the underlying domains $\bar{\mathbb{M}}^{\mathbf{a}}$ and $\bar{\mathbb{M}}^{\mathbf{i}}$ provide each of these functions (e.g., we note them $\mathbf{cond}^{\mathbf{a}}$, $\mathbf{cond}^{\mathbf{i}}$), and we build a similar function in the coalescing domain (noted $\mathbf{cond}^{\bowtie}$). Similarly, **upd** over-approximates assignment. The functions **mat** and **abs** respectively refine and weaken abstract states, following $\leftrightarrow$. The function **sel** computes a measure of similarity of two abstract states. Last, $\sqcup$, $\triangledown$ and $\sqsubseteq$ conserviatively approximate concrete unions, widening and inclusion checking. In the following, we assume $\bar{\mathbb{M}}^{\mathbf{a}}$ and $\bar{\mathbb{M}}^{\mathbf{i}}$ provide these operations, and construct similar operations for $\bar{\mathbb{M}}^{\bowtie}$.

4.3 Post-condition and materialization

Localization. When an assignment or a condition test in the analyzed program accesses an l-value, the analysis first needs to identify what part of the abstract state represents this l-value, using either array or list information. If the l-value is a variable `i`, the cell is represented as a points-to term, since $\bar{\mathbb{M}}^{\bowtie}$ summarizes only array regions. If the l-value is of the form `a[i].f`, then the localization of the cell is done based on numerical and set constraints over the indexes, by checking for each group whether `i` may belong to it. When several solutions are found, the analysis needs to make case splits and to consider one case per solution.

Assignment to a materialized cell. We first consider an assignment operation `lv = expr`, and an abstract pre-condition $\bar{m}$ where each memory cell read or written in the assignment is described by a points-to term. Then, the computation of a post-condition boils down to the update of numerical constraints. The function **upd** (Figure 6) provides such a sound post-condition. When applied to a single term, the definition of $\mathbf{upd}^{\bowtie}$ boils down to $\mathbf{upd}^{\bowtie}(\bar{t}^{\mathbf{a}}_0 \wedge \bar{t}^{\mathbf{i}}_0 \wedge \bar{n}_0) = (\bar{t}^{\mathbf{a}}_1 \wedge \bar{t}^{\mathbf{i}}_1) \wedge (\bar{n}^{\mathbf{a}}_1 \wedge \bar{n}^{\mathbf{i}}_1)$, where $\mathbf{upd}^{\mathbf{a}}(\bar{t}^{\mathbf{a}}_0 \wedge \bar{n}_0) = \bar{t}^{\mathbf{a}}_1 \wedge \bar{n}^{\mathbf{a}}_1$ and $\mathbf{upd}^{\mathbf{i}}(\bar{t}^{\mathbf{i}}_0 \wedge \bar{n}_0) = \bar{t}^{\mathbf{i}}_1 \wedge \bar{n}^{\mathbf{i}}_1$. This simple definition generalizes to pre-conditions made of several terms, provided each cell read or written is present as a points-to term. However, it does not generalize to the case where either of the cells manipulated by the assignment is part of a summary.

Materialization. The algorithm to extract a cell is called *materialization*. It is present both in shape analyses [5, 23] and in array analyses [8, 12, 21]. The operation **mat** (Figure 6) achieves this: **mat** inputs an abstract term $\bar{t}$, and a symbolic variable that denotes an address in the region described by $\bar{t}$, and utilizes the summarization relation $\leftrightarrow$ to produce an abstract state that over-approximates $\bar{t}$ and where the cell of address α is represented exactly. The soundness of **mat** follows from the definition based on $\leftrightarrow$:

$$\mathbf{mat}(\mathbf{sum}(\alpha, \ldots) \wedge \bar{n}, \alpha) ::= \bar{M} \text{ where } (\mathbf{sum}(\alpha, \ldots) \wedge \bar{n}) \leftrightarrow \bar{M}$$

Given $\mathbf{mat}^{\mathbf{a}}, \mathbf{mat}^{\mathbf{i}}$, the definition of $\mathbf{mat}^{\bowtie}$ is done component-wise, and also following the definition of $\leftrightarrow^{\bowtie}$ (Definition 4):

$$\begin{aligned}&\mathbf{mat}^{\bowtie}(\bar{t}^{\mathbf{a}}_0 \wedge \bar{t}^{\mathbf{i}}_0 \wedge \bar{n}_0, \alpha)\\ &= \{(\bar{t}^{\mathbf{a}}_1 \wedge \bar{t}^{\mathbf{i}}_1) \wedge (\bar{n}^{\mathbf{a}}_1 \wedge \bar{n}^{\mathbf{i}}_1) \mid (\bar{t}^{\mathbf{a}}_1 \wedge \bar{n}^{\mathbf{a}}_1) \in \mathbf{mat}^{\mathbf{a}}(\bar{t}^{\mathbf{a}}_0 \wedge \bar{n}_0, \alpha)\\ &\qquad \wedge (\bar{t}^{\mathbf{i}}_1 \wedge \bar{n}^{\mathbf{i}}_1) \in \mathbf{mat}^{\mathbf{i}}(\bar{t}^{\mathbf{i}}_0 \wedge \bar{n}_0, \alpha)\}\end{aligned}$$

While this formula seems to follow straightforwardly from Definition 4, it also conveys an important property. The conjunction of numerical constraints may lead to pruning away some terms that have empty concretization. As an example, we observed in Section 3 that the summarization relation of the coalescing product

of an array abstraction and an abstraction based on list predicates generates simultaneously states where either the region is empty (so that we have an empty array region and an empty list), or non empty (so that, a same element can be extracted simultaneously from the array structure and from the list structure).

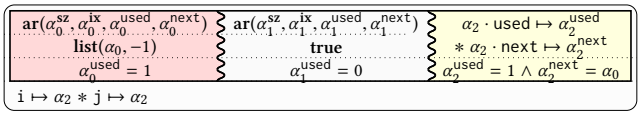
Assignment into a summary. We can now state the algorithm for the analysis of an assignment:

(1) localize each of the cells read or written, that is, determine to which term of the abstract pre-condition it belongs to;
(2) materialize the localized terms using $\mathbf{mat}^{\bowtie}$;
(3) apply $\mathbf{upd}^{\bowtie}$ to compute the post-condition.

The algorithm for the analysis of condition tests is similar. Due to the materialization phase, this algorithm returns a disjunction of abstract states in general, and we will address this with an abstract join operator in Section 4.4. We demonstrate these two algorithms in the following example.

Example 4. *We consider the abstract state of Example 3 (in Figure 5(b)), with an additional variable* `j`*, the constraint* $0 \leq \mathtt{j} < 9$*, and code* **if**`(a[j].used==0) { a[j].next=i; a[j].used=1; i=j; }`*:*

- *the condition reads the value of* `a[j]`*, which leads to the materialization of this cell; this materialization step produces a disjunction of two cases where* `a[j]` *denotes a group made of a single cell; the constraint on* `j` *implies it may belong to either pre-existing array regions, however the condition on the field* `used` *entails it may only belong to the group of free slots;*
- *as the condition step materializes* `a[j]`*, the subsequent assignments do not require further materialization, and simply apply* $\mathbf{upd}^{\bowtie}$*. The resulting abstract state is shown in the graphical form below. Array regions are shown in the top, and variables in the bottom. Remark that the first two array regions contain summaries, whereas the rightmost one contains a single cell.*



This shows an important property of coalescing abstraction: when a cell is localized in either underlying domains, it can immediately be materialized in both. The symmetric case (localization based on inductive predicates) is similar.

4.4 Analysis of loops and generalization

The materialization involved in the computation of post-conditions splits summary terms and increases the size of abstract states. Not only the computation of loop invariants but also the comparison of abstract states make it necessary to perform the opposite transformation. This transformation naturally divides into the *selection* of regions to abstract, and the application of *folding* based on $\leftrightarrow$. We first consider the latter, as it is more simple.

Folding of regions and creation of summary terms. By definition of the summarization relation, if $\bar{m}$ is an abstract state, and $\bar{t}$ is a summary term such that $\bar{t} \leftrightarrow \bar{M}$ and $\bar{m} \in \bar{M}$, then we can conclude that $\gamma_{\bar{\mathbb{M}}}(\bar{m}) \subseteq \gamma_{\bar{\mathbb{M}}}(\bar{t})$, which means that the abstract state $\bar{m}$ can be conservatively weakened into the summary abstract term $\bar{t}$. As stated in Figure 6, we require memory abstractions to provide a partial function **abs** that turns an abstract state into a single summary term. This function is partial as many abstract states cannot be accurately approximated by any summary term. The analyses based on array abstractions or inductive structures abstractions shown in Section 3 generally support a variant of this function. Therefore, we show how to construct $\mathbf{abs}^{\bowtie}$ from $\mathbf{abs}^{\mathbf{a}}$ and $\mathbf{abs}^{\mathbf{i}}$:

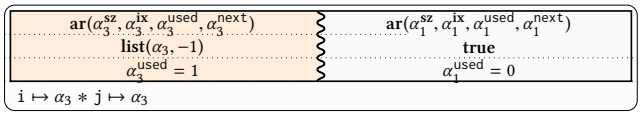
$$\begin{array}{lll} \text{if} & \mathbf{abs}^{\mathbf{a}}((\bar{t}_0^{\mathbf{a}} \wedge \bar{n}_0) * \ldots * (\bar{t}_k^{\mathbf{a}} \wedge \bar{n}_k)) = (\mathbf{sum}^{\mathbf{a}} \wedge \bar{n}^{\mathbf{a}}) \\ \text{and} & \mathbf{abs}^{\mathbf{i}}((\bar{t}_0^{\mathbf{i}} \wedge \bar{n}_0) * \ldots * (\bar{t}_k^{\mathbf{i}} \wedge \bar{n}_k)) = (\mathbf{sum}^{\mathbf{i}} \wedge \bar{n}^{\mathbf{i}}) \\ \text{then} & \mathbf{abs}^{\bowtie}((\bar{t}_0^{\mathbf{a}} \wedge \bar{t}_0^{\mathbf{i}} \wedge \bar{n}_0) * \ldots * (\bar{t}_k^{\mathbf{i}} \wedge \bar{t}_k^{\mathbf{i}} \wedge \bar{n}_k)) \\ & \qquad = ((\mathbf{sum}^{\mathbf{a}} \wedge \mathbf{sum}^{\mathbf{i}}) \wedge (\bar{n}^{\mathbf{a}} \wedge \bar{n}^{\mathbf{i}})) \end{array}$$

While this folding principle is rather simple, its implementation is more challenging. A first difficulty is that corresponding sets of terms should be summarized simultaneously, otherwise $\mathbf{abs}^{\bowtie}$ does not apply. A second difficulty is that the numerical predicates may need to be weakened during this folding step.

Example 5. *We consider the abstract state obtained in the end of Example 4, and merge the first and third array regions into one, using* $\mathbf{abs}^{\bowtie}$*. In* $\bar{\mathbb{M}}^{\mathbf{a}}$*, this results in the creation of a summary predicate described by* $\mathbf{ar}(\alpha_3^{\mathtt{sz}}, \alpha_3^{\mathtt{ix}}, \alpha_3^{\mathtt{used}}, \alpha_3^{\mathtt{next}})$ *that contains all the cells described by* $\mathbf{ar}(\alpha_0^{\mathtt{sz}}, \alpha_0^{\mathtt{ix}}, \alpha_0^{\mathtt{used}}, \alpha_0^{\mathtt{next}})$ *and the cell* $\alpha_2 \cdot \mathtt{used} \mapsto \alpha_2^{\mathtt{used}} * \alpha_2 \cdot \mathtt{next} \mapsto \alpha_2^{\mathtt{next}}$*. In* $\bar{\mathbb{M}}^{\mathbf{i}}$*, a segment summary is synthesized, since,*

$$\mathbf{abs}^{\bowtie}((\mathbf{list}(\alpha_0, -1) * \alpha_2 \cdot \mathtt{used} \mapsto \alpha_2^{\mathtt{used}} * \alpha_2 \cdot \mathtt{next} \mapsto \alpha_2^{\mathtt{next}}) \wedge (\alpha_2^{\mathtt{next}} = \alpha_0)) = \mathbf{list}(\alpha_2, -1)$$

The diagram below depicts the result, with a new summary region in the left (in orange on the figure). To avoid confusion, we let the summaries of the new group be labeled with subscript 3.



Selection of groups for abstract join and widening. The analysis of a condition statement should compute an over-approximation for the union of flow paths. Moreover, the abstract interpretation of a loop requires the computation of a sequence of iterates the convergence of which should be ensured by a widening operator [7]. These operators should *merge* groups and create summaries using $\mathbf{abs}^{\bowtie}$. However this operator may not always be able to produce a precise result, thus the analysis should determine which terms to merge. To perform this selection, we require memory abstractions to supply an operation **sel** that computes a measure of logical similarity of terms, and returns a non-positive value that is greater in absolute value when applied to terms that carry very different properties. As an example, a measure of the similarity of array summaries and points-to terms in an array region is provided by the opposite of the number of different associated numerical predicates. Then, the abstract join (resp., widening) of two abstract states $\bar{m}_0, \bar{m}_1$ is computed as follows:

(1) compute the measure of similarity of each pair $(\bar{t}_0, \bar{t}_1)$ where $\bar{t}_0$ is a term of $\bar{m}_0$ and $\bar{t}_1$ a term of $\bar{m}_1$;
(2) based on these measures of similarity, build a relation $\approx$, such that $\bar{t}_0 \approx \bar{t}_1$ if and only if $\bar{t}_0$ and $\bar{t}_1$ are very similar;
(3) when $\bar{t}_0 \approx \bar{t}_1$ and $\bar{t}_0' \approx \bar{t}_1$, replace $\bar{t}_0 * \bar{t}_0'$ with $\mathbf{abs}(\bar{t}_0, \bar{t}_0')$; repeat this step, and similar rewritings in $\bar{m}_1$ until $\approx$ describes a bijection between terms of $\bar{m}_0$ and $\bar{m}_1$;

(4) apply $\sqcup$ (resp., $\triangledown$) component-wise, following $\approx$.

Example 6. *To illustrate this algorithm, we consider the program:*

```
int j = f( );
if( 0 <= j && j < 9 && a[j].used == 0 ){
  a[j].next = i; a[j].used = 1; i = j; }
```

We assume that executions start from states described by the abstract state shown in Example 3 (Figure 5(b)). The states observed after executing the body of the **if** *branch are described by the abstract state shown in Example 4, and the states that do not enter the body of the* **if** *branch are described by the abstract state shown in Figure 5(b). Thus, the abstract post-condition for this program is the join of these two abstract states. The similarity relation pairs together regions associated to the constraint* `used` $= 1$*, and that contain list segment predicates. Note that* $\alpha_2 \cdot \texttt{next} \mapsto \alpha_0 * \ldots$ *gets weakened into* $\mathbf{list}(\alpha_2, \alpha_0)$*. Therefore, the similarity relation results in the merge of the leftmost and rightmost groups of the abstract state shown in Example 4, as discussed in Example 5. This process produces the abstract join, hence the post-condition of the code fragment.*

Inclusion checking. Inclusion checking is an abstract operation that takes two abstract states and attempts to prove inclusion. It is conservative and returns **false** when inclusion cannot be established. It is used both to check the convergence of abstract iterations over loops, and to check that abstract post-conditions are met. It relies on a similar principle as join and widening, and using $\sqsubseteq$ instead of $\sqcup$, $\triangledown$, thus we do not detail this operator here.

4.5 Analysis and soundness

The analysis computes post-conditions for programs, and is defined by induction over the syntax. When the analysis of a statement creates disjunctions due to materialization (Section 4.3), it applies abstract join to these disjuncts so as to produce a single post-condition. It also applies abstract join at branch merges and widening to compute loop invariants. It returns a sound post-condition, so as to conservatively verify the specification of each primitive:

Theorem 1 (Soundness). *For all program* `s` *and abstract precondition* $\bar{m}^{\bowtie} \in \bar{\mathbb{M}}^{\bowtie}$*, we have* $[\![\mathtt{s}]\!] \circ \gamma^{\bowtie}_{\bar{\mathbb{M}}}(\bar{m}^{\bowtie}) \subseteq \gamma^{\bowtie}_{\bar{\mathbb{M}}} \circ [\![\mathtt{s}]\!]^{\sharp}(\bar{m}^{\bowtie})$.

An important remark is that the analysis with the coalesced domain reuses fundamental algorithms of the underlying memory abstractions and removes the need to re-implement them completely. Instead, it ties them step by step to produce precise post-conditions in the combined domain.

5 EXPERIMENTS

This section reports on the evaluation of the analysis based on the coalescing abstraction for the verification of components of embedded operating systems. We evaluate (1) the expressiveness of our abstraction, namely its ability to describe structural invariants of programs that use an array as an allocation pool as described in Section 1, (2) the efficiency of the analysis to successfully verify real programs and, (3) its usability (the analysis should be easy to deploy and should effectively automate verification). To this end, we implemented the coalesced abstraction into the MemCAD static analyzer [26], and using the Apron implementation of the domain of linear inequalities [15]. This analysis tool is parameterized by the description of the structural consistency property C (Section 2).

System	TinyOS	AOS	Minix	Linux	Nordic
Module	task sched.	task sched.	memory mgmt	Eicon net driver	app. timer
Lists	1	3	2	1	1
Free slots	Yes	Yes	-	Yes	Yes
Tail ptr	Yes	-	-	Yes	-
Length info.	-	-	-	Yes	-
Sortedness	-	Yes	-	-	Yes
Primitives	2	5	4	3	2

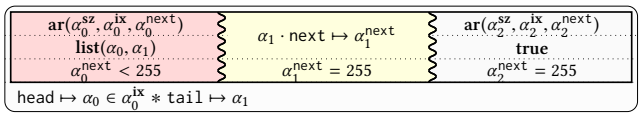
Table 1: Analyzed programs and consistency invariants

Experiments setup. We identified a set of target programs that implement their own memory allocation scheme using an array that stores dynamic structures. We list them below:

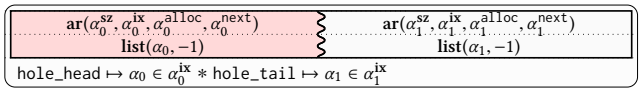
- **Task scheduler of TinyOS**: TinyOS [20] is an embedded OS designed specifically for network applications, and systems with a low-power CPU. It is written in nesC [11], an extension of C. This task scheduler manages tasks in a list stored in a large array.
- **Task scheduler of AOS**: This component was presented in Section 1 and manages tasks in three lists stored in a single array.
- **Memory manager of Minix**: Minix [25] is a Unix-like microkernel, that inspired many other kernels, including Linux. This memory manager relies on a list stored in an array to maintain the table of allocated memory blocks.
- **The Eicon Diva network driver for Linux**: Eicon [1] provides network chips for servers. This Diva network driver maintains queues of requests received by a network adapter, and stores them as a list in an array structure.
- **The Nordic nRF51 application timer**: Nordic nRF51 series [2] are chips for embedded ultra-low power wireless applications. Its timer application [3] stores information about applications in a sorted list stored in an array.

Table 1 shows the diversity of the data-structures these programs operate on: some store several lists in a single array whereas others store only one list; in all cases except Minix, some elements of the array stand for free slots; some structures maintain a tail pointer or length information in a separate variable; last, two of the five test cases use sorted lists. Moreover, each of these programs implements a set of primitives that modify the structure, typically to handle a system call or an operation in the management of the system.

Verification and expressiveness of the coalescing abstraction. The verification process is split into two stages that consist in (1) the specification of the structural consistency property and of the primitives, and (2) the automatic static analysis of a program of the form `assume(`$C \wedge P$`); prim( ); assert(`C`);` for each primitive `prim`, where `assume` (resp., `assert`) causes the analysis to conservatively assume (resp., attempt to verify) an abstract property. The consistency property is specified using a basic language to describe memory and value properties. We show this with the specification supplied for a couple of examples from our test set.



(a) TinyOS scheduler structural consistency C_{T}



(b) Minix memory manager structural consistency C_{M}

Figure 7: Structural consistency invariants

Example 7 (Verification of the TinyOS scheduler). *The TinyOS scheduler uses a singly linked list, with a variable that stores the index of the last element. The* `next` *field characterizes both the next element in the list of tasks, and also the free slots: for each free slot,* `next` *is equal to 255, whereas used slots have a* `next` *field value strictly lower than 255. Note that the last element of the list acts as a sentinel node (its* `next` *value is 255, and it does not correspond to a task). The consistency property C_{T} reflects this partition into three groups of cells (the group of tasks in the body of the list, the last element in the list and the free slots). The specification of this nested structure in the coalescing domain uses the list segment inductive predicate* **list** *introduced in Section 2. It is shown in Figure 7(a). The task scheduler implements two primitives* `tpush` *and* `tpop` *to reserve and free elements. Based on the definition of C_{T}, the verification of these primitives reduces to the static analysis of the programs below:*

```
assume(C_T);    int b = tpop();    assert(C_T);
assume(C_T ∧ 0 ≤ id ≤ 255);    tpush(id);    assert(C_T);
```

Example 8 (Verification of the Minix memory manager). *The Minix memory manager organizes memory blocks using two lists stored in the same array and that respectively record allocated blocks and free elements in the array. All the cells of the array belong to either of these two lists. Therefore, C_{M} partitions the array in exactly two regions, and coalesces each region with a list segment summary predicate introduced in Section 2, as shown in Figure 7(b). Moreover, there is no specific field indicating which group an element belongs to; instead, the partitioning of the array is solely guided by the two lists.*

Four primitives manipulate this structure: `tinit` *initializes the structure,* `allocmem` *searches for a free block and allocates it,* `freemem` *frees a memory block, and* `getmax` *traverses the list of allocated blocks to search for memory allocation information. As in Example 7, the verification of all these primitives boils down to the analysis of the following four short programs:*

```
assume(true);    tinit();    assert(C_M);
assume(C_M);    int ok = allocmem();    assert(C_M);
assume(C_M);    int ok = freemem(base);    assert(C_M);
assume(C_M);    int m = getmax();    assert(C_M);
```

The verification of the other three examples follows the same steps. The structural consistency property of AOS is depicted in Figure 3(b). The invariants associated to each of these examples can be expressed with coalesced abstract states similar to those presented in Example 7 and Example 8. Moreover, the array and inductive predicates required in the five test cases can be provided as parameters to our analyzer.

Sample	Defs. (LOCs)	Prims. (LOCs)	Spec. (lines)	Verified (prims.)	Time (s)
TinyOS	1	54	18	2 / 2	0.22
`tpush`		30		yes	0.11
`tpop`		24		yes	0.11
AOS	6	354	19	5 / 5	6.26
`tinit`		36		yes	0.12
`tcreate`		54		yes	0.81
`tstop`		83		yes	1.68
`tsched`		71		yes	1.36
`tstart`		110		yes	2.29
Minix	4	133	8	4 / 4	1.46
`tinit`		13		yes	0.19
`allocmem`		46		yes	0.38
`freemem`		59		yes	0.58
`getmax`		15		yes	0.31
Eicon	157	80	22	3 / 3	0.64
`insert`		43		yes	0.24
`delete`		18		yes	0.12
`traversal`		19		yes	0.28
Nordic	14	103	13	2 / 2	1.62
`tinsert`		56		yes	1.03
`tdelete`		47		yes	0.59

Table 2: Analysis results (Defs: size of the structure definitions in LOCs; Prims.: size of the code of the primitives in LOCs; Spec: size of the specifications supplied to the analysis tool; Verified: number of primitives verified out of the total; Time: total analysis time for the test case in seconds)

Analysis efficiency. We now discuss the analysis reports. The analyses were performed on a desktop with Intel Xeon E3 at 3.2 GHz with 16 Gb of RAM, and under Ubuntu 12.04.4. Table 2 summarizes the code sizes, analysis results and averaged timings for each primitive and for each test case (total time to verify all primitives). While these programs are all of small size, they all involve sophisticated invariants that require to reason both about array indexes and inductive pointer structures, so that they could not be analyzed without the coalescing domain. We distinguish the definition of the data structures and the body of the primitives to verify (e.g., in the Eicon driver, type definitions account for a very large part of the implementation). The coalescing analysis successfully verifies all the primitives of the five test cases, with respect to the structural consistency specifications, as shown in Example 7 and in Example 8. The invariants that are automatically computed ensure not only memory safety, but also the preservation of the structural consistency invariants. Analysis times are all of the order of at most a few seconds. As the programs that the analysis targets are not meant to be large, but subtle and involving sophisticated invariants (as is the case for all the examples in our test set), these timings are compatible with verification.

Usability of the analysis. Our verification scheme requires hand written specification of the structural consistency property. While

it would be possible to infer a candidate invariant by analyzing the primitives manipulating the structure, we believe that this approach would be very likely to compute an invariant that is *not* what the developer intends it to be. In this context, the verification approach appears as more reliable.

For each of our test cases, the time required for a non OS expert to identify and write down the specification of the structural consistency ranged from a few minutes up to a couple of hours. The developer of the target code would spend significantly less time to do so thanks to their knowledge of the structural consistency invariants. The specification language used by the tool is close to the intuitions underlying the graphical representations used throughout the paper.

6 RELATED WORK

Deductive techniques have achieved great successes in verifying OS components [13, 17]. Our goal is different, as we aim at automating the verification of common programming patterns encountered in low-level components of OSes and embedded software.

The foundation of coalescing is to carefully adjust the level of logical connectors in memory predicates, so as to precisely capture a family of overlaid structures while keeping the memory abstract states simple. Reduced product [7] provides a systematic way to introduce conjunctive reasoning in static analysis, yet its application to memory abstraction is often difficult. Shape analyses rely on it [19, 26] in order to describe overlaid structures. By contrast, our analysis applies non separating conjunction locally, hence it expresses stronger properties, and allows a simple synchronization of coalesced predicates attached to a same region. Per-field separating conjunction [10] can describe linked structures, but does not apply to index arithmetics, so it could not describe our array predicates.

Our analysis focuses on dynamic structures stored in arrays. Few analyses have been developed to tackle such nested structures. On one hand, a large family of works focus on numerical arrays, and use segment abstractions [8, 12, 14], which prevents the inference of properties of non-contiguous sets of cells. Similar abstractions have been used in invariant generation, model checking and theorem proving [4, 16, 18]. While such analyses can verify sortedness, they cannot cope with nested structural invariants such as the property C defined in Section 2. Fluid updates [9] allow a precise tracking of container properties, and analyze precisely operations such as a vector copy, but cannot capture nested structure properties. The analysis of [21] handles non-contiguous regions, and can compute abstractions of numerical constraints over such regions, but cannot infer a precise invariant like C, as it does not support inductive structure. On the other hand, significant progresses have been achieved in the analysis of programs with dynamic structures. Such works either use three-valued logic [23] or separation logic [22], and allow the verification of programs that manipulate dynamically linked data-structures such as variants of lists [5, 23] and trees [6]. However, these shape analyses cannot express that a structure lies inside an array, or a fixed contiguous space. Our work also extends the notion of abstraction parameterized by user supplied structure definitions of [6] to also deal with structures stored in arrays. A notable exception is [24], which extends a shape analysis with structures nested into abstractions of memory blocks. This work can only describe a single structure composed of all the cells in a non empty and contiguous region and will thus not capture the structures that we consider. By contrast, our analysis relies on coalescing of array and inductive predicates, which allows to simplify the associated inductive predicates and reason more effectively about more complex structures.

7 CONCLUSION AND PERSPECTIVES

We identify a pattern commonly encountered in embedded systems and OS code, that constructs and manages dynamic structures in an array, without an external memory allocator. While this pattern yields subtle code that is hard to verify and relies on sophisticated invariants, we propose an automatic static analysis to verify consistency invariants of such structures. It is parameterized by the structural consistency properties, and the value abstraction. The key idea behind it is to rely on the generic coalescing abstraction that we proposed to combine tightly abstractions for arrays and inductive structures. Experiments show that it successfully verifies different instances of this generic pattern in real-world low-level OS components and device drivers.

Coalescing views the underlying memory abstractions as black boxes, thus it opens the possibility to verify other kinds of nested structures. A promising direction is the verification of the implementation of programs such as a memory manager, a garbage collector, or a file system. In all these cases, the verification should handle several distinct levels of structures, that can be described by coalescing several specialized abstractions. To that end, one would need to extend our framework so as to support an arbitrary, possibly unbounded number of summaries stored inside a pool of cells.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Research Council under the European UnionâĂŹs seventh framework programme (FP7/2007-2013), grant agreement 278673, Project MemCAD, from the ARTEMIS Joint Undertaking no 269335 (see Article II.9 of the JU Grant Agreement). This work is also supported by the National Key R&D Program of China (No. 2017YFB1001802).

REFERENCES

[1] 2016. Eicon Driver. https://github.com/UDOOboard/Kernel_Unico/blob/master/drivers/isdn/hardware/eicon/io.c. (2016). Vers.: 2016-10-21.
[2] 2016. Nordic Semiconductor. https://github.com/ARMmbed/nrf5x-dfu-bootloader/blob/master/app_timer.c. (2016). Vers.: 2016-10-21.
[3] 2016. Timer Application. https://github.com/finnurtorfa/nrf51/blob/master/lib/nrf51sdk/Nordic/nrf51822/Board/nrf6310/ble/ble_app_gzll/ble_gzll_app_timer.c. (2016). Vers.: 2016-10-18.
[4] F. Alberti, S. Ghilardi, and N. Sharygina. 2014. Decision procedures for flat array properties. In *TACAS*.
[5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. 2007. Shape Analysis for Composite Data Structures. In *CAV*.
[6] E. Chang and X. Rival. 2008. Relational inductive shape analysis. In *POPL*.
[7] P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *POPL*.
[8] P. Cousot, R. Cousot, and F. Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*.
[9] I. Dillig, T. Dillig, and A. Aiken. 2010. Fluid updates: beyond strong vs. weak updates. In *ESOP*.
[10] Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. 2013. Local Shape Analysis for Overlaid Data Structures. In *SAS*. 150–171.
[11] D. Gay, P. Levis, J. von Behren, M. Welsh, E. Brewer, and D. Culler. 2003. The nesC language: a holistic approach to networked embedded systems. In *PLDI*.

[12] D. Gopan, T. Reps, and M. Sagiv. 2005. A framework for numeric analysis of array operations. In *POPL*.
[13] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. 2016. CertiKOS: an ext. architecture for building certified concurrent OS kernels. In *OSDI*.
[14] N. Halbwachs and M. Péron. 2008. Discovering properties about arrays in simple programs. In *PLDI*.
[15] B. Jeannet and A. Miné. 2009. Apron: a library of numerical abstract domains for static analysis. In *CAV*.
[16] R. Jhala and K. McMillan. 2007. Array abstraction from proofs. In *CAV*.
[17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*.
[18] L. Kovács and A. Voronkov. 2009. Finding loop invariants for programs over array using a theorem prover. In *FASE*.
[19] O. Lee, H. Yang, and R. Petersen. 2011. Program analysis for overlaid data structures. In *CAV*.
[20] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, and E. Brewer. 2005. Tinyos: An operating system for sensor networks. In *Ambient intelligence*.
[21] J. Liu and X. Rival. 2015. Abstraction of arrays based on non-contiguous partitions. In *VMCAI*.
[22] J. Reynolds. 2002. Separation Logic: a logic for shared mutable data structures. In *LICS*.
[23] M. Sagiv, T. Reps, and R. Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *TOPLAS* 24, 3 (2002).
[24] P. Sotin and X. Rival. 2012. Hierarchical shape abstraction of dynamic structures in static blocks. In *APLAS*.
[25] A. Tanenbaum and A. Woodhull. 1987. *OSes: design and implementation*. Vol. 2.
[26] A. Toubhans, E. Chang, and X. Rival. 2013. Reduced product combination of abstract domains for shapes. In *VMCAI*.
[27] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'hearn. 2008. Scalable shape analysis for systems code. In *CAV*.