User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring

Markus Weninger
Institute for System Software
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
markus.weninger@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Software becomes more and more complex. Performance degradations and anomalies can often only be understood by using monitoring approaches, e.g., for tracing the allocations and lifetimes of objects on the heap. However, this leads to huge amounts of data that have to be classified, grouped and visualized in order to be useful for developers. In this paper, we present a flexible offline memory analysis approach that allows classifying heap objects based on arbitrary criteria. A small set of predefined classification criteria such as the type and the allocation site of an object can further be extended by additional user-defined criteria. In contrast to state-of-the-art tools, which group objects based on a single criterion, our approach allows the combination of multiple criteria using multi-level grouping. The resulting classification trees allow a flexible in-depth analysis of the data and a natural hierarchical visualization of the results.

KEYWORDS

Memory, Monitoring, Analysis, Tool, Grouping, Classification

ACM Reference Format:
Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184412

1 INTRODUCTION

The increasing complexity of software systems requires tools and techniques for monitoring the behavior of large and complex applications. Many of these tools trace an application by recording events at run time and writing them to a trace file for later analysis. For example, a memory monitoring tool could record object allocations and garbage collector activity (e.g., object moves) so that the application's heap can be later reconstructed offline for various analyses.

Such monitoring tools produce huge amounts of data, which have to be classified, grouped and visualized in order to be helpful for the user. For example, users might want to know how many objects of a certain type were allocated, at which locations they were allocated, and how long they survived. Unfortunately, many state-of-the-art tools fail to provide a flexible information retrieval technique. Most of them only support hard-coded classification criteria (often *type* is the only one) in conjunction with tabular histograms, e.g., showing the number of instances per class and the number of allocated bytes. They don't allow users to classify the data based on multiple criteria (e.g., type, allocation site and age) and miss features to organize and aggregate the resulting information hierarchically on multiple levels.

Our tool AntTracks [12, 13] is a memory monitoring tool for Java based on the Java Hotspot™ VM [21] that records object allocations and garbage collection moves. It also offers offline analysis of trace files, in which the heap can be reconstructed for any garbage collection point in time. Bitto et al. [3] showed how to reconstruct an application's heap from traces produced by AntTracks. Based on this work, Weninger et al. [25] presented first ideas on object classifiers with the goal to make the classification of memory monitoring data more general and customizable.

In this paper, we extend our work by presenting a generally applicable object classification and multi-level grouping concept. An object classifier processes an object and classifies it based on a certain criterion derived from the object's properties, e.g., classifying heap objects based on their type. Objects with the same classification result are grouped together. As already mentioned, most state-of-the-art memory monitoring tools have two major restrictions: (1) They only offer a restricted set of classification criteria, such as *Type* or *Allocation Site*, and (2) their grouping mechanism is based on just a single classification criterion, i.e., single-level grouping. Our approach eliminates both restrictions. In addition to a set of predefined object classifiers that are usable out-of-the-box, users can define custom object classifiers as small dynamically-loaded code snippets. Furthermore, the grouping is not based on a single criterion but on dynamic classification trees, i.e., on multi-level grouping based on multiple object classifiers. Such classification trees store classification results in a hierarchical manner and allow a more flexible top-down data analysis approach. The concepts of object classification, multi-level grouping and classification trees are not restricted to memory data and may therefore also be used in other domains.

Our scientific contributions are (1) a novel concept of *object classifiers*, a way to classify a collection of objects based on their properties, (2) a multi-level grouping algorithm that classifies a collection of objects based on a user-chosen set of object classifiers into a *classification tree*, (3) various classification tree data structures

ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184412

that differ in terms of classification throughput, memory overhead and information loss, and (4) a quantitative evaluation based on well-known benchmarks as well as a functional evaluation based on typical memory analysis use cases.

2 BACKGROUND

AntTracks consists of a virtual machine based on the Java Hotspot™ VM and a memory analysis tool. The AntTracks VM records memory events into trace files, which can then be analyzed offline with the tool. Since our object classifier approach has been integrated into this tool, it is essential to understand AntTracks's architecture and workflow.

2.1 Trace Recording

The AntTracks VM records memory events, e.g., events for object allocations and object movements executed by the garbage collector (GC), throughout an application's execution and writes them into trace files. Furthermore, it is also capable of recording pointers between objects [11]. After loading such a trace file, the AntTracks analysis tool provides overview of the memory behavior over time and can reconstruct the heap's state and layout for every garbage collection point by incrementally processing the events in the trace.

2.2 Trace Reconstruction and Data Structure

Bitto et al. [3] show that a naïve approach, in which every heap object is represented by a Java object in the analysis tool, would result in an unacceptable memory overhead. Therefore, we developed the data structure shown in Figure 1. It separates the heap into multiple spaces. For example, the *ParallelOldGC*'s heap consists of one eden space, two survivor spaces, and one old space. Each of these spaces encompasses various fields such as the starting address, the size, or the kind of the space (i.e., *eden*, *survivor* or *old*). Additionally, each space contains an address-to-LAB map. A *LAB* (local allocation buffer) represents a sequence of objects that have been processed together by the same thread (e.g., objects that have been allocated by the same thread within the same thread-local allocation buffer (TLAB)). Each entry in the LAB's object array represents one heap object and contains a pointer to a global cache of object representations, called *ObjectInfo*. ObjectInfos are cached structures that contain information which is shared by multiple objects, namely the event which created the object (e.g., an allocation by the interpreter), the object's allocation site, its type and its size. For array allocations, also the array length is stored. Using this mechanism, many different objects can be represented by the same ObjectInfo. Their addresses do not have to be explicitly stored

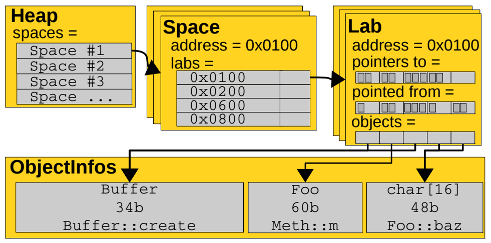


Figure 1: AntTracks's data structure to represent a heap at a certain point in time.

but can be computed from their LAB's address. In addition to the object array, each LAB contains two arrays of the same length to store pointer information. For each entry in the object array, i.e., for each heap object, the respective entry in the *pointers to* array contains the addresses of all objects that are referenced by this object. Analogously, each entry in the *pointed from* array contains the addresses of all objects that point to the respective object.

3 APPROACH

This section presents the domain-independent concepts of classification (i.e., representing an object by a classification result made up of one or more classification values) and multi-level grouping (i.e., arranging classification results in a tree structure). Examples on how these concepts can be applied in a specific domain / tool will be given in the context of Java and the classification of Java heap objects within the AntTracks memory analysis tool. If a specific heap state is shown, it has been reconstructed from a trace of a *DaCapo xalan* benchmark run.

3.1 Source Collection and Source Objects

Classification and grouping always operate on a *source collection* which consists of *source objects* of a certain type. AntTracks's source collection when classifying a heap state are the Java heap objects that have been live at the given point in time.

The source collection does not have to be represented by a single class but may be made up of multiple classes that interact with each other, see Figure 2. One of these classes must act as *the* source collection to the public. This class is required to provide functionality to iterate the contained source objects. In AntTracks, as explained in Section 2, a heap state is modeled by multiple classes (i.e., the heap itself, which further consists of multiple spaces, which further consist of multiple LABs), yet the `Heap` class acts as the source collection to the public.

Similarly, the properties of a source object do not have to be stored in a single object. In AntTracks, for example, they are stored in different locations: Most of them are stored in the `ObjectInfo`, but a heap object's pointers are stored in the LAB, and its address is calculated on demand.

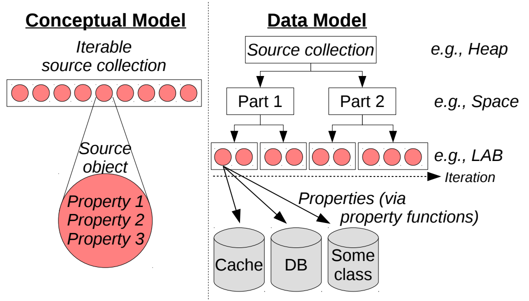


Figure 2: Basic classification concepts: Source collection, source objects and source object properties.

We distinguish the term *object* from the term *source object* because *object* is often used in the context of programming languages to describe a certain instance of a class. A source object, on the other hand, represents properties that may be stored in various places.

3.2 Source Object Properties and Source Collection Iteration

A source object is described by its m properties based on its position P within its source collection, as shown in Definition 3.1.

Definition 3.1. A source object at position P within its source collection is described by its m distributed properties:

$$sop \quad \text{is described by} \quad (prop_1, prop_2, \ldots, prop_m)_P$$

P’s format depends on the source collection. For example, in a list, source objects are identified by their index i, i.e., $P = i$. In AntTracks’s heap data structure, a source object’s position, i.e., the position of a heap object within the reconstructed heap, is described by (1) the space in which the object is, (2) the lab inside the object’s space, and (3) the object’s position within the lab, i.e., $P = (spaceIndex, labIndex, objectIndex)$.

Source collection iteration describes the task of visiting every position in the source collection and obtaining the properties of the respective source object. In AntTracks, iterating the heap means to visit every element in the `ObjectInfo` array of every LAB in every space, and collecting all properties of the currently visited heap object, e.g., calculating its address based on its containing LAB.

3.3 Object Classifiers

As soon as a source object’s properties have been obtained, object classifiers can be used to classify it. Object classifiers are entities that classify a source object based on a certain criterion derived from the source object’s properties. Each object classifier provides a `classify` function, which takes one parameter per source object property and returns the classification result. Additionally, every object classifier contains the following meta-data:

Name. A unique name used to identify the classifier.

Return Type. The `classify` method’s return type.

Description (Optional). Useful to keep the classifier’s names short while still offering additional information about the classifier’s purpose.

Example (Optional). A possible classification result returned by the classifier, e.g., `java.lang.Integer` returned by AntTracks’s *Type classifier*. This can be shown as a classification sample to the user in the UI.

Cardinality. Each classifier can be of one of the following three cardinalities: *One-to-one*, *one-to-many* or *one-to-hierarchy*. Depending on the cardinality, the classifier’s classification result may be made up of a different number of classification values, see Figure 3.

In AntTracks, object classifiers are used to classify Java heap objects based on their properties such as the object’s type, its allocation site and so on. Each classifier, e.g., the *Type classifier*, implements a common Java interface (most importantly the `classify` method), see Section 4.2.

3.3.1 One-to-one Classifier. A *one-to-one classifier* classifies a source object by a unique classification value as classification result (see top part in Figure 3). The returned classification value is an instance of the classifier’s return type, i.e., a one-to-one `String` classifier returns a single `String` as value.

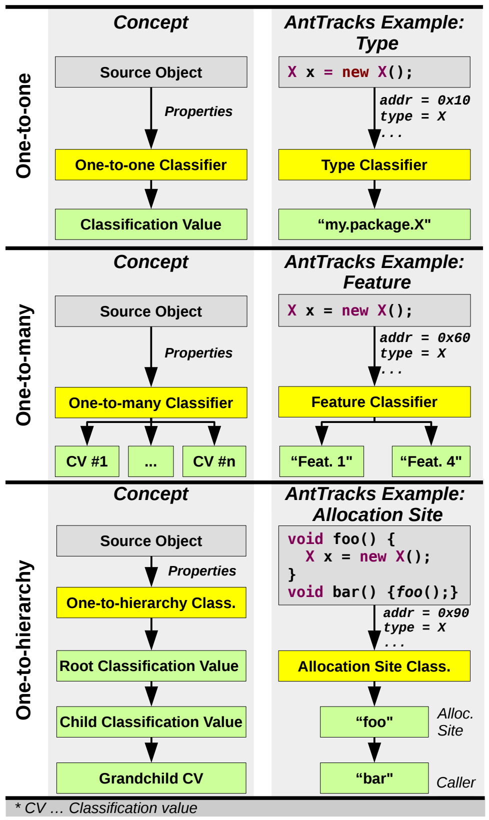
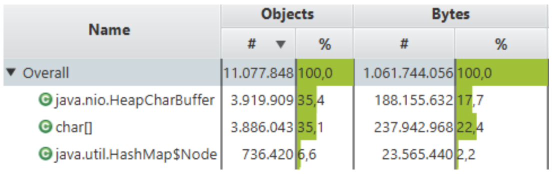


Figure 3: Object classifiers classify a source object based on its properties. The three types of classifiers vary in their classification value cardinality.

An example for a one-to-one classifier is AntTracks’s predefined *Type classifier*, which classifies a Java heap object based on its type’s name. Figure 4 shows a part of AntTracks’s analysis view where each heap object has been classified using the *Type classifier*. *Overall*



Name	Objects		Bytes	
	# ▼	%	#	%
▼ Overall	11.077.848	100,0	1.061.744.056	100,0
java.nio.HeapCharBuffer	3.919.909	35,4	188.155.632	17,7
char[]	3.886.043	35,1	237.942.968	22,4
java.util.HashMap$Node	736.420	6,6	23.565.440	2,2

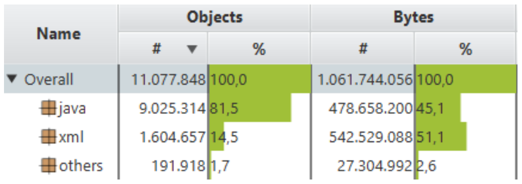
Figure 4: Classifying heap objects by type in AntTracks.

shows the number and byte count of the whole heap, and each child row represent a group of heap objects that have been classified by the same value, i.e., that are of the same type. Each heap object is part of exactly one group, i.e., *one-to-one* classification.

Filters. *Filters* are a special kind of *one-to-one classifiers*, which are of type `Boolean`. Filters are used in the classification process to define whether a source object should be further processed by subsequent operations.

3.3.2 One-to-many Classifier. A *one-to-many classifier* classifies a source object by multiple classification values, as can be seen in the middle part of Figure 3. The result is a set of instances of the classifier’s type: If the classifier’s type is `String`, a set of strings will be returned.

An example for such a classifier is the predefined *Feature classifier* in AntTracks. Assume that (possibly overlapping) code ranges represent specific features [10]. The allocation site of an object may then belong to one or more of these features. The *Feature classifier* performs a feature mapping for every Java heap object and returns the set of features to which its allocation site belongs. Figure 5,



Name	Objects		Bytes	
	# ▼	%	#	%
▼ Overall	11.077.848	100,0	1.061.744.056	100,0
java	9.025.314	81,5	478.658.200	45,1
xml	1.604.657	14,5	542.529.088	51,1
others	191.918	1,7	27.304.992	2,6

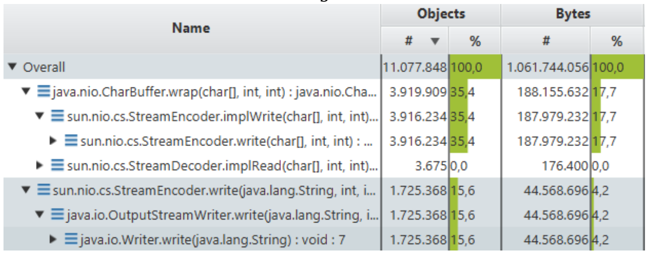
Figure 5: Classifying heap objects by feature in AntTracks.

similar to Figure 4, shows again a part of AntTracks's analysis view. This time, each heap object has been classified using the *Feature classifier*. Since the *Feature classifier* is a *one-to-many classifier*, each heap object can be part of multiple groups (if the classifier returned multiple values, i.e., features, for that heap object).

3.3.3 One-to-hierarchy Classifier. A *one-to-hierarchy classifier* classifies a source object by hierarchical classification values, as shown in the bottom part of Figure 3. Such a classifier returns objects of the classifier's return type in an ordered list. The object at index 0 is the root object, and for all $i > 0$ the object at index $i - 1$ is the parent of the object at index i.

An example for a *one-to-hierarchy classifier* is the predefined *Allocation Site classifier* in AntTracks, which classifies an object based on its allocation site and the allocation's call sites. The root object (at index 0) is the code location where the object was allocated, the object at index 1 is the code location from where the allocating method was called, and so on (i.e., the code location at index i is the callee and the code location at index $i + 1$ the caller). Figure 6

s



Name	Objects		Bytes	
	# ▼	%	#	%
▼ Overall	11.077.848	100,0	1.061.744.056	100,0
▼ java.nio.CharBuffer.wrap(char[], int, int) : java.nio.Cha...	3.919.909	35,4	188.155.632	17,7
▼ sun.nio.cs.StreamEncoder.implWrite(char[], int, int)...	3.916.234	35,4	187.979.232	17,7
▶ sun.nio.cs.StreamEncoder.write(char[], int, int) : ...	3.916.234	35,4	187.979.232	17,7
▶ sun.nio.cs.StreamDecoder.implRead(char[], int, int)...	3.675	0,0	176.400	0,0
▼ sun.nio.cs.StreamEncoder.write(java.lang.String, int, i...	1.725.368	15,6	44.568.696	4,2
▼ java.io.OutputStreamWriter.write(java.lang.String, i...	1.725.368	15,6	44.568.696	4,2
▶ java.io.Writer.write(java.lang.String) : void : 7	1.725.368	15,6	44.568.696	4,2

Figure 6: Classifying heap objects by allocation site in AntTracks.

also shows a part of AntTracks's analysis view similar to Figure 5, yet each heap object has been classified using the *Allocation Site classifier* instead. First-level children of the *Overall* group, i.e., row 2 and row 6, are allocation sites where objects have been allocated. Child relations represent the call chain, e.g., the call sites on row 3 and row 5 called the allocation site on row 2, and the call site on row 4 has been the single caller to the call site on row 3.

3.4 Multi-level Grouping

Single-level grouping splits a set of objects into multiple groups. Each group represents a distinct classification result (i.e., the classifier's return value) and contains all objects that are classified by this result. Typical single-level grouping only supports one-to-one classifiers, i.e., each object is mapped to exactly one classification value. In addition to introducing other classifier types beside one-to-one classifiers, we present multi-level grouping to enhance the flexibility and level of analysis detail.

3.4.1 Classification. Similar to single-level grouping, multi-level grouping is an operation that groups a set of source objects. Yet, instead of applying a single classifier, a list of classifiers is applied one after the other to every source object, and the sorted list of their classification results (where each classification result may be made up of multiple classification values) make up the source object's classification.

Obj.	Classification and results in parentheses
O(1)	`[Age(1) → Feat(F1, F2) → AS(add, A)]`
O(2)	`[Age(1) → Feat(F1, F2) → AS(add, B, D)]`
O(3)	`[Age(3) → Feat(F1) → AS(main, C, A)]`
O(4)	`[Age(3) → Feat(F1) → AS(clone, D)]`
O(5)	`[Age(3) → Feat(F1) → AS(main, C, A)]`
O(6)	`[Age(1) → Feat(F1, F2) → AS(add, A)]`

Table 1: Example classification of 6 Java heap objects based on three classifiers: Age (one-to-one), feature (one-to-many) and allocation site (one-to-hierarchy).

Table 1 shows an example classification for six objects $O(1)$ to $O(6)$. The three classifiers that get applied are (1) the *Age classifier*, a one-to-one classifier categorizing heap objects based on their number of survived GCs, (2) the *Feature classifier* (see Section 3.3.2) and (3) the *Allocation Site classifier* (see Section 3.3.3). Each classification contains three classification results, one per classifier, sorted in the order in which the classifiers were applied.

3.4.2 Classification Tree. Raw information as presented in Table 1 is not very helpful for the user. Classification trees bring such classification results into a hierarchical format that allows (1) flexible processing of data, such as merging, subgrouping, counting and so on as well as (2) straightforward visualization, e.g., as a tree table view, for user-driven analysis.

Figure 7 shows the creation of a classification tree for the objects in Table 1. Rectangles (yellow) represent tree nodes containing their keys as text, and arrows point to their child nodes. Smoothed rectangles (blue) represent the data that a node is holding, i.e., the source objects assigned to the node.

The following example explains how $O(1)$ gets added to the classification tree. The algorithm starts with the root node as the *current node*. During the classification process, when looking for a child node with a certain key that does not exist yet, a new child gets created for that key.

The *Age classifier* returns *1* as the classification result for $O(1)$. For each current node (i.e., the root node), the child matching this classification becomes the new current node, i.e. the status of current node moves from the parent to the child. Then, the *Feature classifier* is applied, which returns *F1* and *F2* as its classification values for the source object $O(1)$. Both features get added as children of *1* and become the new current nodes. Finally, the *Allocation Site classifier* gets applied on the source object and returns the allocation site *add* and its caller *A*. *add* nodes are appended as children

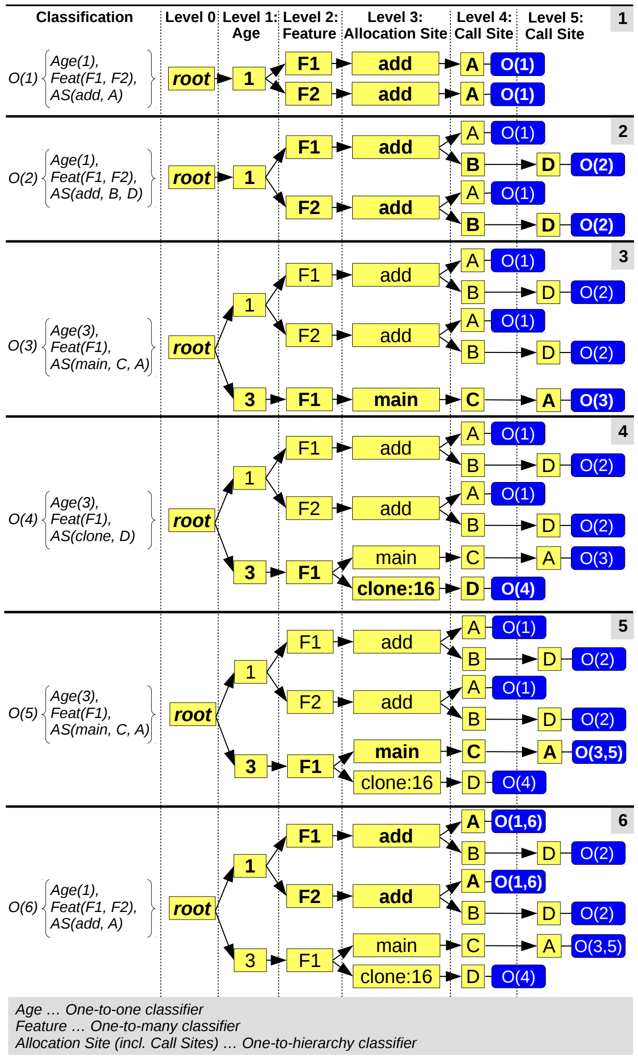

Classification
Level 0
Level 1: Age
Level 2: Feature
Level 3: Allocation Site
Level 4: Call Site
Level 5: Call Site
O(1) Age(1), Feat(F1, F2), AS(add, A)
O(2) Age(1), Feat(F1, F2), AS(add, B, D)
O(3) Age(3), Feat(F1), AS(main, C, A)
O(4) Age(3), Feat(F1), AS(clone, D)
O(5) Age(3), Feat(F1), AS(main, C, A)
O(6) Age(1), Feat(F1, F2), AS(add, A)
root
F1
F2
add
main
clone:16
O(1)
O(2)
O(3)
O(4)
O(3,5)
O(1,6)
Age … One-to-one classifier
Feature … One-to-many classifier
Allocation Site (incl. Call Sites) … One-to-hierarchy classifier

Figure 7: Step-by-step multi-level grouping of six heap objects into a classification tree based on age, feature and allocation site.

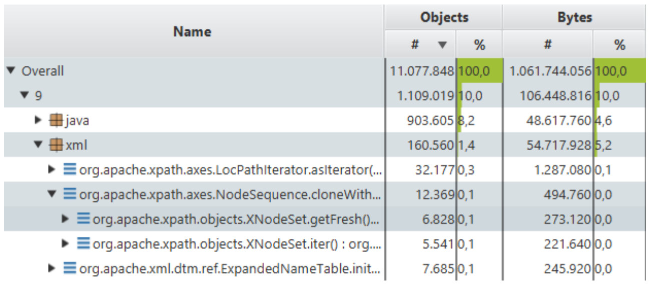
to all current nodes (i.e., to *F1* and *F2*) and *A* nodes are appended to the two *add* nodes.

Since no more classifiers have to be applied, the object is then added as a data entry at the current nodes, i.e., at both *A* nodes. This is the state that is shown in the top part of Figure 7. To reach the state at the bottom of Figure 7 the above steps are repeated for every source object $O(2)$ to $O(6)$.

Figure 8 shows an example on how classification trees get visualized in AntTracks. It displays a part of AntTracks's heap state analysis view where all heap objects have first been classified by *Age*, then by *Feature*, followed by *Allocation Site*.

3.5 Data Representation in Nodes

Source objects have to be associated with certain nodes of the classification tree. Various approaches are possible, some of which sacrifice information in favor of reduced memory overhead (see Figure 9).



Name	Objects		Bytes	
	# ▼	%	#	%
▼ Overall	11.077.848	100,0	1.061.744.056	100,0
▼ 9	1.109.019	10,0	106.448.816	10,0
▶ java	903.605	8,2	48.617.760	4,6
▼ xml	160.560	1,4	54.717.928	5,2
▶ org.apache.xpath.axes.LocPathIterator.asIterator(...	32.177	0,3	1.287.080	0,1
▼ org.apache.xpath.axes.NodeSequence.cloneWith...	12.369	0,1	494.760	0,0
▶ org.apache.xpath.objects.XNodeSet.getFresh()...	6.828	0,1	273.120	0,0
▶ org.apache.xpath.objects.XNodeSet.iter() : org....	5.541	0,1	221.640	0,0
▶ org.apache.xml.dtm.ref.ExpandedNameTable.init...	7.685	0,1	245.920	0,0

Figure 8: AntTracks's visualization of classification trees.

3.5.1 *Lossless Approaches.* Information lossless approaches allow to retrieve all properties of all source objects stored in the classification tree. This is needed if the classification tree should later be used for further complex processing.

Naïve List Approach. A naïve approach is to represent the node's data as a list of objects. A source object's properties (which are distributively stored) would have to be combined into a new object on demand (e.g., `new MyObject(p1, p2, p3)`).

We chose to store source object properties in a scattered way exactly because we want to *prevent* the creation of class instances, which would lead to increased memory footprint (e.g., due to object headers). Further, the more live objects reside in the heap, the less memory is available for new allocations. This results in more frequent GC invocations, which may slow down the application.

Property List Approach. Instead of storing a list of objects, this approach only stores a list of one of the source object's properties. This is possible if the object's remaining properties can be derived from this property, which is the case for nearly all use cases. In AntTracks, for example, heap objects can be identified by their address. The downside of this approach is the additional indirection when obtaining the other properties on demand.

3.5.2 *Lossy Approaches.* The lossless approaches retain object identity, i.e., we know exactly which source objects have been added to which tree nodes. This level of detail may be traded for less memory-consuming tree node data structures.

Mapping Approach. This approach relies on a map, where the key's type is application-dependent and the value is represented by a counter.

When adding a source object to a node, information of interest about the object gets extracted as the *object key*. This object key is

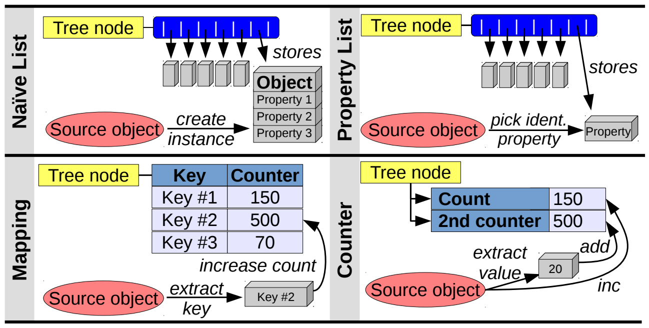


Figure 9: Two lossless list approaches and two lossy approaches based on counters to store node data.

then looked up in the node's data map and the respective counter gets incremented (or created if it does not yet exist).

It is crucial to take two aspects into account when choosing the object key: (1) What data should be reconstructed from the classification tree and (2) that many source objects should share the same object key to keep the number of entries in the map small. For example, AntTracks uses the object size (in bytes) as the object key. While this allows to only aggregate the number of objects and number of bytes represented by a certain node, it offers high memory saving potential which is discussed in more detail in Section 5. For example, if 1000 objects of only three different sizes get added to the same node, this approach just needs three key-value pairs compared to 1000 list entries as in the list approaches.

Counter Approach. This approach is designed to have the lowest memory footprint, while giving up flexibility and accepting the highest loss of information. Every time a source object gets classified at a certain node, counters stored in the node get incremented based on a fixed scheme. In AntTracks, for example, we could store two counters, one for the number of objects and one for the number of bytes classified at the given node.

This approach even loses information about specific properties. For example, it would not be possible to determine how many heap objects of a certain size have been classified, which is possible using the mapping approach.

3.6 Aggregation and Duplicate Detection

Using a one-to-many classifier may cause a source object to be added to multiple nodes. To avoid wrong results when aggregating this data, we have to detect duplicate entries in the tree and ignore them.

3.6.1 List Approaches. Since the entries in every data list are distinct, the lists can be treated as sets. The set of objects in a tree with head n can be computed recursively as the union of the objects in n and in the subtrees (Equation 1). Duplicates will be removed and the resulting set can be used for counting.

$$objects(n) = n.data \cup (\bigcup_{n.children}^{child} objects(child)) \qquad (1)$$

3.6.2 Mapping Approach. By extracting a source object's object key, we lose the object identity which would be needed for duplicate detection. Therefore, we additionally have to keep track of multiple classifications. This can be done by installing a second map, i.e., the *duplicate map*, in each node.

If a source object is added to more than one subtree of a node n, a counter for the object's key is incremented in the duplication map of node n, which is later used for sifting out duplicates when the total number of objects in a tree is computed.

3.6.3 Counter Approach. Similar to the mapping approach, every node could store a duplicate counter per data counter. In all situations where a duplicate counter in the mapping approach would be incremented, the duplicate counter in the counter approach is incremented.

3.7 Advanced Classifiers

For advanced use, a special kind of classifiers are *transformers*. So far, a classifier always took a source object's properties as its input and returned one or more classification values as classification result. A transformer takes a source object and (1) transforms it into a set of other source objects, (2) classifies each of these objects based on a selected set of object classifiers, and (3) multi-groups them based on their classification results (see Figure 10).

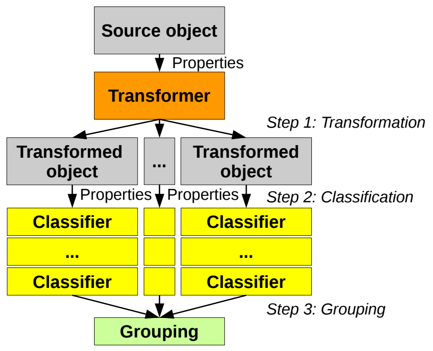


Figure 10: Transformers transform a source object into a set of other source objects, classify each of these objects and group them.

A use case for transformers in the domain of memory monitoring is pointer analysis. First, a heap object gets transformed into the set of all objects that are referenced by it. Second, this set of objects gets classified based on a list of other classifiers selected by the user. Finally, the classification results get multi-grouped into the resulting classification tree. For example, this can be used to analyze type-points-to-type graphs, as done by Jump and McKinley [8, 9].

4 IMPLEMENTATION

The previous section explained the domain-independent core concepts of classification and multi-level grouping based on object classifiers alongside some examples in the context of AntTracks. This section discusses some implementation details on how these concepts have been incorporated into AntTracks and its memory analysis.

Property	Additional info
address	
space	Space index, name, address, length, ...
type	Name, package, fields, ...
size	The object's size in bytes
isArray	true / false
arrayLength	-1 for non-arrays
allocationSite	Call stack, ...
pointedFrom	Addresses of all referencing objects
pointsTo	Addresses of all pointees
eventType	Allocation event (alloc. subsystem, ...)

Table 2: Source object properties for heap objects.

4.1 Source Objects: Java Heap Objects

AntTracks's source objects are Java heap objects that were alive in the monitored application at a given point in time, i.e., the heap objects that make up a certain heap state. Table 2 shows which properties make up a Java heap object in AntTracks, i.e., the source object properties. Every object classifier classifies a Java heap object based on a criterion derived from these properties.

4.2 Object Classifiers

In AntTracks, classifiers implement a common base interface. This interface defines the `classify` method, with its parameter signature matching the Java heap object properties.

To provide a convenient analysis environment for most use cases, AntTracks comprises multiple predefined object classifiers. These classifiers, listed in Table 3, can be used and combined freely on every heap state. An example implementation of the *Type classifier* can be seen in Listing 1.

Listing 1: Implementation of the *Type classifier* in AntTracks.

```
public class TypeClassifier implements Classifier<String> {
  // ... Fields modifiable by user, e.g., showPackage ...
  @Override public String classify(
    long address, Space space, Type type, long size,
    boolean isArray, int arrayLength, AllocationSite allocSite,
    long[] pointedFrom, long[] pointsTo, Event eventType) {
    return type.getName(showPackage);
  }
}
```

When a heap state is opened in AntTracks, a default classification (*Type classifier* followed by the *Allocation Site classifier*) gets applied. This gives a fast overview that shows which types have the most living objects, and where these objects have been allocated.

4.3 Heap Iteration

We implemented three different iteration approaches for AntTracks's heap data structure to evaluate their influence on the classification speed.

4.3.1 Java Streams. This approach has been implemented as a baseline for performance comparison. It uses the default technique for Java streams on custom data structures by implementing a `Spliterator`, the concurrent counterpart of an `Iterator`.

Java Stream Memory Overhead. The main problem with Java streams and spliterators is that they are generic classes working on Java objects of type `T`. Therefore, to support Java streams in AntTracks, we have to transform AntTracks's source objects (i.e., heap objects that are stored as scattered properties) into instances of an auxiliary `HeapObject` class. These short-living objects (which only exist while the stream is processed) may put unnecessary burden on the garbage collector, especially for large heap states.

4.3.2 Fake Spliterator. This approach relies on a custom iteration class that provides a `tryAdvance` and a `trySplit` method, similar to the `Spliterator` implemented for the Java stream approach. However, this *fake* spliterator does not inherit from Java's `Spliterator` interface, but only mimics its behavior. More specifically, the fake spliterator's `tryAdvance` does not match the official interface but has been changed in a way that allows the fake spliterator to process a heap object's properties separately, which has the advantage of avoiding the need for auxiliary objects.

4.3.3 Integrated Iteration Functions. A basic implementation of this approach already existed in the previous versions of AntTracks. It provided sequential iteration functions on each data structure level, i.e., on the `Heap`, the `Space`, and the `LAB`. In our approach, we added support for parallel iteration, which significantly increased performance.

4.4 User-defined Classifiers

Classification in AntTracks is not restricted to predefined classifiers, but allows users to define new classifiers, i.e., *user-defined classifiers*, in two different ways: (1) By using Java's Service Provider Interface (SPI) concept, where new classifiers can be added to AntTracks as pre-compiled JAR files, and (2) by using in-memory on-the-fly compilation to support classifier development at run time.

4.4.1 Service Provider Interfaces (SPI). A *service provider interface* is a set of public interfaces and abstract classes that a third-party developer can implement. In AntTracks, the SPI encompasses abstract classes for classifiers, transformers, and filters. All of them define an abstract `classify` method which can be implemented by third-party developers in a sub-class. If a JAR containing such an implementation is detected on AntTracks's class path (using convenient SPI methods), it will be added to the list of available classifiers or filters.

4.4.2 On-the-fly Compilation. It is also possible to define new object classifiers, transformers and filters at run time. For example, whenever users have to select one of the available classifiers, they are offered to define a new one. The user then has to provide the `classify` method, the classifier's name, description, example and cardinality. This information gets merged into an object classifier template file which will then be compiled with a modified Java compiler that enables compilation without generating a Java class file on disk, i.e., the classifier gets compiled in-memory and on-the-fly.

This compilation relies on the `JavaCompiler` instance returned by `ToolProvider.getSystemJavaCompiler()`. This instance allows modifying the compilation process in various ways. The most important step is to provide a modified `JavaFileManager`. Instead of providing a stream to a file on disk, AntTracks's version returns a `ByteArrayOutputStream` that keeps a class's byte code stored in memory. Additionally, the file manager's class loader has been modified to not only look up classes stored on disk, but also to look up classes that are stored in memory.

5 EVALUATION

To evaluate the applicability of AntTracks's object classifiers and multi-level grouping we show how one can use the tool to detect memory leaks and how to reproduce memory classification done in related work.

Even though lossless classification tree implementations may be needed in certain situations, a lossy approach provides enough information for most use cases, including AntTracks's heap state analysis. Therefore, another goal of this evaluation is to analyze how much classification throughput can be gained as well as how much memory can be saved by accepting the information loss due to using a lossy classification tree implementation. All of these

Name	Description
Address	Classifies objects based on their address.
Type	Classifies objects based on their type's name.
Allocating Subsystem	Either *VM*, *Interpreter*, *C1-compiled code* or *C2-compiled code*.
Array Length	Classifies array objects based on their length. Non-array objects are classified as -1.
Object Kind	Either *Instance* (class instances), *Small Array* (< 255 elements), or *Big Array* (≥ 255 elements).
Space	Classifies objects based on the heap space in which they are contained.
Space Mode	Classifies objects based on the mode, i.e., a GC-dependent space info, of their containing heap space.
Space Type	Classifies objects based on the type (e.g., *Eden*) of the space in which they are contained.
Feature	Classifies objects based on a loaded feature-to-code mapping file.
Allocation Site	Classifies objects based on their allocation site (allocating method + var. number of call sites).
Pointed From	This transformer is used to classify the objects that reference a given object.
Points To	This transformer is used to classify the objects that a given object references.

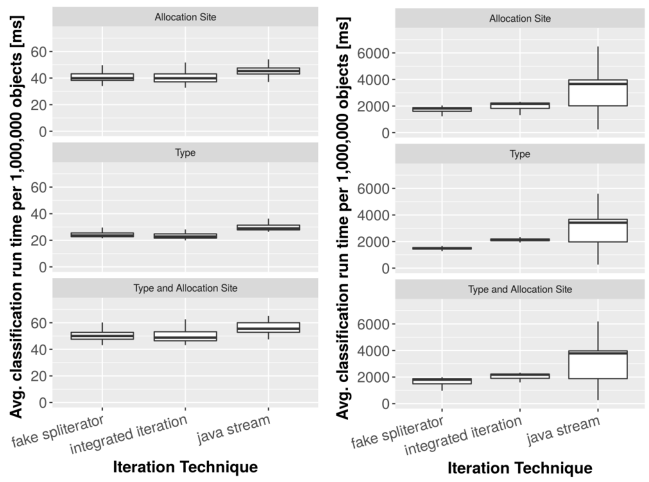
Table 3: Predefined classifiers in AntTracks.

analyses have been conducted based on well-known benchmarks using three different classifier combinations: (1) Type classifier (2) Allocation Site classifier (3) Type classifier, followed by the Allocation Site classifier.

Setup. All measurements were run on an Intel® Core™ i7-4790K CPU @ 4.00GHz x 4 (8 Threads) on 64-bit with 32 GB RAM and a Samsung SSD 850, running Ubuntu 17.10 with the Kernel Linux 4.13.0-16-generic. All unnecessary services were disabled in order not to distort the experiments.

5.1 Performance Evaluation

The goal of this evaluation is to gain insight into how much the classification throughput increases when giving up object identity and if Java streams are suitable to iterate distributed source objects. Thus, we compare both implemented tree node types (i.e., the property list approach (lossless) and the mapping approach (lossy)) using three different parallel heap iteration techniques (i.e., Java stream, fake spliterator and integrated iteration).



(a) Lossy mapping approach **(b) Lossless property list approach**

Figure 11: Performance comparison between the mapping approach and the property list approach.

We used the DaCapo [4] and the DaCapo Scala [6] benchmark suites, in which, according to Lengauer at el. [14], h2 and factorie are the benchmarks with the largest live set. We chose to only analyze these two benchmarks since the other benchmarks from the mentioned suites do not provide heap states in the same dimension. Both trace files (h2: 2.9 GB trace file covering 26 garbage collections with 15,800,000 objects on average per heap state; factorie: 19.5 GB trace file covering 205 GCs with 8,600,000 objects on average per heap state) have been parsed and a classification tree has been generated at every garbage collection end using every parameter combination (i.e., iteration type, classifier, tree type).

Figure 11a shows the average throughput of this classification tree generation when using the lossy mapping approach, while Figure 11b shows the throughput using the property list approach. We can see that the mapping approach is orders of magnitude faster than the property list approach due to the work that is needed to add the object's address to the sorted data list when using the property list approach. This strengthens our assumption to use the mapping approach when object-identity loss is acceptable.

Furthermore, it shows that heap iteration using Java streams is in general slower than the other two approaches. Especially for larger heap states, the streaming approach falls behind the other approaches. As hypothesized, this may be due to the temporary objects that have to be generated during the iteration. Independent of the domain this indicates that Java streams are not suitable for iterating distributively stored source objects. The fake spliterator approach is able to scale and parallelize the best, which explains its advantage when using the property list approach.

5.2 Memory Footprint

Beside providing the better classification performance, it is interesting to see how much memory can be saved when using the object-identity-losing mapping approach instead of the property list approach.

We analyzed a traced run of every DaCapo and DaCapo Scala benchmark and reconstructed the heap state after every garbage collection, if the heap state contained at least 200,000 objects. The *Type classifier* showed that the number of types of live objects at a certain point in time is approximately the same across all benchmarks (around 500 objects), independent of the number of live objects. Some of the benchmarks have few live objects with a high number of different allocation site nodes (i.e., few objects allocated at different sites) while some benchmarks with a large number of live objects only generate a small number of allocation site nodes (i.e., a lot of objects allocated at the same sites). Nevertheless, the tree never reached a critical size in terms of node count for any of the tested applications (tree size always below 20,000 nodes).

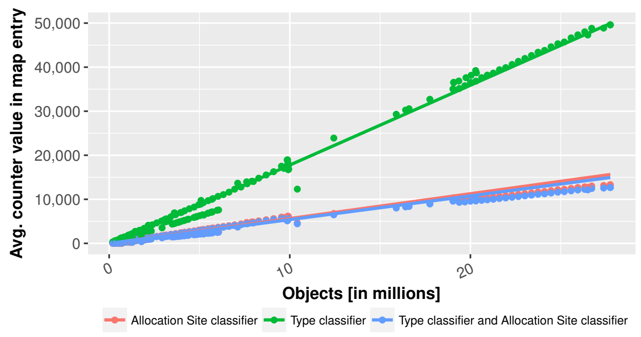


Figure 12: Average object count per data map entry using the mapping approach.

Figure 12 shows that with a rising number of classified objects, the average number of objects represented by a single data map entry in the mapping approach increases. For example, classifying about 10, 000, 000 objects based on the `Type classifier` resulted in data map entries each representing about 18, 000 objects on average (see regression line in Figure 12). Assume that the property list approach is implemented using arrays and needs 8 bytes per classified object (i.e., the heap object's 64-bit address excluding memory needed by auxiliary data structures). Let's further assume that each map entry in the mapping approach points to a key (containing an `int`) and a value (containing a `long`), thus taking up $3*16$ (3∗VM header)+$2*8$ (2∗pointer)+4 (int)+8 (long) = 76 bytes. If one such data map entry represents 18, 000 objects, the property list approach ($8 * 18,000$ bytes) consumes about 1900 times as much memory as the mapping approach (76 bytes).

Based on these results and those presented in Section 5.1, we decided to use classification tree generation based on fake spliterator heap iteration and the mapping approach in AntTracks.

The next section shows that the lossy mapping approach still provides enough information to detect memory leaks and allows general memory analysis.

5.3 Functional Evaluation

AntTracks's goal is to provide a general memory monitoring and analysis tool that primarily focuses on developers and their needs, for example performing memory leak detection. In addition, user-defined classifiers, their flexible combination, and multi-level grouping allows developers and also researchers to use AntTracks for more general and experimental memory analyses.

5.3.1 Memory Leak Detection. Memory leak detection is the main task developers perform when using AntTracks. To evaluate AntTracks's ability to allow memory leak detection, as well as finding the root cause, we used it on an example artificial application that uses a stack[1] for storing its data. It first pushes 1 million objects onto the stack, then pops these 1 million objects, followed by another 100,000 pushes and another 100,000 pop operations. Opening the application's trace displays the overview shown in Figure 13. We can clearly see that we miss a drop of the number of live objects after the 1 million objects got popped from

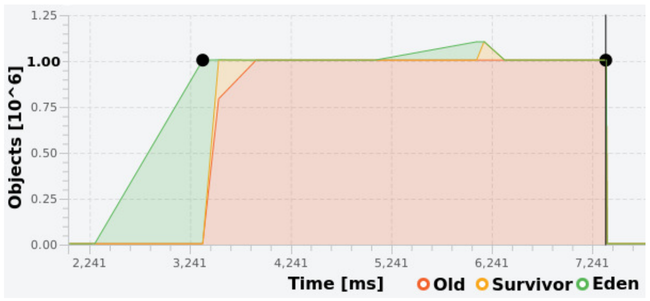
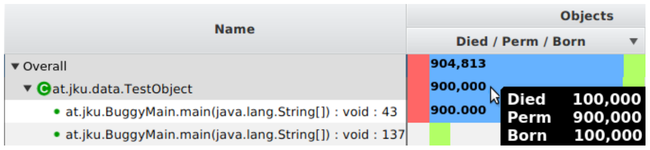


Figure 13: Object count overview of the buggy stack implementation.

the stack, as we would expect in a non-faulty implementation. To further investigate this problem, we utilized AntTracks's heap diffing functionality, which also supports object classifiers and allows to analyze heap changes over time. Figure 14 shows the application

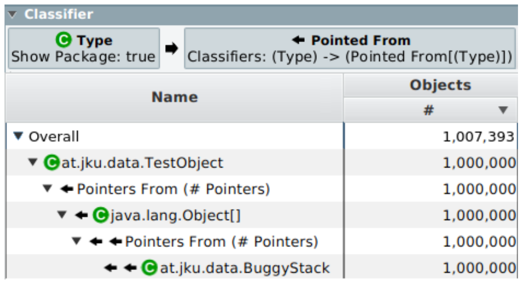


Name	Objects Died / Perm / Born
▼ Overall	904,813
▼ at.jku.data.TestObject	900,000
• at.jku.BuggyMain.main(java.lang.String[]) : void : 43	900.000
• at.jku.BuggyMain.main(java.lang.String[]) : void : 137	

Died	100,000
Perm	900,000
Born	100,000

Figure 14: Heap diff of the buggy stack implementation.

of the *Type classifier* followed by the *Allocation Site classifier* on the time frame selected in Figure 13 (black dots). On the type node (2nd row, `at.jku.data.TestObject`), we can see that only 100.000 objects of this type were deallocated (red bar), while exactly the same amount of objects were allocated (green bar). 900, 000 objects stayed alive during the whole time frame (blue bar). Looking at the indented allocation site nodes (3rd and 4th row), we see how many `TestObjects` that were originally allocated at these sites were born, have survived, or have died.



Classifier

Type
Show Package: true

Pointed From
Classifiers: (Type) -> (Pointed From[(Type)])

Name	Objects #
▼ Overall	1,007,393
▼ at.jku.data.TestObject	1,000,000
▼ Pointers From (# Pointers)	1,000,000
▼ java.lang.Object[]	1,000,000
▼ Pointers From (# Pointers)	1,000,000
at.jku.data.BuggyStack	1,000,000

Figure 15: Pointer analysis of the buggy stack implementation.

Additionally, we would like to know which objects keep those objects alive. Figure 15 shows a rather advanced application of object classifiers: It first classifies a given object by its type, then transforms that object into its set of referencing objects, classifies them by type and then transforms them again into their sets of referencing objects, finally classifying those objects by type. It shows that the `TestObject` instances are referenced from the type `Object[]`, which is again referenced by the type `BuggyStack`. With this information, it is easy to find the bug in the source code. `BuggyStack` is a faulty stack implementation that keeps references to previously

[1]https://www.codeproject.com/Articles/30593/Effective-Java; Item 6: Eliminate obsolete object references; last accessed October 17, 2017

stored objects even after pop operations until a subsequent push operation overwrites them.

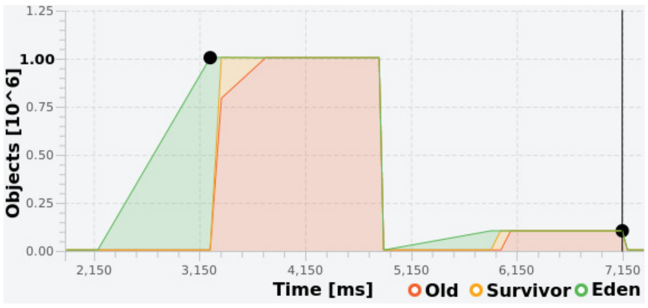
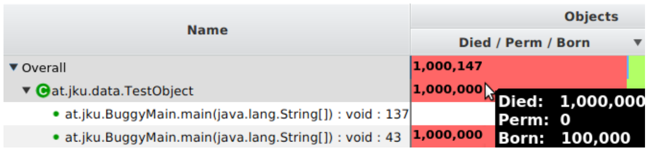


Figure 16: Object count overview of the fixed application.



Name	Objects Died / Perm / Born
▼ Overall	1,000,147
▼ at.jku.data.TestObject	1,000,000
• at.jku.BuggyMain.main(java.lang.String[]) : void : 137	
• at.jku.BuggyMain.main(java.lang.String[]) : void : 43	1,000,000

```
Died:  1,000,000
Perm:  0
Born:  100,000
```

Figure 17: Heap diff of the fixed application.

Figure 16 and Figure 17 show the object counts and the heap diff results after the stack implementation was fixed.

5.3.2 Memory Analysis. Developers as well as researchers may want to classify heap objects based on a criterion not yet covered by one of the predefined classifiers, which is possible by writing a *user-defined* classifier. To showcase the implementation of *user-defined* classifiers, we searched for related work on heap object classification. For example, Mitchell and Sevitsky [17] classified heap objects in terms of *collection health* and *instance health*. Both classification criteria have been successfully implemented as *user-defined* classifiers and can be used and freely combined with other classifiers in AntTracks.

Collection Health. Collection health classifies every heap object as one of four types, depending on its use inside collections. (1) *head*, the head of a collection, e.g., `HashMap`, (2) *array*, array backbones, e.g., `HashMap$Entry[]`, (3) *entry*, recursive list-style elements, e.g., `HashMap$Entry`, and (4) *contained*, anything else.

The classification for collection health is a typical use-case for a user-defined one-to-one object classifier. Every object gets classified by exactly one value, i.e., either *head*, *array*, *entry* or *contained*. According to Mitchell and Sevitsky, every object that is an array of a reference type gets classified as *array*. This is straightforward to check in the classifier implementation[2] since we know the object's type. If an object is not classified as *array*, it falls in the *entry* category if it is of a type `T` and references an object of the same type `T`. This check can be accomplished by following and analyzing the pointers in the object's pointer array. If the object has not been categorized as *array* or *entry*, the object's pointers are checked again. If one of them references an object that is a primitive array or is classified as *array* or *entry*, the object gets classified as *head*. Otherwise, the object gets classified as *contained*.

[2]http://ssw.jku.at/General/Staff/Weninger/AntTracks/ICPE18/CollectionHealthClassifier.java

Instance Health. Instance health splits every heap object's bytes into four different parts: (1) *primitive*, which encompasses primitive array elements and primitive fields (2) *header*, the memory consumed by the virtual machine (3) *pointer*, memory occupied by references between objects (4) *null*, memory reserved for pointers but set to null.

The classification for instance health has been reproduced as a user-defined transformer in AntTracks[3]. The source object gets transformed into four *virtual objects*, one per instance health part, and every part gets assigned its appropriate size (i.e., byte count). The amount of bytes of the *primitive* part can be calculated by iterating the type's fields and filtering them for primitive types. The information about the *header* size (which depends on the VM architecture, as well as whether compressed oops are used) is stored in the symbols information generated alongside the trace file. Since an object's pointer array contains one entry per pointer, either with the referenced object's address or -1 if the pointer is null, the bytes made up by *pointers* and *null* can also be easily calculated.

The judgment schemes presented by Mitchell and Sevitsky, i.e., the ways how to interpret combinations of both classifiers, can now also be analyzed in AntTracks by using both classifiers at the same time. Furthermore, they can be used in combination with any other classifier that AntTracks provides.

Mitchell and Sevitsky used *"the built-in facilities of Java virtual machines (JVM) to trigger writing a snapshot to disk"* [17]. Before being able to write an analysis tool for such heap snapshots, one must obtain knowledge about the binary file format, how to parse it, and how to combine the parsed data into a convenient data structure. Depending on the use-case, results also have to be presented graphically to the user to allow user-friendly manual analysis, which also may take up a significant amount of development time. Compared to that, the implementation of the two classifiers presented above took about two hours each, including writing unit tests (by checking the correct classification of known Java classes such as `HashMap`). The `classify` methods of both classifiers cover less than 150 lines of code (LOC). Therefore, we claim that writing *user-defined classifiers* takes less work, with regard to person hours as well as LOC. Additionally, AntTracks provides convenient visualization out-of-the-box and the possibility to combine the newly developed classifier with any other available classifier.

6 RELATED WORK

Current state-of-the-art tools share one common problem. Nearly all of them represent heap states (or the change of the heap over time) only as type histograms. No free selection of classification exists, not even to mention multi-level grouping. Even basic information such as an object's allocation site is not available in many cases, since most tools rely on heap dumps that do not provide that level of detail. Still, some tools provide additional functionality such as pointer information on object level (plainly reconstructed from a heap dump).

The most basic approach supported by the Java Hotspot VM are the `-XX:+PrintClassHistogramBeforeFullGC` and `-XX:+PrintClassHistogramAfterFullGC` flags. They cause a class

[3]http://ssw.jku.at/General/Staff/Weninger/AntTracks/ICPE18/InstanceHealthClassifier.java

histogram to be printed to the console on every full GC. *JConsole* [18] can connect to a running Java application and retrieves data from its Java Management Beans. Due to the restricted functionality of the memory bean, it can only show the current heap memory consumption separated into eden space, survivor space, and old space. *jhat* [19] can be used to analyze a Java heap dump file which has previously been generated using the *jmap* tool. It starts a webserver that hosts the heap dump results and can be accessed via a webbrowser. Beside a type histogram, also the rootset (i.e., objects that are referenced by a GC root) can be shown. *Visual VM* [23] is a general performance monitoring tool for Java applications that provides memory analysis based on heap dumps. In addition to a type histogram, it allows users to analyze individual objects of a certain type, including functionality to follow an object's pointers and go to the referencing object. It is also able to calculate the retained set of objects. The retained set of an object X is the set of objects which would be removed when X is garbage collected. In addition to that, the *Eclipse Memory Analyzer (MAT)* [7] also allows users to analyze the application's dominator tree [15]. The *Netbeans profiler* [20] is just a slimmed down version of Visual VM and is integrated into the Netbeans IDE.

Other approaches such as the one presented by Aftandilian et al. [1] or De Pauw and Wim [22] focus on visualizing a heap state's object graph. To reduce the complexity of such graphs, certain reduction operations such merging, cutting, and so on, are applied. Such approaches may work well for pointer analysis, e.g., which types references which types, yet most of them lack the flexibility to take other properties into account, e.g., heap spaces or allocation sites.

A query technique that is integrated into some of the mentioned tools is the *Object Query Language (OQL)* [2, 5]. It has been developed by the Object Data Management Group and is an SQL-like query language used to query objects from object-oriented databases. The downside of OQL is its complexity, which results in the problem that no vendor implements the whole standard. For example, the Eclipse Memory Analyzer (MAT) as well as VisualVM only allow queries in the form of `SELECT <select clause> FROM <from clause> WHERE <where clause>`. *Where clauses* can be represented in our approach using filters, while *select clauses* can be represented using an object classifier. Multi-level grouping, as supported in our approach, is neither possible in MAT nor in VisualVM.

7 FUTURE WORK

The concept of object classifiers and multi-level grouping as well as their implementation in AntTracks opened a number of interesting ideas. This section will shortly introduce these ideas and point out possible ways how to approach them.

Extended Pointer Support in AntTracks. Currently, Ant-Tracks provides only basic support for pointer analysis. For every object, it records the referencing and the referenced objects and makes them available for offline analysis. However, state-of-the-art tools [1, 16] often use advanced data structures such as dominator trees for analyzing whole pointer graphs. We plan to use similar data structures also in AntTracks to compute, for example, all objects that are reachable from a certain object (i.e., the transitive closure [24]) as well as the amount of memory that is kept alive by a specific object (i.e., the retained size).

Heap Diffing. Weninger et. al. [25] suggest heap diffing, i.e., analyzing how the heap changes over a certain time span, which is currently already supported to a certain level in AntTracks. The grouping and classification techniques that were described for heap states in this paper can partially also be applied to heap diffing. Extending classifiers with information about a source object's development over time, e.g., how a heap object's pointers changed over time, could further increase the potential application of heap diffing in combination with object classifiers.

Combined Tree Types. We showed that the memory consumption of a lossless classification tree is orders of magnitude higher than that of a lossy one. In a classification tree, often only a small subtree is of interest to the user. Since both classification tree types use node data structures inheriting from the same interface, they could be combined to only give lossless information for parts of the tree that are of higher interest to the user.

AntTracks DSL. To abstract from classifiers and their underlying programming language, the heap could also be analyzed by using a domain-specific query language. Such a language could, for example, be used to ask for the amount of objects of type T that were allocated at site S and survived at least n garbage collections. Based on our classifiers, we plan to develop such a language to provide even better support for expressing application-specific queries in a user-friendly way.

8 THREATS TO VALIDITY AND LIMITATIONS

Visualization of data in memory analysis tools is often strongly coupled with the kind of data that is collected and analyzed by those tools. Even though AntTracks collects more information about objects than most of the presented tools (e.g., only few tools collect allocation site information), the general classification principles using multi-level grouping and classification trees based on object classifiers and as well as AntTracks visualization features are not dependent on that amount of information. Only the number and the complexity of the classifiers that developers can implement is limited by the available information. The fewer source object properties are available, i.e., the less information the tool collects about heap objects, the less flexibility the developer has when it comes to writing classifiers. Assuming that AntTracks only collected type and heap space information for each object, we would still be able to provide the *Type classifier*, the *Object Kind classifier*, the *Space classifier* and so on as predefined classifiers, but due to the missing information, no *Allocation Site classifier* could be provided. Yet, all the available classifiers could still be freely combined, for example, by first classifying all objects by space and then by type, or first by object kind and then by space, or in any other possible combination. This outclasses the flexibility of the data aggregation and visualization techniques available in other tools presented in Section 6.

Similar to the limitation mentioned above, current pointer-based classifiers are restricted to adjacent objects via the from-pointer and to-pointer information. As explained in Section 7, new classifiers may become possible as soon as AntTracks provides full object graph traversal and root pointer information.

To verify that the extra flexibility simplifies memory analysis, specifically that it facilitates detecting and resolving memory-related problems such as memory leaks, a user study is planned as future work. Technical metrics such as *task completion time* or *number of found memory leaks* and subjective metrics such as *user satisfaction* can be collected during the study, based on faulty benchmark implementations or industry applications.

A limitation of our current study is that we have not yet investigated, which combinations of classifiers are best for detecting specific memory-related problems. This is another topic to be tackled by the mentioned user study.

9 CONCLUSION

In this paper, we presented the domain-independent concepts of (user-defined) object classifiers and multi-level grouping, which are novel and general concepts for classifying large amounts of objects, processing them, and arranging their classification results as a tree for later analysis. Object classifiers are entities that classify objects based on a certain criterion derived from the objects' properties. Multi-level grouping is the process of applying multiple object classifiers to a collection of objects and grouping these objects based on the classification results. In contrast to single-level grouping, which results in a key-value map, multi-level grouping results in a classification tree. Such a tree can be visualized in various ways and allows a top-down, fine-grained manual data analysis by the user.

Various lossless and lossy *classification tree* data structures were presented and analyzed with respect to their performance, their memory consumption, and their ability to retain object identity. We showed that the lossy tree structures allow a tremendous reduction of memory overhead when accepting certain information loss in the classification tree.

We integrated the concept of object classifiers and multi-level grouping into the memory monitoring tool AntTracks, a tool that primarily focuses on helping developers to detect and understand memory anomalies, thus replacing its previous rigid classification scheme. Developers benefit from AntTracks's new ability to classify heap states based on any combination of classifiers, which distinguishes our approach from existing state-of-the-art tools. Furthermore, our tool supports *user-defined* object classifiers, i.e, it allows the user to write small, dynamically loaded source code snippets to classify heap objects based on arbitrary criteria. This may also be of interest to researchers who want to perform more general and experimental memory analyses. Our memory analysis approach opens new ways how AntTracks can be used and how memory can be analyzed, and its applicability has been shown in a quantitative and a functional evaluation.

ACKNOWLEDGMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Dynatrace Austria GmbH.

REFERENCES

[1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proc. of the 5th Int'l. Symposium on Software Visualization (SOFTVIS '10)*. 53–62.

[2] A. M. Alashqur, S. Y. W. Su, and H. Lam. 1989. OQL: A Query Language for Manipulating Object-oriented Databases. In *Proc. of the 15th Int'l. Conference on Very Large Data Bases (VLDB '89)*. 433–442.

[3] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. 76–89.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proc. of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA '06)*. 169–190.

[5] R.G.G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. 2000. *The Object Data Standard: ODMG 3.0.*

[6] Technische Universität Darmstadt. 2012. DaCapoScala (last accessed October 10, 2017). http://www.benchmarks.scalabench.org/modules/scala-benchmark-suite/. (2012).

[7] Andrew Johnson and Krum Tsvetkov. 2017. MAT - Eclipse Memory Analyzer (last accessed October 10, 2017). http://www.eclipse.org/mat/. (2017).

[8] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. 31–38.

[9] Maria Jump and Kathryn S. McKinley. 2009. Dynamic Shape Analysis via Degree Metrics. In *Proc. of the 2009 Int'l. Symposium on Memory Management (ISMM '09)*. 119–128.

[10] Philipp Lengauer, Verena Bitto, Florian Angerer, Paul Grünbacher, and Hanspeter Mössenböck. 2013. Where Has All My Memory Gone?: Determining Memory Characteristics of Product Variants Using Virtual-machine-level Monitoring. In *Proc. of the Eighth Int'l. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '14)*. Article 13, 13:1–13:8 pages.

[11] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.

[12] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conference on Performance Engineering (ICPE '15)*. 51–62.

[13] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '16)*. 249–260.

[14] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proc. of the 8th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '17)*. 3–14.

[15] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 121–141.

[16] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks Using Graph Mining on Heap Dumps. In *Proc. of the 16th ACM SIGKDD Int'l. Conference on Knowledge Discovery and Data Mining (KDD '10)*.

[17] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proc. of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. 245–260.

[18] Oracle. 2017. JConsole (last accessed October 10, 2017). https://docs.oracle.com/javase/9/troubleshoot/diagnostic-tools.htm#JSTGD174. (2017).

[19] Oracle. 2017. jhat (last accessed October 10, 2017). https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jhat.html. (2017).

[20] Oracle. 2017. Netbeans profiler (last accessed October 10, 2017). https://profiler.netbeans.org/. (2017).

[21] Oracle. 2017. OpenJDK HotSpot group (last accessed October 22, 2017). (2017).

[22] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proceedings of the 13th European Conf. on Object-Oriented Programming (ECOOP '99)*. 116–134.

[23] Jiri Sedlacek and Tomas Hurka. 2017. Visual VM - All-in-One Java Troubleshooting Tool (last accessed October 10, 2017). https://visualvm.github.io/. (2017).

[24] R. Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 114–121. https://doi.org/10.1109/SWAT.1971.10

[25] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '17)*. 357–360.