Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection

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

Elias Gander
CD Laboratory MEVSS,
Johannes Kepler University
Linz, Austria
elias.gander@jku.at

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

ABSTRACT

Memory leaks are a major threat in modern software systems. They occur if objects are unintentionally kept alive longer than necessary and are often indicated by continuously growing data structures.

While there are various state-of-the-art memory monitoring tools, most of them share two critical shortcomings: (1) They have no knowledge about the monitored application's data structures and (2) they support no or only rudimentary analysis of the application's data structures over time.

This paper encompasses novel techniques to tackle both of these drawbacks. It presents a domain-specific language (DSL) that allows users to describe arbitrary data structures, as well as an algorithm to detect instances of these data structures in reconstructed heaps. In addition, we propose techniques and metrics to analyze and measure the evolution of data structure instances over time. This allows us to identify those instances that are most likely involved in a memory leak. These concepts have been integrated into AntTracks, a trace-based memory monitoring tool. We present our approach to detect memory leaks in several real-world applications, showing its applicability and feasibility.

CCS CONCEPTS

• **General and reference → Metrics**; *Performance*; • **Information systems → Data structures**; • **Software and its engineering → Data types and structures**; **Domain specific languages**; *Dynamic analysis*; *Software performance*; Garbage collection.

KEYWORDS

Memory Monitoring, Data Structures, Growth Analysis, Analysis Over Time, Memory Leak Detection, Domain Specific Language

ACM Reference Format:
Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3297663.3310297

ICPE '19, April 7–11, 2019, Mumbai, India

1 INTRODUCTION

Modern programming languages such as Java use automatic garbage collection. Heap objects that are no longer reachable from static fields or thread-local variables (so-called *GC roots*) are automatically reclaimed by a garbage collector (GC). A memory leak occurs if objects that are no longer needed remain reachable from GC roots due to programming errors. For example, a developer may forget to remove objects from their containing data structures. These objects cannot be reclaimed by the garbage collector and will therefore accumulate over time. Beside excessive dynamic allocations [27–29], memory leaks are one of the major memory anomalies [10].

Applications may involve hundreds of millions of objects at a single point in time. Thus, tools to resolve memory problems are of paramount importance. Most state-of-the-art tools, such as VisualVM [34] or Eclipse Memory Analyzer (MAT) [32], perform heap analyses based on snapshots, i.e., heap dumps. While such tools can group heap objects by their types, they have no notion on how these objects are connected as data structures. This is problematic because memory leaks are frequently related to data structures [41]. By recognizing data structures, users can be provided with further guidance during memory leak detection.

In addition to the problem of missing data structure information, a single heap dump does not give any insights regarding the heap's evolution over time. Thus, some approaches [6, 12, 13, 34], take multiple snapshots and compare them. Nevertheless, this still does not allow temporal analyses on the *object-level*, i.e., its not possible to tell whether a certain object was alive in snapshot *A* and is still alive in a later snapshot *B*.

In contrast to snapshot-based approaches, trace-based approaches continuously record information about events, e.g., allocations or object moves executed by the GC, throughout an application's life time. The recorded trace can later be used to reconstruct the heap for an arbitrary garbage collection point. In addition to that, detailed trace-based approaches are able to track specific objects over multiple garbage collections. One example of a trace-based memory monitoring tool is AntTracks, which is based on the Hotspot Java VM. It was initially developed by Lengauer et al. [17] and has been extended by Weninger et al. [37–39]. All concepts presented in this work have been integrated into AntTracks to prove the feasibility of our approach.

Weninger et al. [36] presented first ideas on how to use memory traces to find the root causes of memory leaks by focusing on the *growth* of *data structures over time*. In this work, we extend and complement our work by a more in-depth description of the approach and the algorithms used, new metrics and metric patterns

for data structure growth analysis, as well as a thorough evaluation of our implementation based on several real-world scenarios in which we detect memory leaks caused by growing data structures.

Our scientific contributions are

(1) a DSL that enables users to describe arbitrary data structures,
(2) an algorithm to detect instances of previously defined data structures in reconstructed heaps,
(3) techniques, metrics and patterns for data structure growth analysis to identify data structures that are possibly involved in memory leaks, as well as
(4) an evaluation of our approach based on memory leak detection in real-world applications.

2 BACKGROUND

AntTracks consists of two parts: The AntTracks VM, a virtual machine based on the Java Hotspot VM [33], and the AntTracks Analyzer, a memory analysis tool. Since the concepts presented in this paper have been integrated into AntTracks, it is essential to understand how AntTracks works.

2.1 Trace Recording by the AntTracks VM

The AntTracks VM records memory events such as object allocation events and object movements executed by the GC by writing them into trace files. It keeps the event size to a minimum and avoids the recording of redundant data [16, 17].

2.2 AntTracks Analyzer

2.2.1 *Reconstruction.*

The AntTracks Analyzer is able to parse previously created trace files. The events in the trace are incrementally processed, which enables to reconstruct the heap at every garbage collection point [1]. A heap state is the set of heap objects that were live in the monitored application at a certain point in time. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by. To allow addressing a specific object within a heap state, every heap object is assigned a unique index.

2.2.2 *Heap Object Classification.*

The AntTracks Analyzer's core mechanism is object classification in combination with multi-level grouping [38, 39] to enable user-driven heap analysis. Using object classifiers, heap objects can be grouped according to certain criteria such as type, allocation site, allocating thread, and so on. For example, the *Type* classifier allows to group objects by their types, e.g. `java.util.LinkedList`. In multi-level grouping, objects are grouped according to the classification results of multiple classifiers. This results in a hierarchical classification tree.

A common classifier combination is to first group all heap objects by their types (using the *Type* classifier) and then by their allocation sites (using the *Allocation Site* classifier). Figure 1 shows an example of a classification tree. Yellow rectangles represent tree nodes and blue circles represent the objects that were classified into the respective tree branch. For example, the objects 0 to 3 are of type `Object[]`, of which the objects 0, 1 and 3 have been allocated in `Stack:init()` and object 2 has been allocated in `MyService:foo()`.

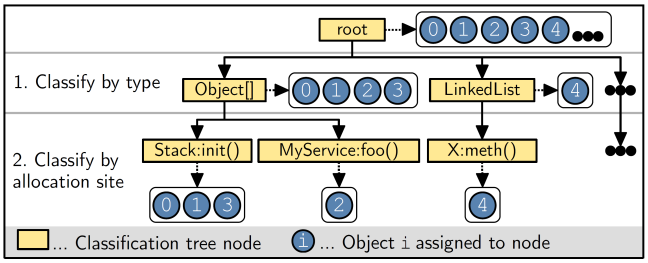


Figure 1: A *classification tree* that first groups all objects by their types and then by their allocation sites.

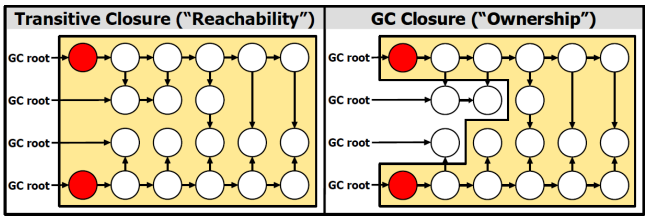


Figure 2: Shared *transitive closure ("reachability")* and *GC closure ("ownership")* of the two red objects.

2.2.3 *Closures and Metrics.*

Users require guidance to decide how they should navigate through a classification tree. AntTracks currently supports three metrics that are displayed for every object group, i.e., for every node in the classification tree:

- *Shallow*
  The shallow object count and the shallow byte count are calculated based on the objects classified at a given node, without taking into account any referenced objects. For example, the shallow object count of the node `Object[]` in Figure 1 is 4. The shallow byte count is the size of the arrays themselves, without taking into account the sizes of the objects referenced by them.
- *Deep*
  The deep object count and the deep byte count are the number of objects / number of bytes of a node's *transitive closure*. The transitive closure contains all objects that are *reachable* from a given object group, as shown in Figure 2.
- *Retained*
  The retained object count and the retained byte count are the number of objects / number of bytes of a node's *GC closure*. The GC closure contains all objects that are *owned* by a given object group, as shown in Figure 2. In other words, the GC closure contains all objects that could be freed by the garbage collector if the given object group would be freed.

3 APPROACH

Over the last years, the memory consumption of applications has grown drastically. This poses a challenge to memory monitoring tools because it results in more complex heap states that have to be visualized in a user-friendly way. Many tools still use flat type histograms (see Figure 4) as their main visualization.

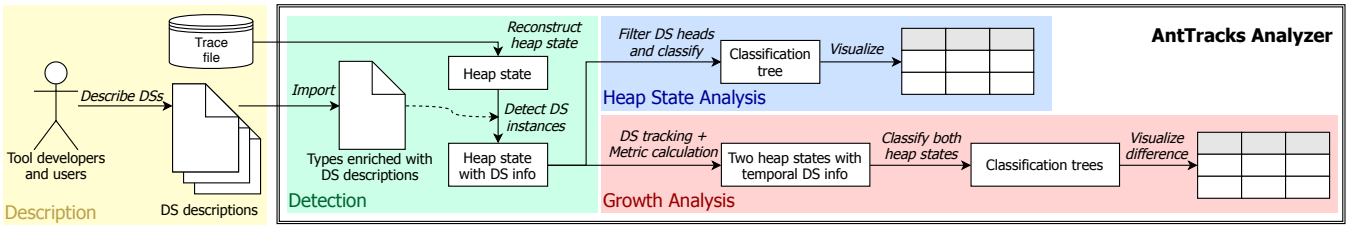


Figure 3: Our approach consists of four stages: (1) *Description* of data structures (DS) by a DSL, (2) *detection* of data structure instances in reconstructed heap states, (3) *heap state analysis*, i.e., data structure analysis at a single point in time, and (4) *growth analysis*, i.e., tracking data structures over time, detecting those with suspicious growth.

Type	# Objects	Shallow Size
`java.util.HashMap$Node`	1.000.000	24 MB
`my.package.MyType`	800.000	16 MB
`int[]`	100.00	50 MB
...	...	...

Figure 4: A *type histogram* displays every type alongside the number of allocated objects and the consumed memory.

In many heap states, most of the objects are auxiliary objects, i.e., internal parts of more complex data structures. These are often located at the top of type histograms. One prominent example are `java.util.HashMap$Node` instances. Though head objects of data structures (e.g., of type `HashMap`) are far more likely the root cause for a memory leak, they are only listed at a lower position.

In this work, we present an approach that greatly reduces the complexity that users have to cope with during memory analysis. The idea is to hide objects that convey little information and to focus on the analysis of data structures instead. Generally speaking, a data structure *is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data* [35]. For memory leak detection, we are especially interested in the relationships among the objects that make up the data structure.

This section explains the core concepts of our approach (see Figure 3): How data structures can be described by a DSL, how they are detected in a reconstructed heap state, how this information can be used to ease heap state analysis, and how information about data structure growth over time can be derived and used in trace-based tools such as AntTracks.

3.1 Data Structure Description

In object-oriented languages such as Java, data structures typically consist of a *head* object, multiple internal objects that serve as a backbone, and leaf objects that represent the actual contents of the data structure. These objects reference each other according to a specific pattern. This pattern has to be known by a memory monitoring tool before it can detect instances of the respective data structure and perform analyses on it.

3.1.1 Benefits of a DSL for Data Structure Description.

The most straightforward way to achieve pattern recognition would be to hard-code the patterns of well-known data structures directly into the tool's data structure instance detection algorithm. However, the set of data structure types that can be detected by the tool would be fixed. Users could not define new data structures, for example data structures specific to their application or data structures introduced by third-party libraries. Also, if the well-known data structures change in the future, e.g., due to renamed types, the tool's source code would have to be modified.

```
1 DS java.util.LinkedList {
2     java.util.LinkedList$Node;
3 }
4 java.util.LinkedList$Node {
5     java.util.LinkedList$Node;
6     (*);
7 }
```

```
1 namespace java.util {
2     DS LinkedList {
3         *;
4     }
5     LinkedList$Node {
6         LinkedList$Node;
7         (*);
8     }
9 }
```

Figure 5: Description of `java.util.LinkedList` in our DSL, without and with using namespaces and wildcards.

To circumvent these drawbacks, we developed a DSL for describing arbitrary data structures in separate files. These data structure description files can then be read by memory analysis tools such as AntTracks to be used for data structure instance detection in reconstructed heap states. Using a DSL for data structure description instead of hard-coded data structure patterns has various advantages. It enables us to ship descriptions of well-known data structures (e.g., data structures in Java's `java.util` package) directly with AntTracks. At the same time, tool users can extend the predefined data structure descriptions with descriptions of their own data structures. Finally, changes to existing data structures do not require changes in the source code, but only in the data structure description file(s) that are much easier to adjust.

3.1.2 DSL Format.

The left side of Figure 5 shows how `java.util.LinkedList` can be described in our DSL. Every type that is a part of the data structure needs a description, i.e., in our example `java.util.LinkedList` and `java.util.LinkedList$Node`. Since Java applies type erasure [3, 4], no information about generics is available at run time and thus we do not include generics in the DSL. `java.util.LinkedList` represents the *head* of the data structure and has to be marked with one of the head keywords (such as `DS`). Internal parts of data structures such as `java.util.LinkedList$Node` are not marked with a keyword. Similar to Java syntax, the name of the type is followed by a pair of curly braces. These contain a set of types (separated by semicolons) that may be referenced by the respective data structure part. We call this set of types *pointed-to types*. For example, an instance of `java.util.LinkedList` may point to instances of `java.util.LinkedList$Node` (line 2), which in turn may point to instances of `java.util.LinkedList$Node` instances (line 5), and so on. Line 6 presents two special language features: (1) a star (i.e., `*`) can be used as a wildcard within the name of a pointed-to type and (2) enclosing a type in parentheses declares it as a *leaf*. The term `(*)` denotes a leaf of any type. Leaf information is used during data structure instance detection to determine the boundaries of a data structure. The DSL also supports namespaces which makes it

possible to omit package declarations in type names. For example, the right side of Figure 5 shows a minimized version of the data structure description. Line 1 defines the namespace `java.util`, thus we can omit the package name for the described types (line 2 and line 5). Since `java.util.LinkedList` only references the list head, we do not have to specify the exact pointed-to type but may use a wildcard instead (line 3). For `java.util.LinkedList$Node`, we may again omit the package name in the pointed-to type (line 6).

3.1.3 Implementation.
To implement our DSL, we used the compiler generator Coco/R [21]. It takes an attributed grammar (in EBNF) of a source language and generates a scanner and a recursive descent parser for it. We chose this approach because it allowed us to rapidly prototype first versions of the DSL and to remain flexible in extending the language's grammar with new production rules. The full grammar can be downloaded here[1].

3.2 Parsing Data Structure Descriptions and Detecting Instances

3.2.1 Assigning Data Structure Descriptions to Types.
Before instances of data structures can be detected in a heap state, the data structure descriptions have to be parsed and assigned to their corresponding types. Types without a data structure description are assigned a non-head dummy description that does not declare any pointed-to types. They will always act as leaves in data structures. Array types of reference types are an exception to this rule. They are assigned a non-head dummy description too, but they declare * (any type) as their pointed-to type. After this step, every type is equipped with its corresponding data structure description.

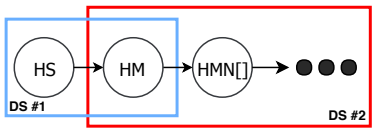
3.2.2 Resolving Type Names.
As described in Section 3.1.2, every type's data structure description defines a set of *pointed-to types* that belong to the data structure. These type names may contain wildcards and have to be resolved to all types matching the name pattern. For example, if a type is defined to have a pointed-to type `*Node`, this has to be resolved to the set of all types whose name ends with *Node*. The pointed-to type * is not resolved into the set of all types (which would lead to enormous memory overhead), but to `java.lang.Object`.

3.2.3 Detecting Instances.
A reconstructed heap contains information about the objects that are live at a certain point in time (e.g., their types), as well as their references between each other. First, the algorithm filters and remembers all objects that are data structure heads, i.e., objects whose types have a *head* data structure description assigned. Then, to determine which objects belong to a certain data structure, the head's pointers are followed recursively. For every object that is met along the way, the first matching branch of the following four is taken:

(1) *The object's type is part of its referencing object's set of pointed-to types and is a data structure head type.*
This it the case if a data structure points to the head of another data structure. The head object is treated as a leaf of the referencing data structure and the descent is stopped. A typical example for this is Java's `HashSet`, which is backed by a `HashMap`. Figure 6 visualizes this pattern. A HashSet only consists of two objects: The head of the hash set and the head of the hash map.
(2) *The object's type is part of its referencing object's set of non-leaf pointed-to types.*
In this case, the object belongs to the referencing data structure, and the descent is continued with its children. For example, this is the case for `LinkedList$Node` instances within a `LinkedList`.
(3) *The object's type is part of its referencing object's set of leaf pointed-to types.*
This means that the object belongs to the data structure, but it is a leaf and thus the descent is stopped. For example, the objects that have been added to a `LinkedList`, i.e., those which are referenced from a `LinkedList$Node` instance, are such leaf objects.
(4) *The object's type is not part of the referencing object's set of pointed-to types.*
This means that the object does not belong to the data structure at all, and thus the descent is stopped.

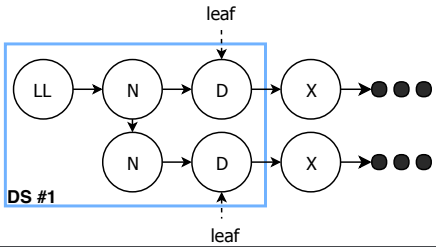
Every visited object is marked to avoid multiple visits.

For example, Figure 7 shows a `LinkedList` that has been detected using the description in Figure 5. The traversal starts at the head `LL`. Next, the first `N` instance is visited. The data structure description of `LinkedList$Node` then tells us to follow further `N` nodes (left side, line 5), or to visit any other object as a leaf without continuing the recursive descent (left side, line 6). Thus, the first `D` instance and the second `N` instance are visited, continuing the descent from the `N` object. As a last step, the second `D` object is visited as a leaf.



HS = HashSet, HM = HashMap, HMN[] = HashMap$Node[]

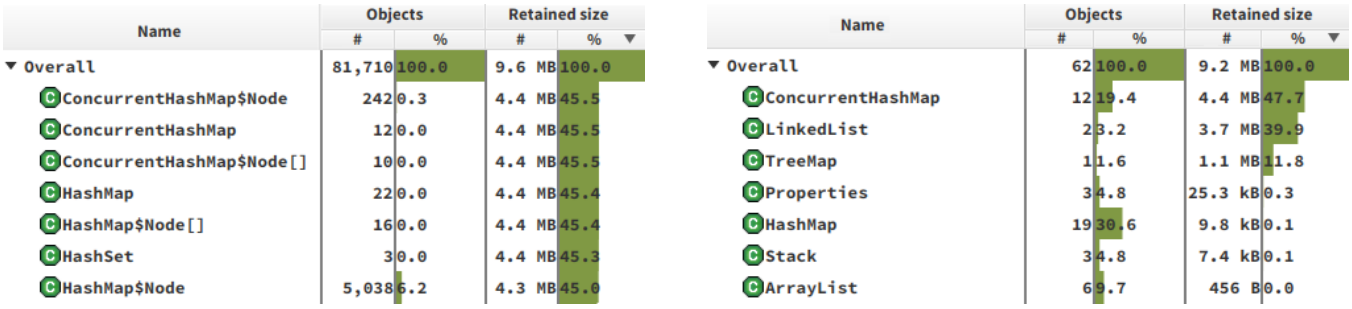
Figure 6: A `HashSet` only consists of two objects: The set's head (`HS`) and the contained hash map's head (`HM`).



LL = java.util.LinkedList, N = java.util.LinkedList$Node, D = Data, X = X

Figure 7: A `LinkedList` data structure instances that consists of the head (`LL`), two nodes (`N`) and two data objects (`D`).

[1] http://ssw.jku.at/General/Staff/Weninger/AntTracks/ICPE19/DSL.atg



Name	Objects #	Objects %	Retained size #	Retained size %
Overall	81,710	100.0	9.6 MB	100.0
ConcurrentHashMap$Node	242	0.3	4.4 MB	45.5
ConcurrentHashMap	12	0.0	4.4 MB	45.5
ConcurrentHashMap$Node[]	10	0.0	4.4 MB	45.5
HashMap	22	0.0	4.4 MB	45.4
HashMap$Node[]	16	0.0	4.4 MB	45.4
HashSet	3	0.0	4.4 MB	45.3
HashMap$Node	5,038	6.2	4.3 MB	45.0

(a) Object view

Name	Objects #	Objects %	Retained size #	Retained size %
Overall	62	100.0	9.2 MB	100.0
ConcurrentHashMap	12	19.4	4.4 MB	47.7
LinkedList	2	3.2	3.7 MB	39.9
TreeMap	1	1.6	1.1 MB	11.8
Properties	3	4.8	25.3 kB	0.3
HashMap	19	30.6	9.8 kB	0.1
Stack	3	4.8	7.4 kB	0.1
ArrayList	6	9.7	456 B	0.0

(b) Data structure view

Figure 8: AntTracks's object-based heap state view compared to AntTracks's data-structure-based heap state view.

3.3 Heap State Analysis

Once all data structures have been detected in a certain heap state, the user may utilize this information to investigate potential memory leaks. In AntTracks, a certain heap state can be investigated by applying user- or predefined classifiers on the objects in the heap in order to group them. New classifiers have been developed that utilize the newly gained data structure information for classification. In the following, we present the application of some of these new classifiers, which can be used for top-down analysis as well as for bottom-up analysis. Furthermore, we present the *data structure view*, a new feature to ease top-down analysis.

3.3.1 *Top-down Heap Analysis.*

In AntTracks, when inspecting a heap state, all objects are initially classified by their types. This is visualized in Figure 8a. Since we are looking for the root cause of a memory leak, i.e., for those objects that keep a lot of other objects alive, we sorted the types by their retained size (i.e., by their ownership). However, the table in Figure 8a still mostly shows internal parts of data structures. These are often inaccessible for developers and are thus of minor interest when looking for the root cause of a memory leak.

Data structure view. We introduce the *data structure view*, a view on the heap state that filters out every object that is not the head of a previously defined data structure. Additionally, data structures that are completely contained in another data structure (i.e., owned by another data structure) are hidden. A typical example for such hidden data structures are `HashMaps` that are completely contained in a `HashSet` (as previously shown in Figure 6).

When applying the data structure view on the same heap state as in Figure 8a, only a small fraction of the original heap objects remains visible, as can be seen in Figure 8b. In this example, internal objects such as `ConcurrentHashMap$Node` instances are hidden by the data structure view. Additionally, some data structures, such as threee `HashSets` and three `HashMaps`, are not shown since they are completely contained in another data structure. Depending on the application, data structure head objects often make up far less than 1% of the application. Also the number of object groups, i.e., entries in the classification tree, is greatly reduced. In the example of Figure 8b, it is now easy to tell that among all data structures, instances of `ConcurrentHashMap` together keep alive 47.7% of the heap. Since the object group of interest still contains 12 objects, we further drill down, i.e., we perform a top-down analysis, as shown in Figure 9. First, we inspect the allocation sites of the maps.

	Name	Objects #	Objects %	Retained size #	Retained size %
1	Overall	62	100.0	9.2 MB	100.0
2	ConcurrentHashMap	12	19.4	4.4 MB	47.7
3	DataStructureDevelopmentExample.main...	1	1.6	4.4 MB	47.5
4	Data structure leaves	5,003	100.0	4.2 MB	100.0
5	DeepLongData	5,000	99.9	4.2 MB	100.0
6	DataStructureDevelopmentExample...	5,000	99.9	4.2 MB	100.0
7	Integer	1	0.0	16 B	0.0
8	HashMap$KeySet	1	0.0	16 B	0.0
9	Object	1	0.0	16 B	0.0
10	ClassLoader.<init>(Void, ClassLoader...	2	3.2	6.2 kB	0.1

Figure 9: Top-down analysis splits object groups until they are small enough to be analyzed in detail.

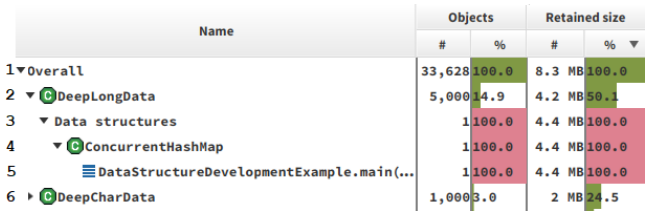
Line 3 reveals that the map that has been allocated in method `main` of class `DataStructureDevelopmentExample` keeps alive 47.5% of the heap on its own.

Leaf classifier. At this point, the user may want to obtain information about the map's leaves, i.e., the actual key and data objects contained in the map. The new *leaf classifier* enables users to classify the leaves of a data structure using any classifier combination. For example, in Figure 9, the *leaf classifier* has been used to classify all leaf objects by their types and allocation sites. This clearly identifies the leaves of type `DeepLongData` (line 5) that have been allocated in the class `DataStructureDevelopmentExample` (line 6) to consume the most memory. Knowing which data structure keeps a large number of objects alive, as well as which leaf objects within this data structure take up the most memory, should suffice to fix the memory leak.

3.3.2 *Bottom-up Heap Analysis.*

In top-down analysis, users search for objects or object groups that keep many other objects alive. An alternative approach is the bottom-up approach. Tool users may search for objects that exist in large quantities and then want to find out which other objects keep them alive.

Our approach performs bottom-up analysis at a higher level of abstraction to reduce complexity. Instead of analyzing which *objects* keep the object group of interest alive, we suggest to look for the *data structures* that keep that group alive. For example, Figure 10 shows a classification tree that has been used to perform bottom-up analysis in AntTracks. On the first tree level (line 2 and line 6), objects have been classified by type. This way, we can see that about 50% of heap is kept alive by objects of type `DeepLongData` (line 2).



	Name	Objects		Retained size	
		#	%	#	%
1	Overall	33,628	100.0	8.3 MB	100.0
2	DeepLongData	5,000	14.9	4.2 MB	50.1
3	Data structures	1	100.0	4.4 MB	100.0
4	ConcurrentHashMap	1	100.0	4.4 MB	100.0
5	DataStructureDevelopmentExample.main(...	1	100.0	4.4 MB	100.0
6	DeepCharData	1,000	3.0	2 MB	24.5

Figure 10: Bottom-up analysis is used to find those objects that keep a given set of objects alive.

Containing data structures classifier. The next step in our approach is to inspect those data structures that contain the objects of interest. To support this, the new *containing data structures classifier* has been developed. This classifier takes an object group and collects the heads of all data structures that contain these objects. These head objects can then be classified using any classifier combination. The information about which objects are contained in the different data structures is obtained and remembered during data structure instance detection, as explained in Section 3.2.3.

For example, the *containing data structures classifier* has been used on the `DeepLongData` objects in Figure 10 to classify the containing data structure heads by their types, followed by their allocation sites. As we can see, all `DeepLongData` objects are contained in a single data structure (line 3), which is of type `ConcurrentHashMap` (line 4) and has been allocated in class `DataStructureDevelopmentExample` (line 5). This map could now again be investigated further by using the top-down approach described in the previous section.

3.4 Data Structure Growth Analysis

The analysis of data structures at a single point in time may already yield useful insights on the structure of the heap. Growth analysis further supports the user in the search for memory leaks. It considers the growth of data structures over time, which makes it even easier for users to spot those involved in memory leaks. This section describes how objects can be tracked over time using trace-based approaches such as AntTracks. In addition, we present metrics to analyze data structure growth as well as metric patterns based on different growth types. Finally, we present how AntTracks's existing classification system has been integrated into the new growth analysis. This way, users can, for example, detect data structures that grew over time, tell which set of leaf objects grew the most, and finally where these leaves have been allocated.

3.4.1 *Data Structure Instance Tracking.*

We argue that trace-based approaches are better suited for temporal analyses than snapshot-based ones. Trace-based approaches can derive temporal information on the *object-level*, while snapshot-based approaches have no concept of *object identity*. Using only snapshots without additional information (e.g., object tagging), it is not possible to decide whether an object that was alive in a certain snapshot still lives in a later snapshot. Figure 11 illustrates this. In a snapshot-based approach, it can only be inferred that both snapshots contained one `X` object, but not whether these two objects are actually the same instance.

Using AntTracks, we are able to derive this information by replaying the recorded GC move events. Furthermore, we are able

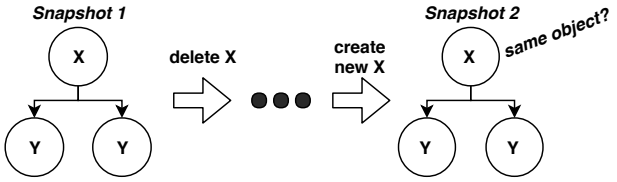


Figure 11: Analysis based on multiple snapshots lacks information on the *object-level*.

to reconstruct the heap object reference graph, i.e., all references between objects, for both points in time. Using this knowledge, we can specifically search for data structures which (1) survived a certain time window (objects that have died cannot be the root cause of a memory leak) and (2) reference / keep alive more objects than before.

The workflow for data structure instance tracking consists in the following steps:

(1) The user selects two garbage collection points between which the data structure growth analysis should take place.
(2) The heap is reconstructed for the first point in time and is stored. The addresses of all data structure heads in this heap, i.e., the *start addresses*, are stored as well.
(3) If a data structure head dies during a garbage collection, we stop tracking it. For surviving heads, their new addresses (which can be reconstructed from GC move events) are stored alongside their start addresses.

Following this algorithm until the end of the selected time window, we obtain (1) the reconstructed heap state at the start, (2) the reconstructed heap state at the end, and (3) a list of all data structures that survived, i.e., their start and final addresses.

3.4.2 *Growth Metrics.*

For both points in time, i.e. the start and the end of the selected time window, various metrics can be calculated for every data structure head. Subsequently, the absolute growth (both for object count and byte count) can be calculated for each metric. In this section, we present those metrics that have proven most useful in identifying problematic data structures. For simplicity, we refer to both, the object count and the byte count, as size.

Retained size growth. The retained size denotes *ownership* and is calculated from the GC closure, as explained in Section 2.2.3. A large retained size means that if the head of the data structure were collected by the garbage collector (i.e., all references to it were removed), a large number of objects / bytes could be collected with it. If the retained size of a data structure grew considerably between two points in time, it is a strong indication for the fact that the data structure is involved in a memory leak. By default, AntTracks's data structure growth analysis sorts all data structures by this growth metric. This allows us to highlight those data structures that have a suspiciously large growing ownership.

Transitive size growth. A data structure's *transitive size* denotes *reachability*, i.e., how many objects / bytes (inside or outside the data structure) can be reached from it. At a first glance, the growth of a data structure's reachability may not seem very useful as a metric on its own. Nevertheless, we will show that it becomes a valuable metrics as soon as it is put into relation with other metrics.

(Deep) Data structure size growth. The two previous metrics ignored data structure boundaries. Now, assuming a list with only a few objects inside it. If the list itself does not grow, i.e., no new objects are added, but its data objects grow, the metrics mentioned above would change.

This is why we introduce the two new metrics: *data structure size* and *deep data structure size*. These metrics are calculated from the newly proposed closures *data structure closure* and *deep data structure closure*. The metrics are supposed to show whether the data structure itself grew, i.e., whether new objects have been added to it.

The *data structure closure* contains all objects that belong to the given data structure. However, objects that belong to other data structures within the given data structure are not included. On the other hand, the *deep data structure closure* also includes objects that are part of such contained data structures. For example, let us revisit the `HashSet` from Figure 6. Every hash set's data structure closure contains only two objects: The `HashSet` object itself and the contained `HashMap`. A hash set's deep data structure closure, however, also contains all objects that belong to the hash map, and, if the hash map contains further data structures, also the objects that belong to them.

Heap growth portion (HGP). While it is possible to work purely with absolute growth metrics, our evaluation has shown that metrics are easier to interpret when they are put into relation to the absolute heap growth. Given that the overall heap size increased, the *heap growth portion* (i.e., the portion by which the growth of a specific data structure contributes to the overall heap growth) can be calculated for every metric as shown in Equation 1.

$$HGP_{metric}(ds) = \frac{\Delta metric(ds)}{\Delta heapsize} \cdot 100 \tag{1}$$

For example, $HGP_{retained}(ds)$ puts the retained size growth $\Delta retained(ds)$ of data structure ds into relation with the overall heap growth $\Delta heapsize$.

Assume that the overall heap size increased by 1GB and a list's retained size increased by 0.7GB. This would result in a $HGP_{retained}$ value of 70%, i.e., the ownership growth of this data structure contributes 70% to the total growth of the heap, which is a strong indication that this list causes a memory leak.

3.4.3 Metric Patterns.

This section discusses five typical metric patterns (see Figure 12) that may occur in various applications. Based on these patterns, users can easily identify the growth type of a data structure, and can determine how to proceed with the investigation.

Single-ownership container growth. If a data structure has a strong retained size growth in combination with a strong (deep) data structure size growth, we can infer two important properties: (1) New objects have been added to the data structure (i.e., container growth), and (2) the data structure keeps the newly added objects alive (i.e., single-ownership). This indicates that it is possibly involved in a memory leak. Nevertheless, this type of memory leak is rather easy to resolve, since the accumulating objects are kept alive by this data structure alone. This means that the leaf objects can be collected by the GC as soon as they are removed from the data structure.

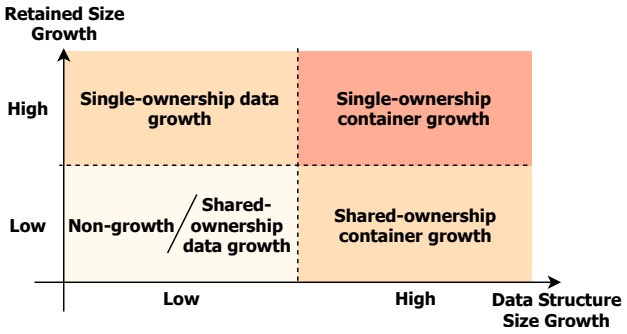


Figure 12: Five common metric patterns.

Inspecting the allocation sites of the data structure as well as of the accumulating leaf objects should yield enough information to identify the source code locations of interest.

Shared-ownership container growth. Similar to *single-ownership container growth*, new objects have been added to the data structure. Yet, the newly added objects are not kept alive by this instance alone. Other data structures may be involved as well.

To find a way to free the newly added objects, the tool user has to analyze the data structure's leaves in more detail. It is not enough to just remove the leaves from this data structure to make them eligible for garbage collection. They have to be removed from *all* their containing data structures. To find these containing data structures, bottom-up analysis (as shown in Section 3.3.2) can be performed.

Single-ownership data growth. We use this term for a strong retained size growth in combination with a weak (deep) data structure growth. In contrast to *single-ownership container growth*, not the data structure itself is growing (i.e., no or only few new elements have been added). Instead, the ownership over the contained data grew. There are two explanations for this.

One possible reason is that the already owned data grew. For example, imagine a list that stores and owns 100 objects which do not reference any other objects. Over the analyzed time window, the list has not been extended, but the contained objects now reference other objects. This has the effect that the data structure size remains unchanged while the retained size grows.

Another possibility is that multiple data structures share ownership on data objects (or parts of them). The retained size of all involved data structures would be small at the start of the selected time window. However, the retained size would grow for one data structure if the shared ownership changed to single-ownership, e.g., because the shared data has been removed from all data structures except this one.

Non-growth / Shared-ownership data growth. There are two possible patterns for the case that neither the retained size nor the (deep) data structure size grew considerably. They are distinguished based on their deep size growth.

We call the first one *non-growth* data structures. In addition to low retained size growth and low (deep) data structure size growth, also the deep size did *not* change considerably. This is the case when the size of a data structure approximately stayed the same, i.e., neither were many new objects added to the data structure nor

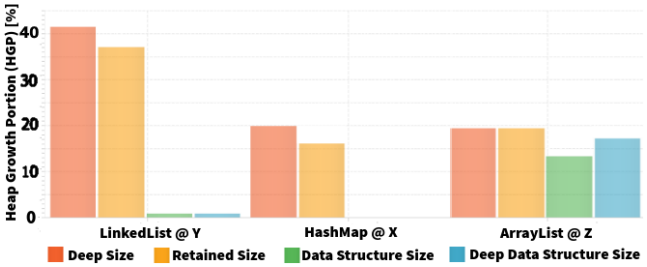


Figure 13: Bar chart to give a quick impression on data structure growth.

did the contained data grow considerably. Thus, data structures classified as *non-growth* do not contribute to a memory leak in the selected time window.

If, however, the deep size *did* grow, the data structure growth can be classified as *shared-ownership data growth*. This means that the data structure itself did not grow, but its data became larger (similar to single-ownership data growth). Yet, in contrast to single-ownership data growth, the data structure does not own the newly referenced objects.

3.4.4 *Visualization and Classification.*

At this point, all data structures have been tracked over the selected time window and their growth metrics have been calculated. Now these metrics have to be visualized to users in a way that allows them to decide which data structures they should investigate in more detail.

Visualization. AntTracks presents the data structure growth analysis results to the user in two ways. The first one is a bar chart which displays the *HGP* values for deep size growth, retained size growth, data structure growth and deep data structure growth. The bar chart shows the ten data structures with the strongest growth of the currently selected metric. By default, the data structures are sorted and selected based on the retained *HGP*, but users may change this sorting. This visualization gives a quick overview of the metric combinations of those data structures that had the strongest growth of the selected metric. An example is given in Figure 13.

More detailed analysis is possible using a tree table view, similar to the one used in heap state analysis. Every data structure is presented by a single record. The table columns shown by default are absolute and *HGP* values of deep size growth, retained size growth, deep data structure size growth, as well as the object count. By default, the data structures are sorted by their retained size growth, but the user may change the sorting order to any other metric. This way, the data structures can be searched for metric combinations.

Classification. If a suspicious data structure is detected, the user can use AntTracks's classification system to gain further insight on it. Using the selected classifiers, the data structure is then classified twice: Once at the start of the selected time window, and once at the end. The difference between the two resulting classification trees is then calculated and visualized. This is especially useful to analyze a data structure's leaf growth behavior. A typical approach is to first classify the data structures by their types, then by allocation sites, followed by the leaf classifier which classifies the leaves by their types and allocation sites. Figure 14 shows an example of this. First,

	Name	Objects		Retained size	
		Before	After	Absolut...	HGP [%]
1	java.util.LinkedList [4,137,708,648]	1	1	960 kB	19.3
2	DataStructureDevelopmentExample.main(...) :24	1	1	960 kB	19.3
3	Data structure leaves	5,000	20,000	600 kB	12.1
4	FlatData	5,000	20,000	600 kB	12.1
5	DataStructureDevelopmentExample.main(...) :175	0	5,000	200 kB	4.0
6	DataStructureDevelopmentExample.main(...) :255	0	5,000	200 kB	4.0
7	DataStructureDevelopmentExample.main(...) :303	0	5,000	200 kB	4.0
8	DataStructureDevelopmentExample.main(...) :91	5,000	5,000	0 B	0.0

Figure 14: Tree table view that displays the growth of a data structure using a given classification.

one can see that the LinkedList's retained *HGP* is about 19%. In line 2, the list's allocation site is shown. Then, the leaf classifier has been applied (line 3). The *Objects* column shows that the number of leaves raised from 5,000 leaves at the start of the time window to 20,000 leaves at the end of the time window. The *Retained size* column shows that this increase in the number of leaves accounts for about 12% of the overall heap growth. Line 4 shows that all of these leaves are of type `FlatData`. Lines 5 to 8 show the various allocation sites of the leaves. It can be seen that the new leaves have been allocated at three different allocation sites (line 5 to 7). Line 8 conveys the information that the number of leaves that have been allocated at this particular allocation site has not changed.

4 APPLICATION TO CASE STUDIES

To evaluate the usefulness of data structure descriptions and their usage in heap analysis over time, we applied them on two different real-world systems.

The first system is *Dynatrace easyTravel* [9]. Dynatrace focuses on application performance monitoring (APM) and distributes easyTravel as their state-of-the-art demo application. It is a multi-tier application for a travel agency, using a Java backend. An automatic load generator can simulate accesses to the service. When easyTravel is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

The second system is *AntTracks itself*. AntTracks is under constant development by the authors, as well as by students that do projects with and within AntTracks. Lately, we were dealing with an increasing memory footprint over time during the parsing of trace files. To find the root cause of this memory leak, we analyzed AntTracks using AntTracks's data structure growth analysis, and document this scenario here.

4.1 easyTravel

EasyTravel was executed on the AntTracks VM, which generated a trace file. This trace file was then opened in the AntTracks Analyzer. After parsing the trace file, multiple charts are presented to the user, displaying the application's memory behavior. For example, Figure 15 shows the number of allocated objects separated by heap space type (y-axis) over time (x-axis). In most generational garbage collectors, objects are allocated in a heap space called *eden*, are then moved to a *survivor* space if they survive at least on garbage collection, and are eventually promoted to an *old* space after a certain number of garbage collections. The growing number of objects in the old space (red) clearly indicates that the heap consumption grew over time.

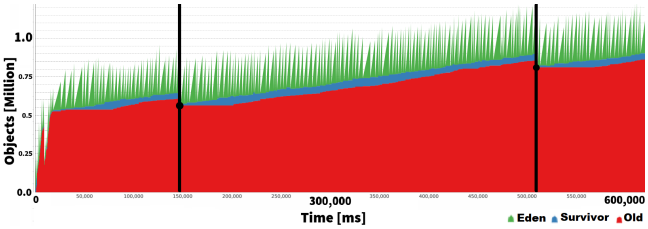


Figure 15: Object count evolution that hints at a memory leak in easyTravel.

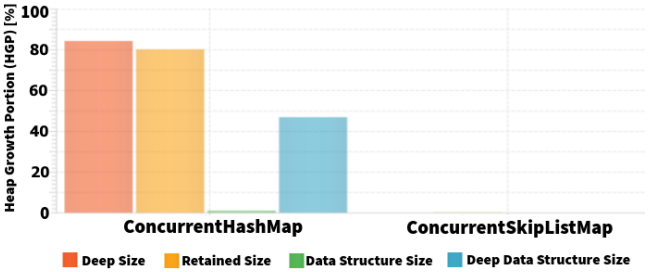


Figure 16: Metric growth bar chart that highlights the `ConcurrentHashMap` as memory leak suspect due to its strong retained size growth.

	Name	Objects	
		Before	After
1	Overall	25,748	25,748
2	java.util.concurrent.ConcurrentHashMap...	1	1
3	JourneyService.findLocations(String,...	1	1
4	Data structure leaves	32,058	136,224
5	Location	31,781	135,026
6	GeneratedConstructorAccessor33.n...	31,769	135,014
7	Constructor.newInstance(Object[]...	12	12
8	JourneyService$QueryKey	276	1,196
9	JourneyService.findLocations(Str...	276	1,196
10	ConcurrentHashMap$CounterCell	0	1
11	Collections$EmptyList	1	1

Figure 17: Classification of the conspicuous map based on allocation site followed by leaf classification based on leaf types and leaf allocation sites.

To perform data structure growth analysis, the user can specify a time window by selecting two points in time (the vertical black lines in Figure 15). In this example, we selected the end of two major garbage collections. At these points we can assure that all heap objects that were not reachable anymore have been collected by the GC.

Once all surviving data structures have been tracked over the selected time window, their growth are visualized in the bar chart (Figure 16) and the tree table view. What can be seen at a glance is that the retained *HGP* of the `ConcurrentHashMap` is about 80%. This clearly identifies this specific `ConcurrentHashMap` as the memory leak culprit. What can further be derived from the other metrics is that nearly no new objects have been added to the data structure itself (i.e., very low data structure size growth). Yet, since the deep data structure size grew strongly, we can derive that the map must contain further data structures, and that these data structures have grown. As explained in Section 3.4.3, this can be categorized as *single-ownership container growth*.

Since we now know that this concurrent hash map is the major suspect for the memory leak, we want to gain more information about it until we are able to fix the leak. Figure 17 shows the classification tree that we used to analyze the map. The first classifier that has been used on the map is the *allocation site classifier*. It tells us that the map has been allocated in the method `findLocations` of class `JourneyService` (line 3). The second classifier that has been applied is the *leaf classifier*. As the name suggests, this classifier can be used to inspect the leaves of a data structure. The classifier has two modes: either *own leaves* or *deep leaves*. In the first mode, the classifier would inspect only the leaves of the map itself, without checking the leaves of contained data structures. Since we know that the map contains further data structures, the *deep leaves* mode has been selected. Furthermore, the classifier was configured to classify all leaf objects based on their types and allocation sites. Looking at the *Objects* column, we can see that the number of leaves grew from about 32,000 to about 136,000 (line 4). We can further see that nearly all of these leaves are of type `Location` (line 5). Unfortunately, their allocation sites are not useful, since they are somewhere hidden in a framework (line 6). The second growing leaf type is `JourneyService$QueryKey` (line 7). These leaves have been allocated in method `findLocations` of class `JourneyService`.

Even though we had no prior knowledge about the system, we decided at this point that we gained enough insight to investigate the memory leak on the source code level. To prevent the proliferation of the concurrent hash map, we must prevent that its `Location` and `JourneyService$QueryKey` leaves accumulate. In the source code, the map of type `ConcurrentHashMap<QueryKey, Collection<? extends Location>>` was easily found. In the method `findLocations` (the allocation site of the accumulating `QueryKey` instances) we found that the map should have served as a cache for location searches. Once a search was executed for a given `QueryKey`, the key was stored in the map, alongside its search result (a `Collection<Location>`). Subsequent searches for the same key should have found the respective entry in the map. Yet, `QueryKey` neither implements `hashCode` nor `equals`. Thus, every request resulted in a cache miss and consequently a new cache entry, which led to this typical memory leak.

4.2 AntTracks

Figure 18 shows AntTracks's object count evolution during trace file parsing. Similar to easyTravel, we can see an increase of objects in the old generation of the heap.

Yet, in contrast to easyTravel, which had to be analyzed without prior knowledge about its data structures, this time we could use the data structure DSL to describe AntTracks's most important data structures prior to analysis. Since the leak occurred during trace parsing, the first data structure that was described was AntTracks's internal representation of heap states. To mimic the structure of the real heap, AntTracks separates the heap under reconstruction in a number of `Space` instances, which are further divided into `LAB` (local allocation buffer) instances, which then store the actual information about objects, mostly using arrays. We also described AntTracks's data structure that keeps track of symbols information such as type names, allocation sites, and so on, since this data could also have been corrupted during trace parsing.

```
namspace java.util { // By default shipped with AntTracks
  HashMap$Node {
    HashMap$Node;
    (*);
  }
  DS HashMap {
    HashMap$Node[];
  }
  // ... other java.util classes
}

namespace at.jku.anttracks.util { // Added for specific use case
  DS ApplicationStatistics {
    java.util.HashMap; // HashMap<Thread, MeasurementGroup> and others
  }
  ApplicationStatistics$MeasurementGroup {
    *; // Various internal objects, including List<Measurement>
  }
  ApplicationStatistics$Measurement { }
}
```

Listing 1: Description of AntTracks's data structure to track the execution time of certain code segments.

AntTracks has an internal performance evaluation feature called `ApplicationStatistics`, which is implemented as a singleton. It supports to creation of `Measurement` objects, which can be used to evaluate how much time is spent by which thread in certain code segments. Multiple measurements are then grouped together in a `MeasurementGroup` instance. These data structure parts have also been described, and their descriptions can be seen in Listing 1.

After calculating the data structure growth over the time window selected in Figure 18, an overview bar chart (Figure 19) and a tree table view is shown. By looking at the bar chart, it becomes clear that the memory leak is caused by the `ApplicationStatistics` instance. The metric pattern is akin to that of the memory leak found in easyTravel: a typical *single-ownership container growth*. It may be noteworthy to mention that a data structure's *HGP* values can be above 100%, as can be seen in Figure 19, where the `ApplicationStatistics`'s retained *HGP* is around 115%. In this case, the overall heap grew by about 100MB, while the `ApplicationStatistics`'s ownership grew by 115MB, which can happen if previous multi-object ownership changed to single-object ownership, as explained in Section 3.4.3.

Figure 20 shows the classification tree used to analyze the memory leak. Since `ApplicationStatistics` is implemented as a singleton, it is not necessary to check its allocation site. The result of the *leaf classifier* already provided enough information to resolve the memory leak. The classifier has been configured to classify each leaf by its type, followed by its allocation sites, as well as the call sites of the allocating methods. Line 3 shows that the overall number of leaves has skyrocketed from about 6.6 million to 10.2 million. Nearly all of the leaves are of type `Measurement` (line 4), and all of them have been allocated in the `ApplicationStatistic` class (line 5). To further distinguish the measurements, the methods that called the allocating method can be inspected. This reveals that two call sites, both located in the method `parseGCRootPtr` of class `TraceParserSlave` (line 6 and 7), caused the extensive `Measurement` allocations.

Checking the `TraceParserSlave` class, the instrumented parts within `parseGCRootPtr` were easily detected. These parts were

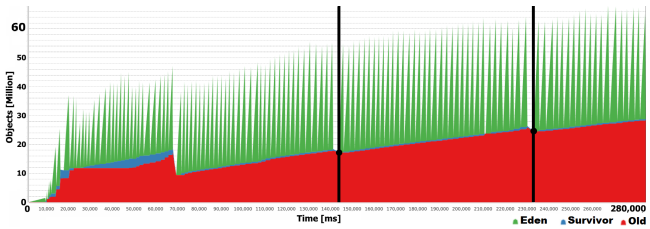


Figure 18: AntTracks's object count evolution shows a similar pattern as in easyTravel.

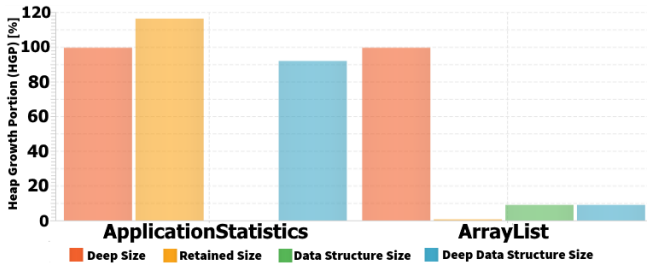


Figure 19: Bar chart clearly showing the single-ownership container growth of the `ApplicationStatistics` instance.

	Name	Objects	
		Before	After
1	Overall	11,486	11,486
2	at.jku.mevss.anttracks.util.ApplicationStatistics...	1	1
3	Data structure leaves	6,632,638	10,263,731
4	ApplicationStatistics$Measurement	6,632,610	10,263,703
5	ApplicationStatistics.createMeasurement(Strin...	6,632,610	10,263,703
6	TraceSlaveParser.parseGCRootPtr(EventType,...	4,187,985	6,454,359
7	TraceSlaveParser.parseGCRootPtr(EventType,...	2,340,602	3,662,890
8	TraceSlaveParser.parseGCRootPtr(EventType,...	8	14
9	MutableLong	16	16
10	Thread	12	12

Figure 20: Classifying the leaves of `ApplicationStatistic` by type, allocation site, and call site.

frequently called and thus created a vast amount of `Measurement` instances. Since they were not essential to AntTracks's functionality, they were simply removed to resolve the memory leak.

5 RELATED WORK AND STATE-OF-THE-ART

To support memory leak detection as well as to facilitate memory leak resolving, various approaches and tools have been developed over the last years. Šor and Srirama [43] classify these approaches into the following groups:

(1) *Online approaches* that actively monitor and interact with the running virtual machine, separated into approaches that
   (a) *measure staleness* [2, 11, 23, 24, 41]. Staleness is not a quantifiable metric, instead, the longer an object is not used, the more stale it becomes. The idea behind approaches that measure staleness is that objects that do not get collected by the GC for a long time but become stale are more likely to be leaking than non-stale objects. The challenge that these approaches face is that object access tracking is extremely expensive.
   (b) *detect growth* [5, 12, 13, 30, 31]. These approaches group the live heap objects (mostly based either on their types or allocation sites) and detect growth using various metrics.

These metrics range from simple absolute count differences between allocations and deallocations [5] to more complex ones based on the structure of the object reference graph [12, 13].

(2) *Offline approaches* that collect information about an application for later analysis, separated into approaches that
   (a) *analyze heap dumps as well as other kinds of captured state* [15, 18–20]. Compared to online approaches, offline approaches often perform more complicated analyses based on the object reference graph, involving graph reduction, graph mining and ownership analysis.
   (b) *use visualization* to aid manual leak detection [6, 22, 25].
   (c) *employ static source code analysis* [7, 42].
(3) *Hybrid approaches* that combine online features as well as offline features [8, 26, 40].

For example, one of the approaches most similar to our approach is *container profiling* by Xu and Rountev [41]. They also focus on data structures, but instead of detecting growth, they track operations on containers and detect container staleness. Their approach requires ahead-of-time modeling of containers, i.e., the user has to introduce a "glue layer" in the monitored application's source code that maps methods of each container type to primitive operations (e.g., ADD, GET, and REMOVE). Compared to that, our data structure description mechanism using a DSL is much less invasive.

Future work (see Section 6) encompasses plans to automatically infer data structure descriptions from source code and memory traces. For example, Mitchell and Sevitsky [19] developed *LeakBot*, a tool that performs memory analysis using object aggregation. They present various metrics to detect possible top-level *leak roots*, which may correspond to data structure heads. Jump and McKinley [14] introduced *dynamic shape analysis*, which seeks to *characterize data structures* by summarizing the object pointer relationships into *degree metrics*. Metrics like these may facilitate the process of automatically inferring data structure descriptions.

6 FUTURE WORK

Our data structure description DSL can be used to describe arbitrary data structures. However, users might find it tedious to describe a greater number of custom data structures which they use in their project. To relieve the user of this task, reasonable data structure definitions should be inferred automatically from static information, such as the source code, in combination with dynamic information obtained during trace parsing. The DSL presented in this work would not become obsolete by such a feature because automatically detected data structure descriptions would most likely require corrections or extensions by the user. Moreover, in some cases, users might want to describe undetectable reference patterns as data structures.

In AntTracks, the heap and subsets thereof are currently represented in the form of charts and tree table views. In the future, users should also be able to browse through objects and their reference patterns in a visualized reference graph. However, considering the number of objects and references in heaps of modern applications, visualizing complete reference graphs is infeasible in terms of performance. Moreover, graphs of such dimensions are also impossible to comprehend for users and consequently are not of much use to locate the root cause of a memory leak. The newly gained information about data structures in the monitored application could be used to greatly reduce the number of nodes and edges that have to be visualized. Instead of displaying every object as a separate node, objects that belong to a given data structure could be collapsed into a single node, ultimately reducing the object reference graph to a *data structure reference graph*. Guided by the metrics that were presented in this paper, users could locate the root cause of a memory leak by visually browsing through such a graph, investigating the data structures they are interested in.

7 THREATS TO VALIDITY

In this paper, we presented five metric patterns that commonly occur during data structure growth analysis. Each of these patterns suggests different analysis steps. This poses two potential threats to validity: (1) Users may find it hard to comprehend all metrics and their patterns without prior training, and (2) one could argue that the presented list of patterns is not exhaustive and that further patterns could be defined (for example, patterns involving shrinking metrics have not been discussed in this paper). To deal with these threats, future work includes automatic decision making by the tool. New heuristics should be defined that allow the tool to automatically detect metric patterns. Based on detected patterns, the tool could either perform certain classifications or analysis steps automatically, or it could provide suggestions to the user on how to proceed in the analysis, similar to a learning-by-doing tool approach.

Most probably, the major threat to validity of our work is its currently restricted evaluation based on a limited set of use cases. We plan to search for open-source projects that suffered from memory leaks in the past with the goal of building a reference set of real-world applications that could be used to evaluate memory leak detection tools. Using this set of applications, alongside other applications with seeded memory defects, we plan to conduct a user study with our industry partner as well as with university students. In addition to comparing AntTracks to existing tools, e.g., in terms of found memory leaks, we want to gain insight in how well the study participants are able to understand and use existing memory leak detection features, as well as what other features users expect from a memory monitoring tool. This could help the community to improve the quality of memory monitoring tools in general.

8 CONCLUSION

In this paper, we presented a memory leak detection approach that puts data structures into the focus of its analysis. To prove its applicability, we integrated this approach into AntTracks, a trace-based memory monitoring tool.

Our approach encompasses an easy-to-use domain specific language to describe arbitrary data structures, as well as an algorithm that detects instances of those data structures in reconstructed heaps. Further, we presented a new feature in AntTracks called *data structure view*. It hides objects of lower interest, i.e., data-structure-internal objects, during heap state analysis and emphasizes data structure head objects. This reduces the complexity of heap state analysis users have to deal with in state-of-the-art memory monitoring tools. To inspect conspicuous data structures, we developed

new data-structure-specific analysis features to support top-down as well as bottom-up memory analysis.

Our main contribution is a new technique for analyzing the growth of data structures over time: We (1) showed how to use memory traces to track data structures throughout an application's lifetime, (2) introduced metrics that describe various aspects of data structure growth, (3) discussed how certain metric patterns hint at certain types of memory leaks, and (4) presented techniques to prioritize, visualize and analyze data structures that may be the root cause of a memory leak. Data structure growth analysis aims to further ease the analysis of memory leaks by reducing the number of steps a user has to take to identify the root cause of such leaks. Finally, we evaluated the applicability of our approach using two case studies.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

REFERENCES

[1] 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)*.

[2] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: Bit-encoding Online Memory Leak Detection. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proc. of the 13th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*.

[4] Robert Cartwright and Guy L. Steele, Jr. 1998. Compatible Genericity with Run-time Types for the Java Programming Language. In *Proc. of the 13th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*.

[5] K. Chen and J. Chen. 2007. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Proc. of the 31st Annual Int'l Computer Software and Applications Conf. (COMPSAC '07)*.

[6] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '99)*.

[7] Dino Distefano and Ivana Filipović. 2010. Memory Leaks Detection in Java by Bi-abductive Inference. In *Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering (FASE 2010)*.

[8] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended Analysis for Performance Understanding of Framework-based Applications. In *Proc. of the 2007 Int'l Symposium on Software Testing and Analysis (ISSTA '07)*.

[9] Dynatrace. 2019. Demo Applications: easyTravel. https://community.dynatrace.com/community/display/DL/Demo+Applications+-+easyTravel

[10] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proc. of the 40th Int'l Conf. on Software Engineering: Companion Proceeedings (ICSE '18)*.

[11] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*.

[12] 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 Symp. on Principles of Programming Languages (POPL '07)*.

[13] Maria Jump and Kathryn S. McKinley. 2009. Detecting Memory Leaks in Managed Languages with Cork. *Software: Practice and Experience* 40, 1 (2009).

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

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

[16] 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. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.

[17] 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. Conf. on Performance Engineering (ICPE '15)*.

[18] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *Proc. of the 20th European Conf. on Object-Oriented Programming (ECOOP '06)*.

[19] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '03)*.

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

[21] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. 2019. The Compiler Generator Coco/R. http://www.ssw.uni-linz.ac.at/Coco/

[22] Wim De Pauw and Gary Sevitsky. 2000. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency: Practice and Experience* 12, 14 (2000).

[23] Derek Rayside and Lucy Mendel. 2007. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '07)*.

[24] Derek Rayside, Lucy Mendel, and Daniel Jackson. 2006. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Proc. of the Int'l Workshop on Dynamic Systems Analysis (WODA '06)*.

[25] S. P. Reiss. 2009. Visualizing The Java Heap to Detect Memory Problems. In *5th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '09)*.

[26] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. 2000. Automatic Removal of Array Memory Leaks in Java. In *Proc. of the 9th Int'l Conference on Compiler Construction (CC '00)*.

[27] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proc. of the 2nd Int'l Workshop on Software and Performance (WOSP '00)*.

[28] Connie U. Smith and Lloyd G. Williams. 2002. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*

[29] Connie U. Smith and Lloyd G. Williams. 2003. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*

[30] V. Sor, P. Oü, T. Treier, and S. N. Srirama. 2013. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance (ICSM '13)*.

[31] Vladimir Šor, Nikita Salnikov-Tarnovski, and Satish Narayana Srirama. 2011. Automated Statistical Approach for Memory Leak Detection: Case Studies. In *On the Move to Meaningful Internet Systems (OTM 2011)*.

[32] Eclipse Foundation. 2019. Eclipse Memory Analyzer (MAT). https://www.eclipse.org/mat/

[33] Oracle. 2019. The HotSpot Group. http://openjdk.java.net/groups/hotspot/

[34] Oracle. 2019. VisualVM: All-in-One Java Troubleshooting Tool. https://visualvm.github.io/

[35] Peter Wegner and Edwin D. Reilly. 2003. Data Structures. In *Encyclopedia of Computer Science*.

[36] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.

[37] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proc. of the 15th Int'l Conf. on Managed Languages & Runtimes (ManLang '18)*.

[38] 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 Conf. on Performance Engineering (ICPE '17)*.

[39] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 9th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '18)*.

[40] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *Proc. of the 32Nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '11)*.

[41] Guoqing Xu and Atanas Rountev. 2008. Precise Memory Leak Detection for Java Software Using Container Profiling. In *Proc. of the 30th Int'l Conf. on Software Engineering (ICSE '08)*.

[42] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *Proc. of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*.

[43] Vladimir Šor and Satish Narayana Srirama. 2014. Memory Leak Detection in Java: Taxonomy and Classification of Approaches. *Journal of Systems and Software* 96 (2014).