Software and Computational Systems Lab

Master Thesis

in Computer Science

Symbolic Heap Abstraction with Automatic Refinement

Johannes Knaut

Aufgabensteller:	Prof. Dr. Dirk Beyer
Betreuer:	Karlheinz Friedberger
Abgabedatum:	30.09.2018

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.
This paper was not previously presented to another examination board and has not been published.

Munich, 30.09.2018

. . . .
Johannes Knaut

Abstract

Programs with error conditions that depend on the length of a list can be analysed if the threshold for abstraction is higher than the length that the error condition depends on. The best threshold is just as high that the list is not getting abstracted. This thesis at first looks at the obvious approach of using the longest list length encountered on the error path. Then, a more fine-grained approach is developed that introduces loose connections between lists and their maximum encountered length.

Contents

1 Introduction **2**

2 Background **4**
- 2.1 Literature Review 4
- 2.2 Symbolic Memory Graphs (SMGs) 7
  - 2.2.1 Properties of SMGs 8
  - 2.2.2 SMG Consistency 10
  - 2.2.3 SMG List Abstraction 12

3 SMG Analysis in CPAchecker **14**
- 3.1 Erroneous Candidates for Circular DLLs 14
- 3.2 Materialisation of Nullified DLL Segments 19
- 3.3 Value Replacement of Abstracted Segments 20

4 Symbolic Heap Abstraction with Automatic Refinement **22**
- 4.1 Motivating Examples 22
  - 4.1.1 Symbolic Heap Abstraction 22
  - 4.1.2 CEGAR-based Approach using Maximum Length 24
  - 4.1.3 Threshold Candidate Generation 26
- 4.2 Refinement of Heap Abstraction Threshold 27
  - 4.2.1 Naive Maximum Length Approach 27
  - 4.2.2 Threshold Candidate Approach 28

5 Conclusion **32**

6 Acknowledgments **33**

List of Figures **34**

Bibliography **35**

Chapter 1

Introduction

In contrast to an array, a linked list is a very flexible data structure. Its length can grow without limits just by adding new links. Items can be inserted just by destroying a connection, adding a new link and renewing the connection. Data management can be done very efficiently, nodes can be allocated on-the-fly and released whenever they are not needed anymore. Lists can be traversed forward and backward, the end of a list can link to its beginning, also so-called sentinel nodes can be kept. Lists can have nested lists, that can be nested once again. And lists are also collections of elements and can be thought of as abstractions for linked elements that are all of the same size and share a common type.

This thesis is revolving around Doubly-Linked List Segments, which are the major abstraction used in the analysis of Symbolic Memory Graphs (SMGs). SMGs are a graph-based notation that can be used to model the heap during the analysis of a program. There can be situations in the analysis of a program, where an abstraction of single objects into a comprising structure is necessary for the analysis to make progress, for example in order to find a fixed point for a loop and to escape unlimited unrollings. So, one problem is to get to such an abstraction. Then, there is also the problem, that there can be many different possible abstractions as there can be many possible candidates on the heap. Also, it is not clear where to begin with an abstraction, i.e. how high should the number of linked elements become until an abstraction is executed, and is the implied threshold a global threshold or can it be attached to certain properties.

An important concept in this context is counterexample-guided abstraction refinement, a technique which uses a very abstract model in the beginning and gets incrementally more concrete, while being guided by counterexamples.

As described in [5], if the counterexample that was found by the analysis is feasible, the user is provided with a witness and the analysis terminates. When the path is found to be infeasible, the abstract model was too coarse and will be refined by exploiting information from the counterexample. Then, the analysis can continue with the refined abstraction. If no counterexamples are found, the analysis terminates and reports that the program is safe.

Outline. The thesis has two main chapters, that follow a background chapter that looks at research in shape analysis in a chronological fashion. After going over selected literature, Symbolic Memory Graphs are introduced and some properties of their nodes and edges as well as other concepts are explained. The chapter closes with the important algorithm for list abstraction. After that, the first main chapter details three encountered problems in SMG analysis in CPAchecker and suggests solutions. Then, in the last chapter two approaches for refinement of the heap abstraction threshold are motivated by example programs and then the approaches are presented.

Chapter 2

Background

2.1 Literature Review

In the following, selected research papers of the last 20 years are reviewed with the aim to give a broader overview over past work on shape analysis before focusing entirely on Symbolic Memory Graphs.

Estimating Trees, DAGs and Cycles. According to Ghija and Hendren [8] the goal of shape analysis is to estimate the shape of dynamic data structures, that are accessible from a given pointer. Knowledge about the shape of a data structure can give useful information to exclude paths between specific pointers or to decide if different heap accesses from a pointer can lead to the same heap object. After estimating a useful shape for a data structure accessible from a pointer, the goal is also to retain this information as long as possible and not to exchange a tree or directed acyclic graph attribute with a cyclic one, that has much less information to offer. The authors present an analysis that is directed at programs that use simple recursive data structures, that are built compositionally. The analysis uses approximation of the shape for each heap-directed pointer and direction and interference relationships for pairs of heap-directed pointers. Instead of using more complex abstractions, the focus is on providing practical abstractions. The implementation as a context-sensitive interprocedural analysis in a C compiler performs well for programs using simple data structures but is not powerful enough for programs with more complicated structural changes.

Detecting Memory Leaks using a Pointer Graph. Once a reference to previously allocated memory is lost, subsequent operations cannot restore it. Programs with memory leaks can waste high amounts of memory, may slow down the program or even crash due to shortage of available memory.

Scholz et al. [12] present an approach to detect memory leaks that is based on symbolic evaluation of programs, which is a static symbolic analysis, that uses symbolic variable values and path conditions. Also the notion of a pointer graph is introduced, which if disconnected, implies occurrence of a memory leak. For the description of the heap they use a heap algebra and for referring to allocated objects and symbolic pointers are used, such that all allocated objects are given a unique symbolic number. A pointer graph describes the connectivity of the objects on the heap. One of the heap operations new, free or put can destroy the connectivity if after the operation at least one object can no longer be referenced in the program. The nodes of the tree are heap objects as well as pointer variables and an artificial root node. If the root node can be found in the predecessor sets of each node in the tree, the graph can be proven to be connected. If not succeeding in approximating the solution, respective predecessor sets are assumed to be empty and it can be reported that a memory leak may occur.

Detecting Invalid Memory Access by a two-step approach. Besides leaking memory, another source of program failures is the access to parts of the memory that are not meant to be accessed, for example by dereferencing a pointer that points to invalid memory. This can result in a program crash or the program to get into an undefined state which is usually worse if undetected. Some programming language environments automatically check before a memory access if it is safe to access it, but for C this is not the case. The memory safety analyzer CCURED [11, 6] can transform C programs to memory-safe programs by proving memory accesses safe and inserting run-time checks for accesses for which proving does not succeed. Beyer et al. [2] use CCURED in a first step to annotate the program locations that cannot be proven by a type-based approach with run-time checks, then in a second step they use the more powerful and more expensive analysis of the model checker BLAST [9] to check remaining annotated accesses and either prove their safety or give an execution trace in case of a property violation, or in case of a timeout keep the run-time check that was inserted by CCured in the program.

Using Lazy Abstraction Refinement for Shape Analysis. Shape analysis can detect recursive data structures and use compact representations for them, however it is an expensive analysis. In [3], Beyer et al. therefore do not apply shape analysis globally but only where necessary by applying the lazy abstraction paradigm to shapes. An abstract reachability graph is computed on the fly and its nodes are annotated with both predicate and shape information. This can be summarized as lazy abstraction construction. When applying lazy abstraction refinement only nodes on the path of a spurious counterexample

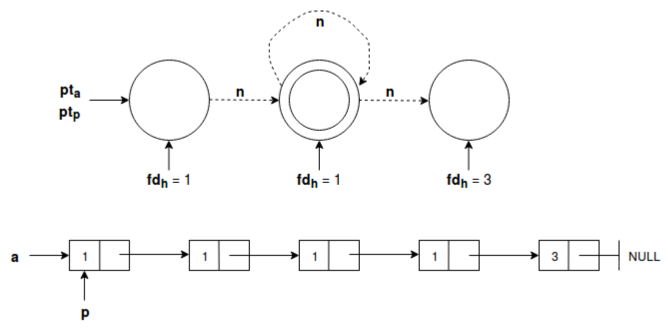

Figure 2.1: Shape graph and possible instance of the graph.

are refined. For each location on this path the developed algorithm decides which pointers and predicates to track and how to refine the heap abstraction in order to remove the spurious error path. Initially the predicates are set to true and the heap abstraction is set to the trivial shape graph. A shape graph is a representation of recursive heap data structures comparable to SMGs. Figure 2.1 shows in the upper part a shape graph annotated by predicates and in the bottom part a possible instance of the graph. The graph represents all instances of lists that store 1 in the field h of all of their nodes except the last node which stores 3, and which have pointers a and p point to the list's first element.

Combining Shape Analysis with Arithmetic Analysis. Magill et al. [10] use an arbitrary arithmetic analysis as back-end for processing counterexamples found in a shape analysis using shape invariants. The counterexamples are generated such that they only contain arithmetic statements and represent all paths that satisfy the shape formulas and could lead to the potential error. By conjoining shape invariants with arithmetic invariants, that are found by arithmetic analysis while trying to prove that a counterexample is not reachable, the strengthened shape invariant can be used in shape analysis and is more likely to rule out the memory error. Also the generated counterexample may contain loop constructs instead of a specific number of loop unrollings in order to generate a strengthening that rules out all spurious counterexamples for the loop because the counterexample program over-approximates the set of counterexample paths.

Discussion. The paper of Magill et al. [10] contains a motivating example which corresponds to the example used in this thesis in listing 4.2 to motivate a CEGAR-based approach for the analysis using the maximum list length in the precision. Generally, since properties of heap data structures such as the list length are integer valued and shape analyses are not inherently designed to reason about arithmetic relationships, the combination with an arithmetic analysis can be used to support shape analysis in cases where reachability of errors depends on list lengths or tree depths, which are commonly not explicitly tracked in heap analysis. Comparing the paper with this thesis, although in this thesis no shape invariants are used, the focus on finding an abstraction threshold that is as small as possible, is similar to strengthening an invariant. Both approaches aim to be only as concrete as necessary to refute the counterexample using shape analysis as efficient as possible.

2.2 Symbolic Memory Graphs (SMGs)

In [7], Dudka et al. have introduced the notion of Symbolic Memory Graphs (SMGs), a graph-based representation of sets of heaps, that supports low-level memory operations.

SMGs are bipartite graphs: According to Asratian et al. [1], a graph is bipartite if its vertices can be sorted into two groups such that edges only join vertices from different groups. Furthermore, SMGs are directed graphs and their nodes and edges are labelled. SMGs consist of two types of nodes and two types of edges:

- Nodes
  - Objects
    - Regions
    - Doubly-linked list segments (DLSs)
    - [Singly-linked list segments (SLSs)]
    - Null object (#)
  - Values
    - 0
    - All values other than 0
- Edges
  - Has-Value Edges
  - Points-To Edges

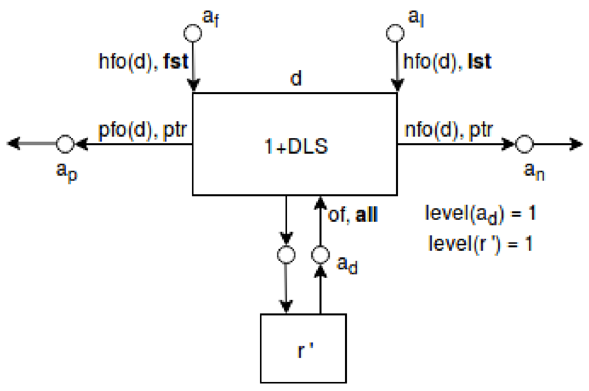

Figure 2.2: SMG with DLS and nested abstract region.

2.2.1 Properties of SMGs

Nodes. Nodes are divided into objects and values. Regarding the different object kinds, this thesis focuses on regions and doubly-linked list segments (DLSs). Another object kind is the singly-linked list segment (SLS), that however can be viewed as a restriction of a DLS and is omitted in the description for simplification. A region represents memory allocated either on the stack or on the heap. As a special region, the null object, also written as #, represents the NULL target. All NULL-pointers point to this region. A DLS is the result of merging a sequence of doubly-linked regions during heap abstraction. Values represent either addresses or data. Two values can only be distinguished by checking if their concrete values are equal or different. The only special value is the value 0, which represents sequences of zero bytes of any length, including nullified blocks of any size, and is used as the address of the null object.

Edges. There are two kinds of edges, one that leads from objects to values, and one that leads from values to objects. Has-value edges lead from objects to values and express that an object stores a value. The has-value edge is labelled by an offset and a type, which determine at which offset the field lies in which the value is stored and of which type this field is. Fields are allowed to overlap but they must lie within the boundaries of the object. Points-to edges lead from values to objects and express that an address points to an object. The points-to edge is labelled by an offset and a target specifier. The address can point before, inside or behind the object. The target specifier specifies if the target of the pointer is either a region (reg), the first element of a list (fst), the last element of a list (lst) or if each element of a top-level list is pointed from

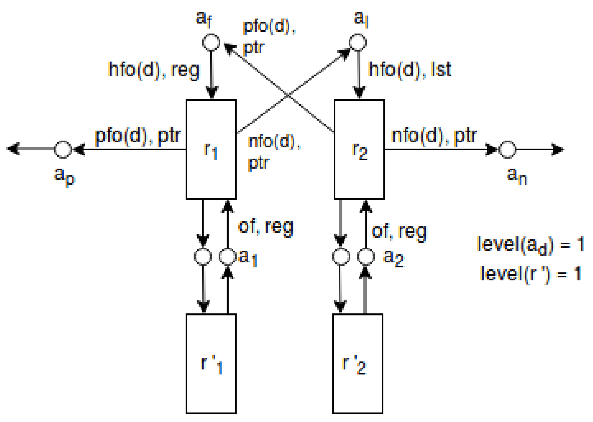

Figure 2.3: Possible concretisation of SMG with DLS and nested abstract region.

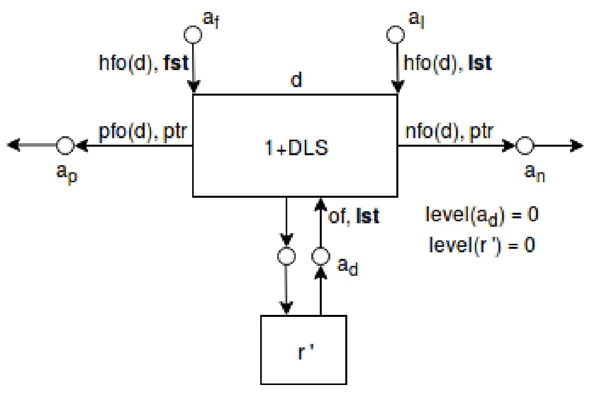

Figure 2.4: SMG with DLS and shared region.

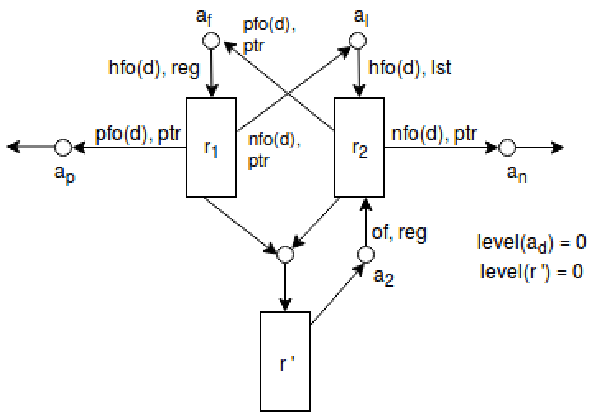

Figure 2.5: Possible concretisation of SMG with DLS and shared region.

its nested list (all). An example for the use of the all specifier is depicted in figure 2.2, where a DLS d is shown that has a nested region r' which stores a pointer that points back to d with the target specifier set to all. In figure 2.3 a possible concretisation of d is shown. If r' is not a nested region but on the same level as d, then the concretised regions of the DLS both point to the shared object r' as depicted in figures 2.4 and 2.5. Because r' is not nested, we also cannot use the all target specifier but let r' point to the last list element.

Attributes. Nodes are labelled by several attributes. Each object is labelled by its kind, size, nesting level and validity. DLSs additionally are labelled by their minimum length, their head offset and their next and prev field offsets. Values are labelled by their nesting level only. The object kind is one of {reg, dls}, the null object is defined as a region. The size of a region is the amount of memory allocated for it. The size of a DLS is the size of one of its nodes or the size of one of the regions that were merged into the DLS. The nesting level specifies at which hierarchical level the object or value appears, whereas 0 is the top level and 1 is the level of objects or values nested below it. The validity of the null object and of deallocated regions is false.

2.2.2 SMG Consistency

Consistency Rules An SMG is called a consistent SMG, iff the following consistencies hold:

- Basic object consistency:
  - The null object is invalid, has size, level and address 0.

  - All DLSs are valid.
  - Invalid regions have no outgoing edges.

- Field consistency:
  - Fields do not exceed boundaries of objects.

- DLS consistency:
  - Each DLS has a next and a prev pointer
  - The next pointer is always stored in memory before the prev pointer
  - The offsets of points-to edges to the first and last node of a DLS must be equal to the head offset of the DLS
  - There is no cyclic path of only 0+DLSs and their addresses

- Nesting consistency:
  - Each nested object of level x has exactly one parent DLS with level x - 1
  - There must be a path from the parent object to its nested object
  - The inner nodes of a nested object on level x must be of level x or higher
  - Addresses with target specifier fst, lst or reg are always of the same level as the object they point to
  - Addresses with the all target are one level higher than the object they point to
  - Points-to edges to a DLS that have the all target specifier can only lead from objects nested below that DLS

Examples of Inconsistent SMGs If in figure 2.2 we only change the level of r' to 0, we create an inconsistent SMG because the last condition for nesting consistency is violated. The violation exists because the all target specifier is used in the points-to edge to d, which demands r' to be a nested object of d. Therefore, we change the all target specifier to the lst target specifier. We also could use the fst target specifier. The consequence of this change is that the SMG is still inconsistent because the fourth condition for nesting consistency is violated because address a_d has not the same level as the object it points to. Therefore, the next change is to set the level of a_d to 0, which is the same level as of DLS d. After this change we obtain a consistent SMG as in figure 2.4.

2.2.3 SMG List Abstraction

Join of SMGs. The join of SMGs is a binary operation that takes two SMGs G1 and G2 and returns their common generalisation, the SMG G. If both SMGs are semantically equal, G is semantically equal to both the input SMGs and the join status is $\simeq$. If G1 is a generalisation of G2, G is required to be semantically equal to G1, and the join status is $\sqsupset$. In the symmetric case, if G2 is a generalisation of G1, G and G2 must be semantically equal and the join status is $\sqsubset$. If neither G1 nor G2 is a generalisation of the other SMG, it must only hold, that G is a common generalisation of both SMGs, and the join status is $\bowtie$.

List Abstraction Algorithm. SMG List Abstraction uses a candidate approach, such that at first the valid list candidates are searched on the heap and then these candidates are processed in an order that is cost-efficient, whereas high cost corresponds to a big loss of precision. The DLL finder, that is looked at more closely in the next chapter, will find a set of abstraction candidates that store the actual list candidates together with their detected length and join status. After each time the DLL finder finds that the current object is properly linked to the next object, it will increment the length of the list candidate for the corresponding join status. The join status is found by executing a join on the sub-SMGs of the two objects. Note however, that in this case the join is only used to check if the objects are mergeable and for recording the join status, the join itself is discarded after that. After the DLL finder traversal has finished detecting lists on the heap, the returned set of abstraction candidates is checked for valid candidates, which are the ones, that have a greater length than the length threshold based on the join status. There are three different thresholds, one for equality, one for entailment and one for incomparability. For instance Dudka et al. found (equalityThreshold = 2, entailThreshold = 2, incomparableThreshold = 3) to be a good configuration. After obtaining the final set of valid abstraction candidates, it can now be searched for the best candidate, i.e. the candidate that achieves the lowest cost, or in case of ambiguity, the one with the greatest length or a random member of the best group. In CPAchecker [4] a score is computed, where an equality join status gives 50 points, an entailment status gives 30 or 31 points and an incomparability status gives no points. In CPAchecker also the length of the list is added to the score and if the candidate has recursive fields or a list is already included in the candidate, the score gets another small increase. The higher the score, the lesser cost and the lesser the precision loss. The best candidate then gets abstracted by iteratively merging the first two elements of the candidate making the merge result the first object in the next iteration

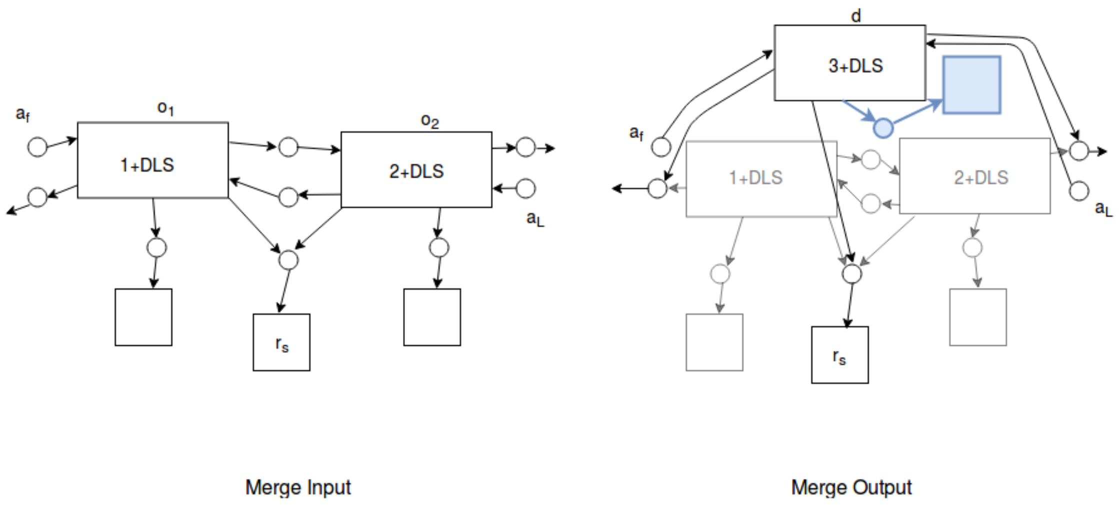

Figure 2.6: Merging two neighbouring objects.

until all elements have been merged.

Elementary Merge Operation. Each of these single merge operations is performed as in figure 2.6. At first, objects o_1 and o_2 are merged, resulting in the DLS d, which has a minimum length that is the sum of the minimum lengths of the merged DLSs. Then the sub-SMG is created by joining the sub-SMGs of o_1 and o_2. In the join algorithm used for abstraction, two simultaneous searches are started from o_1 and o_2 which are part of the same SMG. As the searches are done on the same SMG, they can arrive at the same object at the same time. In figure 2.6 the object r_s is found to be a shared object reachable from both objects and is therefore not included in the resulting sub-SMG, which is colored blue in the figure. After obtaining the merged DLS and it s new sub-SMG all pointers that pointed to the start of o_1 or to the end of o_2 before the merge have to be redirected to d using the target specifier fst or respectively the target specifier lst. In the end, all objects and values on the heap, that are not reachable from the stack anymore are removed from the SMG as well as all adjacent edges.

Chapter 3

SMG Analysis in CPAchecker

3.1 Erroneous Candidates for Circular DLLs

The detection of a list candidate of length 5 for a circular doubly linked list of length 4 depicted in figure 3.1 is obviously wrong. However, this bug was disguised in two different ways:

- In the later actual merging of list candidates, the incremental fundamental merge operation as described in 2.2.3 is done in a loop over the length of the candidate. This is fine, if the candidates are computed correctly, but in the case of a candidate that represents a circular list, the continued abstraction of the DLS with the next object, which is in the last step the DLS itself, leads in the case of the candidate of length 5 to a DLS with minimum length 8, because the DLS of length 4 is merged with itself. To not let this happen again, a check was inserted, which in the case of identity of the two objects that are to be merged, triggers an exception such that the programmer can be made aware of it.
- Because the score which was described in 2.2.3 is dominated by the join status, even if the list is longer than it should be, there is still a chance that it will not be merged such that the error stays undetected.

.

Starting the traversal for node N0. Following a trimmed-down version of the DLL finder algorithm in listing 3.1, the traversal starts with the first object on the heap, in this case N0. The algorithm maintains a data structure which is called the join progress, which contains a candidate map that maps start objects and an offset pair to their list candidates. The join progress also maintains a candidate length map which maps candidates to their length.

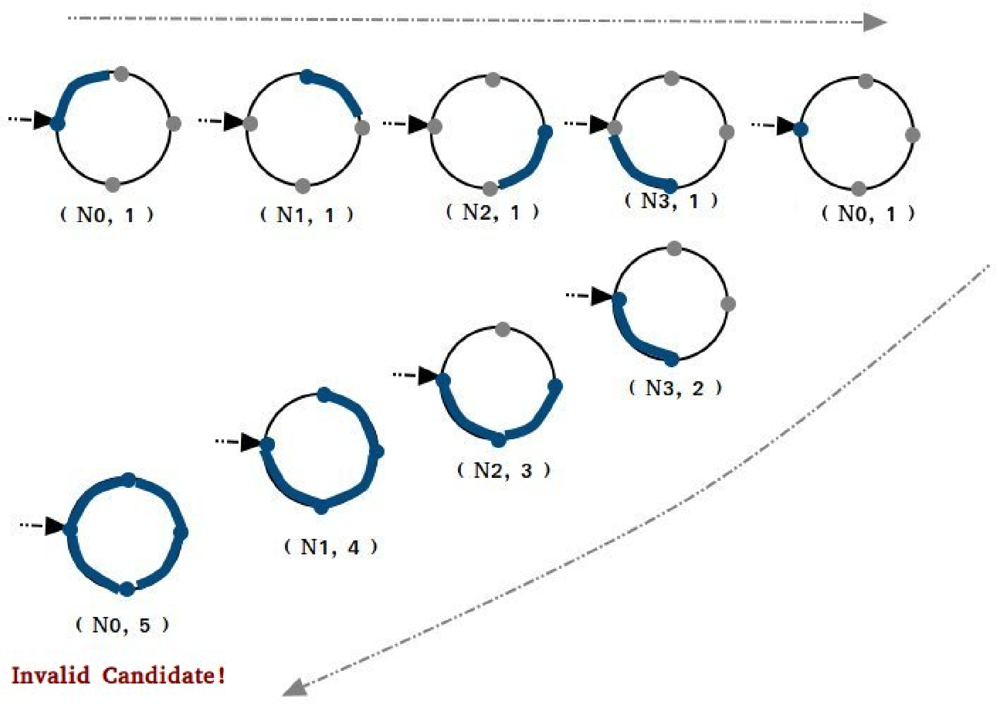

Figure 3.1: Erroneous traversal of a circular DLL.

When the traversal for N0 starts, the algorithm checks at first in its progress if there is already a candidate map for this object. If this is the case, the traversal would finish for N0 and the algorithm would continue with the next heap object. In this case N0 has no candidate map and the algorithm proceeds to create candidates for the object. It proceeds with several checks on the object, being a valid object, having the same size and a fitting object kind. N0 can only be part of a DLL if itself is a DLL or a region, in other cases the traversal stops. Then the has-value edges are checked, and if the number is lower than 2, the object cannot be part of a DLL, because the minimum requirement of a prev and a next edge. Then, each has-value edge is considered as the next pointer edge, the object pointed by it is checked to have the same kind, size and level like N0 and then the has-value edges are checked in order to find N0 by a possible prev pointer edge. If all checks succeed for an object and an offset, an initial list candidate with length 1 is created for N0 and inserted into the candidate map. This list length candidate is depicted in the upper left part of figure 3.1.

Continue traversal for N1 and the current candidate. The traversal now continues directly with the next object which is N1. After this call finishes, the traversal for N0 will finish too and the next heap object is considered. As N1 is not in the candidate map yet, the algorithm starts a new traversal for N1. This will trigger traversals for N2 and N3 in the same way. The next object of N3 is N0. After creating the candidate of length 1 for N3, the traversal continues with N0 and this candidate. The algorithm will find N0 in the candidate map and will not start a new traversal for it, but will retrieve the candidate for N0.

Missing check if first and last objects collide. At this point the traversal should stop for N0, but it does not stop and proceeds to update the previous candidate, which is the candidate with start object N3 and length 1 which is updated to length 2. The recursion then returns to the traversal of N3, which leads to an increment of the previous candidate length again, resulting in (N2, 3). Then the traversal of N2 continues and at last the one of N1, eventually producing a candidate with length 5, which is wrong because the list only has four nodes.

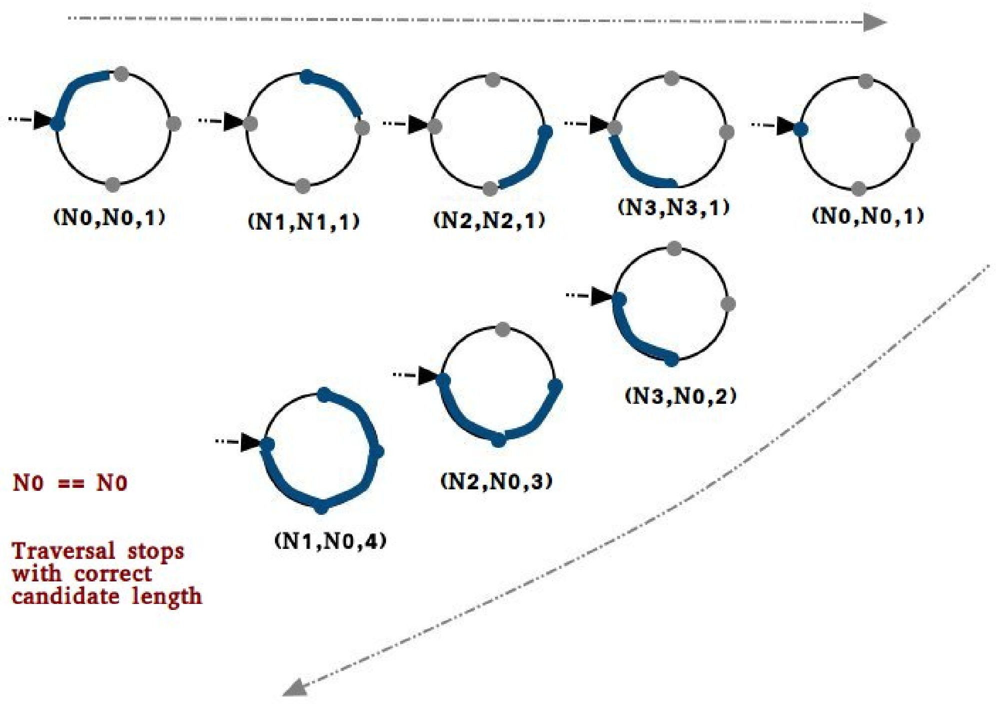

Figure 3.2: Improved traversal using last object as stop condition.

Improving the DLL finder algorithm. The problem in the algorithm which is shown in listing 3.1 is that there is no way to check if an object is already responsible for an increment of a list candidate length in the case of circular DLLs. If the length is incremented again for the same node, the candidate is wrong and can lead to unnoticed errors in abstraction. By letting a doubly linked list candidate additionally store the last object, the algorithm has a possiblity to check if the previous candidate is going to be linked to a candidate that has the previous candidate already as its last object. This way overly long list candidates can be prevented.

```
for(object : smg.getObjects()) {
    startTraversal(object, progress);
}
return progress.getValidCandidates(threshold);
```

Listing 3.1: Simplified DLL finder algorithm

```
if(progress.hasCandidates(object)) {
    return;
}
(nextObject,nfo,pfo)= checkDoublyLinked(object, smg.getHVEdges());
candidate = Candidate(object, (nfo,pfo), 1);
progress.addCandidate(candidate);
continueTraversal(nextObject, candidate);
```

Listing 3.2: startTraversal

```
if(!progress.hasCandidates(nextObject)) {
    startTraversal(nextObject);
}
candidate = progress.getCandidate(nextObject, (nfo, pfo));
checkMergeability(startObject, nextObject);
// prevent merge at circular link
if(candidate.getLastObject() == previousCandidate.getStartObject()
    ) {
    return;
}
if(isLastInSequence(nextObject)) {
    progress.updateLength(previousCandidate, 2);
} else {
    progress.updateLength(previousCandidate, candidate.length+1);
}
// update last object of previous candidate
previousCandidate.updateLastObject(candidate.getLastObject());
```

Listing 3.3: continueTraversal

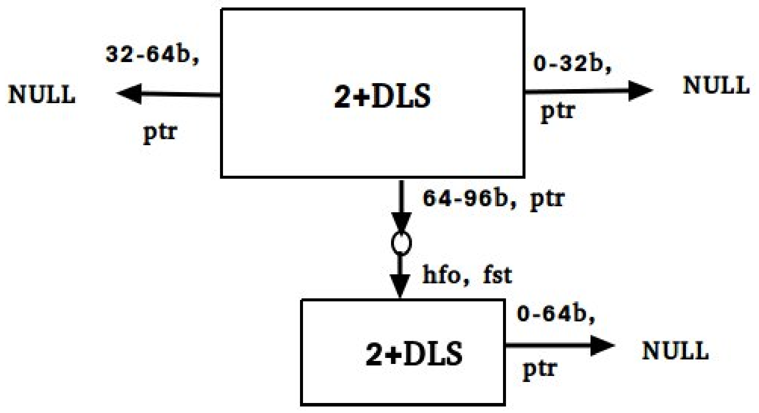

Figure 3.3: One summarized has value edge for a nullified region of 64 bytes.

3.2 Materialisation of Nullified DLL Segments

As depicted in figure 3.3 the result of joining nullified fields of an SMG object can lead to temporary removal of link edges like the missing prev edge of the DLL in the lower part of the figure. Because the offsets are still stored in the shape of the abstract list and the type of the field is known to be a pointer to a DLL, the missing link can be retrieved. However, after the edge is removed from the set of has value edges of the SMG, program code, that retrieves has value edges from this edge set can fail, if it does not account for the possibility of joined nullified fields. This was the case in CPAchecker for the materialisation procedures, which in a materialisation of a segment like in the figure, tried to retrieve the prev edge of the DLL from the set of has value edges, where it was not found which led to an exception.

Hidden link edges. According to page 8 in the technical report of Dudka et al. [7], for an SMG to be consistent, DLS consistency must hold, which states that the next offset is never greater than the prev offset. There is accordingly a check for this consistency condition in the list finder algorithms, which otherwise stops the traversal. It is possible that both link edges will be hidden by joining nullified fields, if the respective object has a nullified field directly before the two link fields. Only the next field edge is hidden if there is a nullified field directly before it and one or more non-nullified fields directly before the prev field edge. Only the prev field edge is hidden if there is a nullified field directly before it, and no nullified field directly before the next field edge. Note that a link field cannot be hidden if itself is not nullified.

Solution by using read reinterpretation operator. There are already operators implemented that reinterprete nullified memory and retrieve link edges that are missing due to a previous field join. By using the read reinterpretation operators to read the value of a field, exceptions of missing edges can not occur.

3.3 Value Replacement of Abstracted Segments

For some programs the SMG analysis reported an inconsistent SMG due to a region, that was stored in an address value. The object present in an address value represents the memory of the address. As explained in section 2.2.3, after the elementary merge operation has finished, all pointers that pointed to the first and last object before the merge have to be redirected to the newly abstracted DLS. However, if the first or the last object of the sequence was a region, and was pointed to by an address value, the address value will then point to the DLS but still store the reference to the old region, as depicted in figure 3.4. In this way the redirection of the pointers introduces inconsistency in the SMG, because the region could still be used even if it is not explicitely in the SMG anymore. To avoid this, pointers that are address values are now replaced by fresh symbolic values leading to figure 3.5.

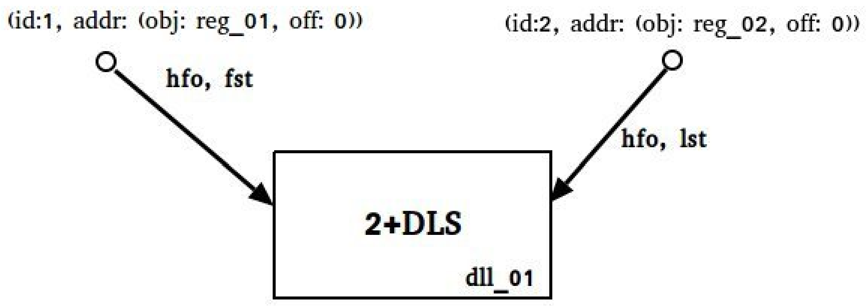
(id:1, addr: (obj: reg_01, off: 0))
(id:2, addr: (obj: reg_02, off: 0))
hfo, fst
hfo, lst
2+DLS
dll_01

Figure 3.4: Known Address Values store an id and an address.

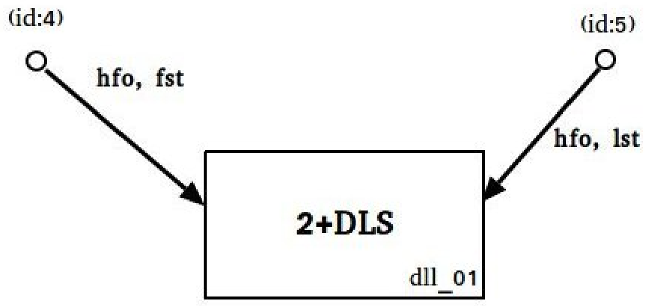

Figure 3.5: Symbolic Values only store an id.

Chapter 4

Symbolic Heap Abstraction with Automatic Refinement

4.1 Motivating Examples

4.1.1 Symbolic Heap Abstraction

Listing 4.1 shows a motivating example for using abstract list segments in SMG analysis. In the loop starting in line 5 a nondeterministic number of freshly allocated DLL nodes is appended to each other and all nodes are initialized with the same data value. In the following traversal starting in line 10, the data of the list is checked and if a node's value is found to be different from the initial value, an error function is called. (Note that due to line 14 the second pointer to the list is necessary to keep track of allocated memory.)

```
 1 void example_that_requires_abstraction() {
 2     int data = 1;
 3     // create list of nondeterministic length
 4     DLL s = NULL;
 5     while(nondet()) {
 6         append(&s, data);
 7     }
 8     // check data of all elements
 9     DLL ptr = s;
10     while(ptr) {
11         if(data != ptr->data) {
12             error();
13         }
14         ptr = ptr->next;
15     }
16 }
```

Listing 4.1: Motivating example for using abstraction

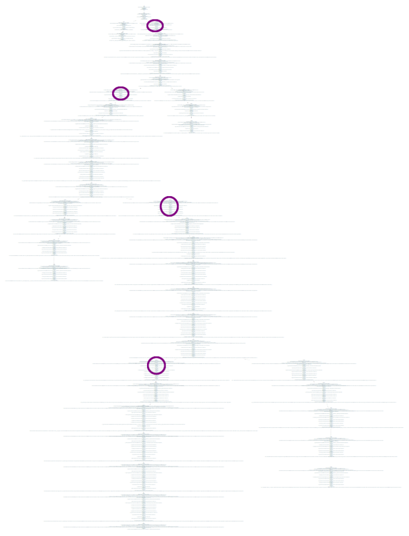
Figure 4.1: Truncated simplified ARG for SMG analysis without abstraction on listing 4.1

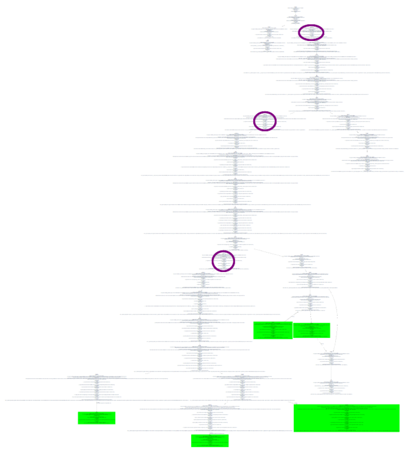
Figure 4.2: Simplified ARG for SMG analysis with abstraction on listing 4.1

Without abstraction. Running an SMG analysis without heap abstraction on this input results in non-termination because each time a new node is appended to the list in line 6, a fresh region is created on the heap resulting in a fresh SMG state that is not covered by the reached set. Thus line 6 will be revisited infinitely often, such that the analysis never comes to the conclusion that the error function in line 12 cannot be reached. In figure 4.1 the upper part of the resulting ARG is shown. The circled states mark the states which are reached whenever the append function entry is examined. We can see that each time a new state is created and no coverage relations exist.

With abstraction. When instead using SMG analysis with heap abstraction on the example in listing 4.1, the analysis terminates with the correct result. In the loop starting at line 5, the analysis will detect a list structure when the specified length threshold is reached for some list candidate. By appending further nodes to the resulting abstract list segment, only the minimum length of the segment can increase but the type and number of objects on the heap stay constant. If two states only differ in the minimum length of a list segment, the state with the list with the higher value is covered by the other state, which means that a fixed point will be reached eventually and the analysis of the loop will terminate.

Figure 4.2 shows the resulting finite ARG when using a heap abstraction threshold of two, which means that lists are detected and abstracted at length two and higher. The circles mark the states reached at each entry of the append function and green states mark states that are covered by another state. The append function is entered only three times in this case. After the second time the list gets abstracted to an abstract list segment. The third time then symbolizes the case when a node is appended to a list, such that another entry of the append function is not needed anymore in the analysis because from this point on the loop just keeps appending nodes to a list. Following the path in the ARG after the third append, the resulting list of minimum length three could also represent a list of four or more nodes due to the concept of minimum length. For this case there is also a state in the ARG, the second green state from the left. The path ends with this state, because it is already covered by the state with the list of minimum length three, which is the left-most green state in the graph.

4.1.2 CEGAR-based Approach using Maximum Length

In listing 4.1 a fixed global heap abstraction threshold had to be specified to run SMG analyis with abstraction. To keep track of as much structure on the heap as possible, a small length threshold like two or three seems like the proper choice. The reasoning for that is that a small threshold leads to a higher probability that lists are abstracted to segments. However, there are cases where the abstraction of a data structure is not the best option. An example is when there is a part in the code that aggregates features of a data structure, for example using aggregate functions like sum, count or average, and an error condition depends on the knowledge of the exact number of elements of a data structure. In this case forgetting the exact length by abstraction has to be avoided.

Listing 4.2 gives an example where the call of an error function depends on the precise count of list elements. In the loop starting in line 6, a list of fixed length five is created. Then, in the next loop starting in line 13, the elements of the list are counted and in each iteration a check is done if the count has exceeded five. Another example instead of the element count would be if the error condition depends on the exact sum of the data stored in all list elements.

```
1 void example_that_requires_precision() {
2     int data = 1;
3     // create list of length 5
4     DLL s = NULL;
5      int i = 0;
6      while(i < 5) {
7          append(&s, data);
8          i++;
9      }
10     // check length of list
11     i = 0;
12     DLL ptr = s;
13     while(ptr) {
14         ptr = ptr->next;
15         i++;
16         if(i > 5) {
17             error();
18         }
19     }
20 }
```

Listing 4.2: Example showing limitations of a fixed abstraction threshold

Running SMG analysis with a heap abstraction threshold of less than 6 on this example leads to a false result, because the error function in line 17 is reachable. Using a threshold of 6 or higher leads to the correct result. The same correct result is produced when using no abstraction at all, which raises the question if the approach to use no abstraction whenever such a case is detected, would be sufficient. However, this is not the case as the following example shows. By combining the last two listings in listing 4.3 an example is given where SMG analysis without abstraction does not terminate due to the reasons described in section 4.1.1, and by using SMG analysis with heap abstraction wrong results are possible depending on the choice of the heap abstraction threshold.

```
void example_that_requires_refinement() {
    example_that_requires_abstraction();
    example_that_requires_precision();
}
```

Listing 4.3: Motivating example for refining the abstraction threshold

A solution is to use a CEGAR-based approach and start the analysis with a minimal abstraction threshold, that can be refined in case of a spurious counterexample. By inspecting the error path and finding out the maximum length of all lists that can be detected at each path position, a threshold can be found that guarantees that no list on the path will be abstracted such that the error paths due to too coarse abstractions can be refuted.

4.1.3 Threshold Candidate Generation

Using Maximum Length. When a counterexample is found by the analysis, it has to be checked for feasibility to find out if the analysis can stop with a false result in case of a feasible counterexample or continue analysis after refinement of the counterexample. This check can be done by inspecting the error path without heap abstraction and checking if the target state is then reachable or not. During this inspection the heap of each encountered SMG state can be used as input for the SLL and DLL finders in order to find all abstraction candidates and their lengths. This way the maximum list length that can occur on the error path can be found and can be used as an over-approximation of the length that is necessary to avoid the list abstractions on the path that are the cause for the counterexample.

```
// create two lists with different lengths
DLL s = create_dll(5);
DLL t = create_dll(20);
// check length of shorter list
int i = 0;
DLL ptr = s;
while(ptr) {
    ptr = ptr->next;
    i++;
    if(i > 5) {
        error();
    }
}
```

Listing 4.4: Motivating example for using maximum length candidates

Using Object-Length Mapping. Listing 4.4 gives an example where the maximum of all encountered list lengths does not give the optimal threshold. In this example the threshold found would be 21, because 20 is the maximum list length found on the error path, but the optimal threshold would be 6, because it would suffice to refute the counterexample in an SMG analysis using abstraction. In general, if other lists than the ones responsible for the counterexample are created on the error path, the overall maximum length can be much higher than the actual length necessary to refute the counterexample. If there can be found a mapping between start objects of list candidates and the corresponding maximum encountered list length, the resulting lengths can be used as candidates for a threshold that is as small as possible and as high as necessary of all lengths in the candidate set. The candidates can then be sorted in ascending order and tried as threshold by rechecking if the target is reachable by using abstraction with the current threshold. As soon as the target is not reachable anymore, the most recent threshold can be used for refinement.

4.2 Refinement of Heap Abstraction Threshold

If the analysis finds a target state, the CEGAR algorithm will trigger the refinement of the reached set. The refiner will then make a cut at a specific state in the abstract reachability tree and remove this state and its subtree. Then it will readd states with a new precision to the waitlist. It will readd parents of children that were removed and of children that were covered by removed states because they may not be covered anymore. In both of the following approaches, the state at which position to remove the subtree as well as the new precision for states to be readded have to be found. Both approaches assume that $\text{merge}_{\text{sep}}$ is used as merge operator in the SMG CPA, such that each target has only one target path. Also, as the analysis stops after a target state is reached, there is only one target state in each refinement.

Cut State. Both approaches use as cut state the first state in the target path for which at least one list abstraction candidate using two as threshold is found. This state potentially contains the list that causes the target to be reachable, such that states at a later path position cannot be easily removed instead. To remove at an earlier position in the path does not make sense because no list candidates are found at these positions which would not change with another heap abstraction threshold of the new precision.

New Precision. The new precision uses the new abstraction threshold which is in both cases the increment of the found maximum length.

4.2.1 Naive Maximum Length Approach

Listing 4.5 shows a basic approach of refinement of the abstraction threshold using the maximum list length on the error path. To find out if the error path is feasible, the states encountered on the path have to be considered. For each state all abstraction candidates are collected by calling the SLL and DLL finders which detect list structures on the heap by systematically following next and prev links of SMG regions. The maximum length is updated each time a list candidate is found that is longer than the previous longest list. The first time that a list candidate is encountered, the cut state is set to the current state and keeps unchanged for the rest of the refinement. If the target state is reached, the path is feasible and reported to the CEGAR algorithm. If however, a state which is not the target, has no successor state, the path is found infeasible or spurious and a refinement is started. In this case the cut state and its subtree are removed from the reached set and the parents of removed children and of children that were covered by removed elements are readded to the waitlist with a new precision that contains the increment of the found maximum length as abstraction threshold. The counterexample is then reported to CEGAR as spurious.

```
for(state : errorPath.getStates()) {
    listCandidates = state.getAllCandidates(dllFinder, sllFinder);
    maxLen = max(maxLen, listCandidates.getMaxLen());
    if(cutState == null && maxLen > 0) {
        cutState = state;
    }
    if(!state.hasSuccessor() && !state.isTarget()) {
        reached.removeSubtree(cutState, SMGPrecision(maxLen + 1));
        return "spurious";
    }
}
return "feasible";
```

Listing 4.5: Naive refinement

4.2.2 Threshold Candidate Approach

As seen in listing 4.4, there are examples where the maximum length over all list candidates on a path is not the best threshold to use. Listing 4.6 shows an approach where a mapping from start objects of lists to their maximum encountered length is used to keep track of more than one list length such that a list of several threshold candidates can be produced from it if refinement is necessary. In line 3, the current SMG state is used to update the mapping. In line 5, the cut state is set to the first state for which the mapping is not empty, which means that it is the first state for which there is a list candidate on the heap, which is done similarly in the naive refinement. In the case of a non-target state which does not have a successor, refinement is necessary. In the loop starting in line 8, a feasibility check uses heap abstraction with the current thresholdCandidate until the error path is not found to be feasible. In this case, the best threshold was found and the refinement of the ARG can be executed with the new precision. Also the counterexample can be reported as spurious.

```
1 mapping = Map.empty() // maps start objects to list lengths
2 for(state : errorPath.getStates()) {
3     mapping = updateMapping(mapping, state);
```

```
    if(cutState == null && !mapping.isEmpty()) {
        cutState = state;
    }
    if(!state.hasSuccessor() && !state.isTarget()) {
        for(threshold : getSortedCandidates(mapping)) {
            if(!feasibleWithAbstraction(errorPath, threshold)) {
                reached.removeSubtree(cutState, SMGPrecision(
    threshold));
                return "spurious";
            }
        }
    }
}
return "feasible";
```

Listing 4.6: Threshold candidate refinement

Object-Length Mapping. In order to keep track of more than one list length, there needs to be some way to map lists to their maximum lengths. The idea is to use the start object of a list candidate for this mapping. Over the course of a program path the same start object can however belong to different lists. Also it is not guaranteed that a list has the same start object over a series of states. For example if a circular doubly linked list, for which the list pointer moves to the next node, is newly abstracted, the new abstraction can have another start object although the list nodes have not changed at all. This can lead to many more threshold candidates than necessary. In the following, a partial mapping from start objects to the maximum encountered length is maintained. If only a set of encountered lengths would be kept instead, there would likely be very man of them. For example, if a list of 5 nodes is built, the set would contain all values from 2 to 5, because they could all represent the maximum encountered lengths of different lists. By using a map instead, we hope to achieve a result for this example that has only one entry which maps the start object of the list to the maximum encountered length 5.

Heuristic for exchanged start objects. To identify lists between states, that are likely the same lists, we need to look at cases where they can be detected as similar even if the start object changes. When comparing the abstractions of the heaps of two consecutive states there are cases when exactly one start object has changed. If after abstraction of a state we find a start object that is not yet in the current mapping and at the same time only one object is missing in comparison to the last abstraction, the list has likely changed its start object. In this case we remove the old entry and insert the new object with the maximum of the new and the old entries' lengths. To continue the example of the list with 5 elements, we can imagine that the first node of the list was removed by the program, such that the start object would change and we would get a newly encountered maximum length of 4. By using the above explained heuristic, we however keep only tracking 5 as a maximum length and update the old start object to the new start object.

The heuristic is implemented by putting the start object of an abstraction candidate in one of two maps based on it being already in the object-length mapping or not. If the mapping does not contain the startObject, it is put into tempMap in line 11. Otherwise it is put into checkMap in line 14. In this case its length is also updated, if it is greater than its previously stored maximum length. After all abstractions are done, it can be decided what to do with the entries in tempMap. In the case that there has been one change in comparison to the mapping of the last state and there is also one newly encountered start object, the old entry is removed and the new entry is inserted into the mapping with the maximum length of the two entries. This way soundness is not violated even if the assumption does not hold, because there is still kept track of the maximum length.

Keeping the threshold set small. In the cases where the above condition does not hold, all newly found entries are checked for their length. The entry is only inserted into the object-length mapping, if the length is not yet contained. This way the mapping is kept small while soundness is not violated because all encountered maximum lengths, and thus also the overall maximum length, are still present.

```
 1 updateMapping(mapping, state) {
 2     stateCopy = state.copy();
 3     do {
 4         // Execute heap abstraction on a state copy step by step
 5         abstractionCandidate = stateCopy.
     executeHeapAbstractionOneStep();
 6         startObject = abstractionCandidate.getStartObject();
 7         len = abstractionCandidate.getLength();
 8
 9         // Put new start objects into temp map
10         if(!mapping.contains(startObject)) {
11             tempMap.put(startObject, len);
12         } else {
13             // Put already seen start objects in check map
14             checkMap.put(startObject, len);
15             if(len > mapping.get(startObject)) {
16                 mapping.put(startObject, len);
17             }
18         }
```

```
    } while(!candidate.isEmpty());
    diff = Sets.difference(mapping.keys(), checkMap.keys());
    if (tempMap.size() == 1 && diff.size() == 1) {
        diffObject = getOnlyElement(diff);
        tempObject = getOnlyElement(tempMap.keys());
        mapping.put(tempObject, max(mapping.get(diffObject),
    tempMap.get(tempObject)));
        mapping.remove(diffObject);
    } else {
        for (entry : tempMap.entries()) {
            if (!mapping.values().contains(entry.getValue())) {
                mapping.put(entry);
            }
        }
    }
}
```

Listing 4.7: Update of the Object-Length Mapping

Chapter 5

Conclusion

The assumption that only one start object changes during a state change is according to first tests not very promising. However, the approaches are in need of a proper evaluation in order to make statements on the heuristic's performance. The copying of the SMG state at each path position during the feasibility check and the exhaustive use of execution of heap abstraction which is usually only used at function and loop heads gives reason to question the efficiency of the approach. There is also the possibility to use a very high number for the threshold, which could analyse programs like the first motivating example has shown and still be effectively as concrete as the pure SMG analysis without abstraction.

Chapter 6

Acknowledgments

I would like to express my appreciation to Karlheinz Friedberger for his patience and for his valuable suggestions during the course of this master thesis. Also especially for making time for meetings and answering emails whenever needed as well as reminding me of my time plan from time to time. I would also like to thank Marie-Christine Jakobs for valuable insights into CPAchecker during the practical course prior to the thesis and her willingness to help. Further thanks to Martin Spiessl as well as Michal for their offers to ask them anything I want, to Dirk Beyer, Martin Hofmann, Thomas Lemberger and Leah Neukirchen for their informative advanced course on formal specification and verification and at last to Philipp Wendler for the comprehensible introduction to available thesis topics.

List of Figures

2.1 Shape graph and possible instance of the graph. 6
2.2 SMG with DLS and nested abstract region. 8
2.3 Possible concretisation of SMG with DLS and nested abstract region. 9
2.4 SMG with DLS and shared region. 9
2.5 Possible concretisation of SMG with DLS and shared region. . 10
2.6 Merging two neighbouring objects. 13

3.1 Erroneous traversal of a circular DLL. 15
3.2 Improved traversal using last object as stop condition. 17
3.3 One summarized has value edge for a nullified region of 64 bytes. 19
3.4 Known Address Values store an id and an address. 21
3.5 Symbolic Values only store an id. 21

4.1 Truncated simplified ARG for SMG analysis without abstraction on listing 4.1 23
4.2 Simplified ARG for SMG analysis with abstraction on listing 4.1 23

Bibliography

[1] A.S. Asratian, T.M. Denley, and R. Haggkvist. Bipartite graphs and their applications. In *vol. 131, Cambridge University Press*, 1998.

[2] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. Symbolic pointer analysis for detecting memory leaks. In *FASE'05 Proceedings of the 8th international conference, held as part of the joint European Conference on Theory and Practice of Software conference on Fundamental Approaches to Software Engineering*, pages 2–18, 2005.

[3] D. Beyer, T.A. Henzinger, and G. Theoduloz. Lazy shape analysis. In *Proceedings of the 18th International Conference on Computer-Aided Verification (CAV)*, pages 532–546, 2006.

[4] D. Beyer and M.E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806*, pages 184–190, 2011.

[5] D. Beyer and S. Löwe. Explicit-state software model checking based on cegar and interpolation. In *V. Cortelessa and D. Varró (Eds.): FASE 2013, LNCS 7793*, pages 146–162, 2013.

[6] J. Condit, M. Harren, S. McPeak, G.C. Necula, and W. Weimer. Ccured in the real world. In *Proc. PLDI*, pages 232–244, 2003.

[7] K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In *Logozzo F, Fähndrich M (eds) SAS, Lecture Notes in Computer Science, vol 7935, Springer*, pages 215–237, 2013.

[8] R. Ghija and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96 Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, 1996.

[9] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proc. SPIN, LNCS 2648*, pages 235–239, 2003.

[10] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *Static Analysis, 14th International Symposium, SAS 2007*, pages 419–436, 2007.

[11] G.C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proc. POPL*, pages 49–61, 1995.

[12] B. Scholz, J. Blieberger, and T. Fahringer. Symbolic pointer analysis for detecting memory leaks. In *PEPM '00 Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 104–113, 1999.