dsOli: Data Structure Operation Location and Identification

David H. White
Software Technologies Group
University of Bamberg, Germany
david.white@swt-bamberg.de

ABSTRACT

Comprehension of C programs can be a difficult task, especially when they contain pointer-based dynamic data structures. This paper describes our tool dsOli which aims to simplify this problem by automatically locating and identifying data structure operations in C programs, such as inserting into a singly linked list. The approach is based on a dynamic analysis that seeks to identify functional units in a program by observing repetitive temporal patterns caused by multiple invocations of code fragments. The behaviour of these functional units is then classified by matching the associated heap states against templates describing common data structure operations. The analysis results are available to the user via XML output, and can also be viewed using an intuitive GUI which overlays the learnt information on the program source code.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Human Factors, Languages

Keywords

Program Comprehension, C Programs, Machine Learning, Pointer-based Dynamic Data Structures

1. INTRODUCTION

Programs making heavy use of pointers are notoriously difficult to understand and analyse, especially when the programmer is given the freedom allowed by languages such as C (see, e.g., the history of static pointer analysis [1]). For any reasonably sized program, this will inevitably mean also understanding the pointer-based dynamic data structures. In this paper we describe a prototype tool we have developed for the approach given in [5] to automatically identify such data structures and their associated operations in C programs.

The scope of this identification includes not only the structure (e.g., a singly linked list (SLL)), but also a classification of the operations that manipulate the data structure (e.g., insert to the front of an SLL), and coding style (e.g. null-termination or usage of an inline header node in a linked list). By identifying the operations, we are also able to determine the behaviour of the data structure as a whole, e.g., an SLL used as a stack or queue. The GUI component of the tool allows the information learnt to be overlayed on the source code of the program under analysis, such that the user can quickly locate and identify data structure operations, and in addition visualize the affected pointer relationships.

The approach underlying the tool is based on dynamic analysis; the program is instrumented such that the sequence of memory states observed during an execution can be recovered. By using techniques from machine learning, we search in this sequence for repeating temporal patterns of, e.g. pointer writes, that are caused by multiple invocations of code fragments. If any data structure operations were executed, then some of these patterns will represent code fragments of the operations. To identify and classify these, we employ template matching to examine the memory transformation due to the code fragment.

While the current approach requires a program's source code, we intend to extend it to binary programs and in the limit, obfuscated programs. In these future scenarios, the utility provided by this type of analysis will be significantly greater. This is one of the reasons why we locate operations purely by repetition rather than employing some domain knowledge. Essentially, we do not require that the syntactic functional units in a program correspond to the semantic functional units. This method also suits certain C programs; for example, in performance critical or low-level code, adhoc operations or inlined code may be preferred over elegant encapsulation. To the best of our knowledge, this is the first tool that targets program comprehension of such data structure operations in C programs. The closest related work to our approach is the tool DDT [3] which supports identification of operations on a wider range of data structures for optimization purposes but, in contrast to our work, DDT requires the interfaces to data structures to be highly structured, such as those typical in libraries.

In addition to program comprehension, we are employing the learnt information to aid verification of memory safety

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597800

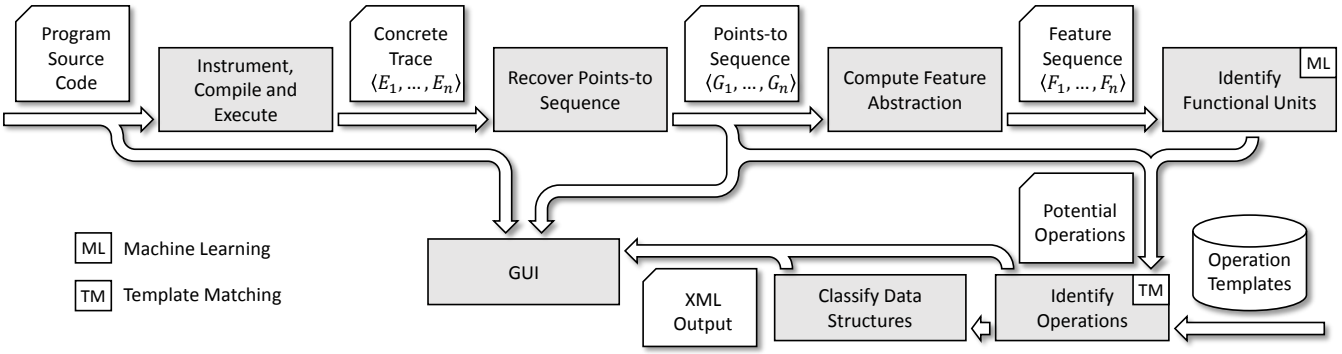

Figure 1: An overview of the approach implemented in dsOli.

properties, such as those conducted by the tool VeriFast [2]. To this end we are especially interested in the verification of device drivers, which typically use list-based structures. As such, our tool supports the identification of pointer-based data structures such as lists, queues and stacks.

2. APPROACH

An overview of our approach is given in Fig. 1. The analysis commences from a *concrete trace* obtained by executing the C program under analysis. The program is first instrumented such that this trace contains *program events* of interest to our analysis, for example, pointer writes and dynamic memory allocation/deallocation. From the concrete trace we construct a sequence of *points-to graphs*. A points-to graph is a directed graph which describes a subset of the program state; vertices correspond to heap allocated objects and pointer variables, while edges represent pointers. Specifically, the points-to graph sequence $\langle G_1, \ldots, G_n \rangle$ is computed for the event sequence $\langle E_1, \ldots, E_n \rangle$, where graph G_i describes the points-to structure after E_i is performed.

We will use the SLL data structure in the code of Fig. 3 as a running example to illustrate our approach. To capture an event sequence from executions of `insert()`, all lines with an alphabetic marker will be instrumented to record the associated pointer write. In addition, a memory allocation event will be generated for line A.

2.1 Locating Operation Invocations

We do not assume that data structure operations are well encapsulated in functions to support the usage scenarios outlined in the introduction. Thus, the next stage of the analysis is concerned with locating sub-sequences of the trace that *potentially* correspond to the invocation of data structure operations. Actually, we attempt to solve a more general problem, which is to determine the *functional units* of a program. If the data structure operations are well encapsulated by the discovered functional units, then to identify an operation's semantics it is only necessary to observe changes in the points-to graph at the boundaries between functional units. Of course we will also discover functional units that are not data structure operations, but we eliminate these by observing change, or lack of change, in the points-to graphs.

Our approach is based on the observation that programs are, by nature, highly repetitive due to function calls and iterative structures. We exploit this property to identify the functional units of a program by repetition. Indeed, any program that makes heavy use of dynamic data structures will need to invoke the interface operations many times. In addition, there are typically few control flow paths through operations (e.g., inserting to the front, middle or end of a linked list) and within those paths there are few variations (e.g., traversing over a differing number of nodes to reach an insertion point).

The difficulty is in selecting an abstraction that exposes the repetition such that we may generalize over similar code fragments, but does not overly blur the trace which would result in repetition being observed everywhere. As we are concerned with pointer-based dynamic data structures, our abstraction favours these aspects of program state. Essentially our goal is to construct a *feature vector* F_i for each points-to graph G_i, which contains elements describing a) structural properties (e.g., local connectivity changes due to a pointer write), and b) temporal properties (e.g., how the current pointer write relates to the previous one).

Continuing the running example, consider invocations of the SLL `insert()` function shown in Fig. 3 where the paths ABC, ADEEFG and ADEEEEFG are taken. Under this abstraction, these invocations would likely result in the following three sub-sequences appearing in the feature sequence: $\pi_1 = \langle F^1, F^2, F^3 \rangle$, $\pi_2 = \langle F^1, F^4, F^5, F^5, F^6, F^7 \rangle$ and $\pi_3 = \langle F^1, F^4, F^5, F^5, F^5, F^5, F^6, F^7 \rangle$ where features vectors with the same superscript have identical values. The key idea is that although the concrete addresses being operated on are different in each invocation, the local topology around those addresses and the sequence of changes remains similar, and hence recognizable. Note that a simpler abstraction could use line numbers or program counter value, however, this would fail to recognize repetition over similar/identical code fragments and would be completely unsuitable for future work on obfuscated code where the instruction memory may be in constant flux.

To search for repeating patterns we note that compression locates repetition. We evaluate compression using the *Minimum Description Length* (MDL) [5], which minimizes the size of the hypothesis summed with the size of the feature sequence encoded given that hypothesis. In our case, the hypothesis will be a *set of patterns* used to compress the sequence. As described above, control flow constructs can cause variation over invocations of functional units, so we explicitly model this by allowing the patterns to take a regular expression form. Returning to our example above, a good regular expression style pattern for locating *occurrences* of

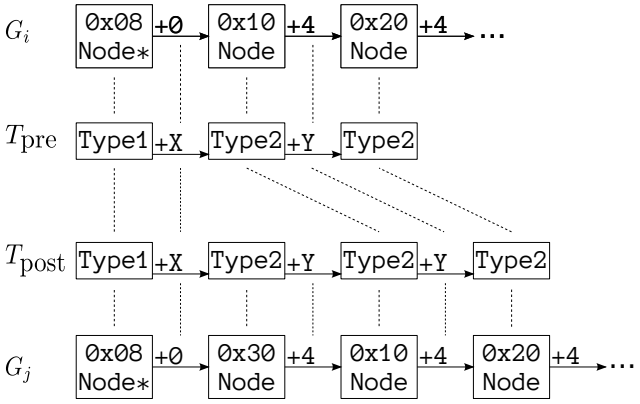

Figure 2: A template that matches inserts to the front of an SLL (T_{pre} & T_{post}) is shown matched to a potential operation (G_i, G_j) derived from an invocation of `insert` in Fig. 3.

the `insert()` function would be $F^1(F^2F^3|F^4F^{5*}F^6F^7)$. We employ a *genetic algorithm* to search for the best set of patterns where the fitness function is defined using the above MDL measure. The search is terminated when no improvement has been observed in the population for 100 generations. Mutation operators are used to insert/remove patterns from a hypothesis and alter specific patterns by extending, contracting and adding regular-expression-like operators.

2.2 Identifying Operations & Data Structures

The best set of patterns found during the search is used to tile and hence segment the points-to sequence into pairs of points-to graphs, where each pair describes program memory at the start and end of the segment. We term these pairs *potential operations*, as we must still determine the data structure operation (or absence of operation) performed during the segment.

A *template* is defined for each standard data structure operation, such as those described in a data structure textbook; the user may also manually add additional templates. For each potential operation (G_i, G_j), dsOli attempts to match each template and thus classify the operation observed between G_i and G_j. Each template consists of two graphs: T_{pre} which is matched to G_i and T_{post} which is matched to G_j; thus, a template captures the points-to transformation associated with an operation. Both matches must respect a third mapping between T_{pre} and T_{post}, which specifies the aspects of the points-to graph that must remain unchanged during the operation. We must also allow a successful template match to override other templates, this is necessary to deal with, e.g., an SLL template matching part of a doubly linked list. This is one of the ways in which we prevent false-positives. For more discussion regarding false-positives and a comprehensive evaluation, please see [5].

The template graphs in Fig. 2 describe an insertion to the front of a singly-linked list, and a match is shown with the potential operation derived from π_1 (execution of lines A, B and C from Fig. 3). Note the additional constraints imposed by the template on pointer offsets and types. This match, as well as those resulting from π_2, π_3 and others are identified and summarized near the markers ①, ② and ③ in Fig. 3.

Lastly, we use the set of operations that manipulated a data structure to determine its classification. For example, if only inserts and removals to the front of an SLL were observed, then it would be named: "SLL used as a stack".

3. TOOL IMPLEMENTATION

dsOli is divided into a number of sub-modules corresponding to Fig. 1, organized roughly in a pipeline architecture. It comprises about 20k LOC, over a number of languages introduced in the following, and took 12 person-months to develop. As the tool is in the prototype stage, efficiency is not a key goal and the analysis takes in the order of a few minutes to tens-of-minutes based on the length of the program trace and the average size of the points-to graphs. Nevertheless, it has been run on realistic examples including parts of the web server Boa (`www.boa.org`), a linked list in the Linux kernel (`www.kernel.org`) and the key-value store Redis (`www.redis.io`).

3.1 Instrumentation

We make use of the *C Intermediate Language* (CIL) [4] to perform the instrumentation step, which essentially provides a structured subset of C for the purpose of analyzing and manipulating C programs. After the input program has been transformed into CIL, we can access the CIL abstract syntax tree by the provided OCaml visitor framework. We use this to insert calls to logging functions at relevant places in the code. In addition to the previously mentioned instrumentation for pointer writes and memory allocation/deallocation, we also track local pointer variables leaving scope to keep the points-to graphs consistent. We add instrumentation to record entering/exiting basic blocks, which we use to improve the functional unit location phase. Lastly, we note that variability introduced at compile time is not an issue for our approach, since we currently only search for repetitive patterns over a single execution.

It is possible in our tool to perform selective instrumentation to reduce the amount of superfluous events in the trace, hence speeding-up and removing unimportant information from the analysis as well as allowing the user to target specific scenarios. This can be performed at the file level, at the function level and at the struct declaration level. By default, pointer writes are recorded in any struct variable that contains a self-referential field and any variable which points to such a struct. This selective instrumentation is enabled by extracting type information from the CIL abstract syntax tree.

3.2 Functional Unit Location

The genetic algorithm that searches for the set of patterns that best compresses the feature sequence is implemented in C++ using *Evolving Objects* (`eodev.sourceforge.net`). Since basic blocks have straight line control flow by definition, it makes sense to treat these as atomic, i.e., a mutation operation may never split a basic block. To speed up the search, the set of mutations that may be applied to a pattern are constrained by sub-sequences observed in the feature sequence. Due to this restriction it is possible to pre-compute the set of all possible mutations, which greatly increases the search speed. Lastly, we note that both the search and pre-computation phases are trivially parallelizable, so we enable this by default.

3.3 Identifying Operations

The templates for identifying operations are written in XML and as such are easily extendable by the user. The two sub-graphs which comprise a template state how the structure in the points-to graphs should change from before to after the operation. In addition, constraints can be placed on the concrete types of objects and the offset at which a pointer is stored in an object.

Operations are identified using Prolog. The relevant pair of points-to graphs are phrased as Prolog clauses and the template is translated into a query. The *SWI-C++* interface (`www.swi-prolog.org/pldoc/package/pl2cpp.html`) is employed to integrate the Prolog component in our application.

3.4 dsOli Output & Usage Scenarios

dsOli provides analysis results to the user via two outputs. The first is an XML file where section one describes all occurrences of operations in the trace and their classifications; this information is used by the GUI to overlay high-level operation descriptions on the source code. The second section gives instantiation information for template matches. This can be used to determine, e.g., which source code line was responsible for creating a vertex or edge that was matched by a template. This information is employed by our upcoming verification interface to automatically construct pre/post-condition annotations on the source code. The second tool output is the sequence of points-to graphs, which are described in the *dot* language (`www.graphviz.org`).

The GUI component is written in Java and uses the Swing API. It is responsible for overlaying the tool output on the source code and displaying the appropriate points-to graphs. Since these graphs may be large, we do not construct individual components in the GUI to represent each vertex and edge. Instead, an svg file is rendered from the dot file and areas of the displayed image are made click-able to allow for user interaction (e.g., to select an address).

4. GUI USAGE AND FEATURES

A snapshot of the main component of the GUI is shown in Fig. 3. A key feature of this display is the sidebar appearing to the left of the code view which summarizes the data structure operations discovered by the analysis. For each operation, a vertical bar is drawn (markers ①, ② and ③), titled with the operation's name, which delimits the code fragment that comprises the operation. A "prong" (④) sticking out from the vertical bar toward a particular line of code indicates that this line of code was involved in at least one occurrence of the operation. If there exist occurrences of the operation where differing control flow paths were taken, then these are merged and represented by a single vertical bar so long as there exists some overlap between the paths.

If the user clicks on a prong, then all occurrences of the operation for which the path included that line of code are displayed in the left-most pane (⑤). The user may then select one of the displayed occurrences (⑥) and step through the particular events which comprise that occurrence. While stepping through, the relevant prong and associated line of source code are both highlighted (④ & ⑦). Additionally, in a separate window (Fig. 4), the points-to graphs of the current and previous event are displayed. In this way, the user can easily track changes to the points-to graph while stepping through an occurrence of an operation. This is especially useful for program comprehension if it is necessary for the user to investigate the operation's behaviour in detail.

Vertices in a points-to graph which represent `struct`s are split into their component fields. For each field, its address, name and type are shown. To reduce clutter, a null pointer is shown by shading the relevant address in blue, while an undefined/untracked value is shown in gray.

To aid the user in keeping track of changes to the points-to graph, the pointer written in an event and the vertex containing that pointer are highlighted in red. In addition, any pointer writes within the operation occurrence and prior to the current write are shown in green. However, if the user wishes to track changes over several operations, this is insufficient. To solve this problem, we allow the user to select addresses to be highlighted in all subsequently viewed points-to graphs. This selection can be done by typing an address into the boxes labeled "Search" of Fig. 4, or by clicking on an address. This set of addresses also allows the user to automatically zoom the view based on the selected addresses, which is especially useful for large points-to graphs.

Finally, for any pointer shown in a points-to graph, the user may click on it and select "source location", which will highlight the line of source code responsible for setting this pointer.

5. CONCLUSION

dsOli enhances comprehension of C programs by allowing the user to automatically discover and identify pointer-based dynamic data structure operations. The results of the tool can be used directly by consulting the XML output or by visualizing them using the GUI. Future work concerns improving the approach and tool to handle programs for which the source code is unavailable, extending the variety of pointer-based data structures that may be discovered, leveraging the analysis output for other domains such as pointer verification, and improving the scalability.

6. ACKNOWLEDGEMENTS

The author wishes to thank Felix Härer and Steffen Witt for developing the GUI component and Gerald Lüttgen for his input on the approach. Part of this work was supported under grant no. LU 1748/4-1 from the DFG.

7. REFERENCES

[1] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM, 2001.

[2] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and java. In *NFM*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

[3] C. Jung and N. Clark. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO*, pages 56–66. ACM, 2009.

[4] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[5] D. H. White and G. LÃijttgen. Identifying dynamic data structures by learning evolving patterns in memory. In *TACAS*, volume 7795 of *LNCS*, pages 354–369. Springer, 2013.

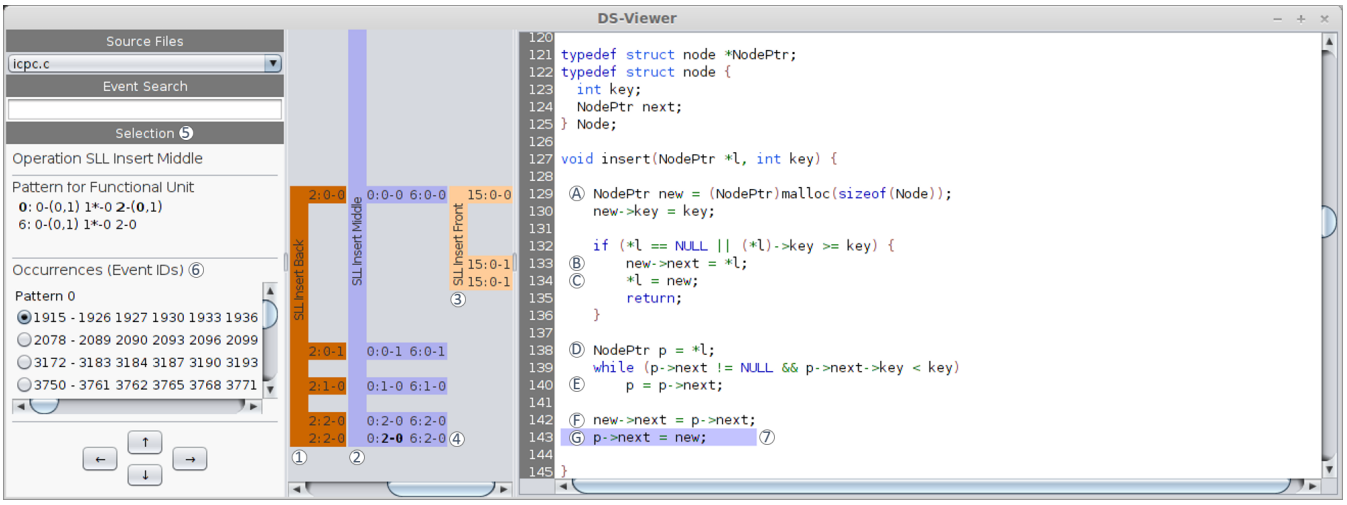

Figure 3: The code view of the GUI with operation summaries shown in the middle (markers ①, ② and ③) and occurrences on the left (⑤). Note the highlighted source code line ⑦ as the user steps through an occurrence ⑥ of SLL Insert Middle ②. The vertical bar at marker ② extending upwards out of view is due to the functional unit location algorithm grouping a small additional part of the program with some invocations of this operation, which is a natural artifact of our approximate location algorithm. Nevertheless, this imprecise yet powerful technique still allows operations to typically be correctly identified, as is the case here. For additional discussion on this, please consult [5].

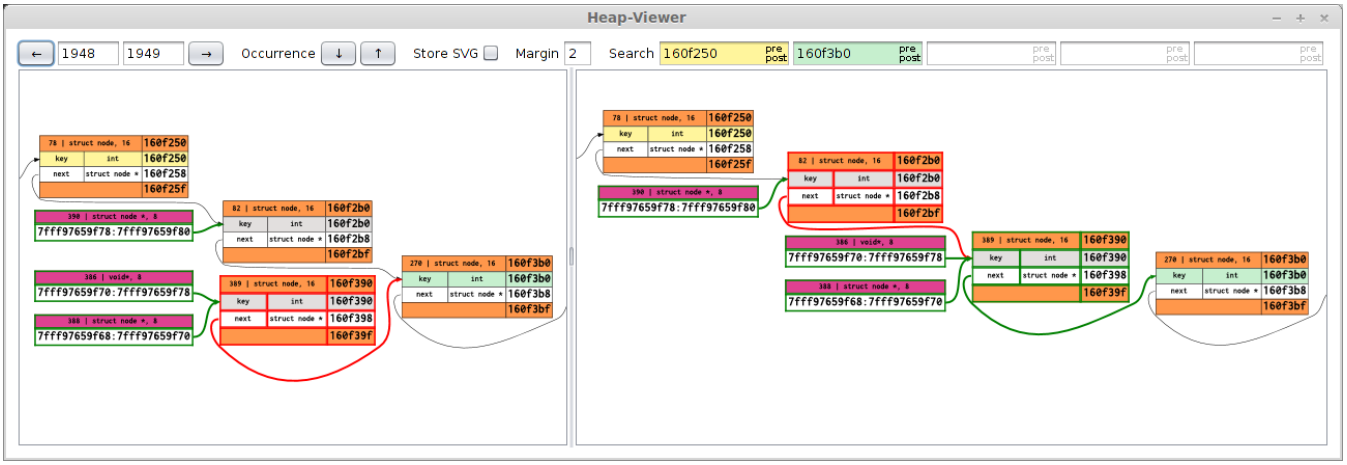

Figure 4: The secondary GUI window showing the points-to graphs for both the current event (right) and previous event (left). The current event was due to the execution of line 143 in Fig. 3, which had the effect of making a connection between the newly allocated node and the node prior to its insertion position. Note the two selected addresses in the boxes at the top middle (shown in yellow and green), which have been used to automatically zoom the relevant section of the points-to graph. When viewed in color, orange indicates heap allocated memory and pink indicates stack allocated memory. A red highlight shows the pointer and object changed in a specific event, while a green highlight shows prior changes to pointers and objects during the occurrence currently being viewed.