dsOli: Data Structure Operation Location and Identification

David White, University of Bamberg, Germany

Abstract

Comprehension of C programs can be a difficult task, especially when they contain pointer-based dynamic data structures. This poster describes our research into simplifying this problem by automatically identifying such data structures, e.g., a singly linked list (SLL) and its associated operations such as insertions and deletions.

Our 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 a code fragment. The behavior of these functional units is then classified by matching the associated heap states against templates describing common data structure operations.

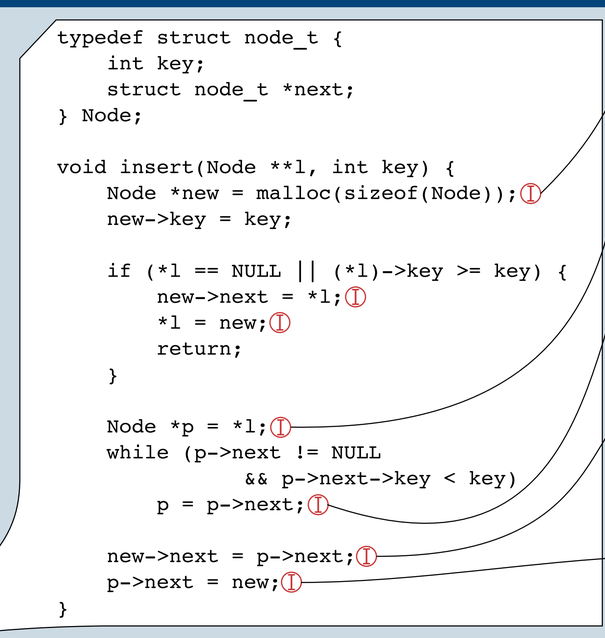
The analysis results are available to the user via XML output, and can also be viewed using an intuitive GUI which overlays the learnt information on the program source code. In addition, the output may be used to generate source code annotations which have application for automated software verification. We intend to extend this method to handle binary programs and, in the limit, obfuscated programs such as malware where the benefit to the user will be even greater.

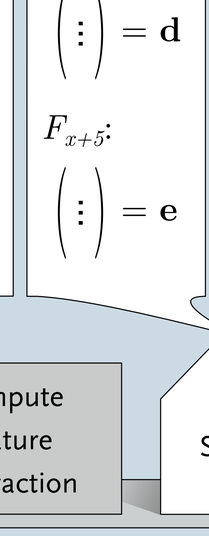
Preparing the Input

An overview of our tool **dsOli** (**D**ata **S**tructure **O**peration **L**ocation and **I**dentification) is given in the central figure. The analysis commences from a *concrete trace* obtained by executing the 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 (indicated by Ⓘ on the code opposite).

From the concrete trace we construct a *sequence of points-to graphs* describing the pointer structure in program memory after each event. The approach is demonstrated on a small C code fragment that performs insertions to an SLL. Part of the points-to graph sequence is shown on the right and corresponds to an invocation of `insert()`.

```
typedef struct node_t {
    int key;
    struct node_t *next;
} Node;

void insert(Node **l, int key) {
    Node *new = malloc(sizeof(Node));Ⓘ
    new->key = key;

    if (*l == NULL || (*l)->key >= key) {
        new->next = *l;Ⓘ
        *l = new;Ⓘ
        return;
    }

    Node *p = *l;Ⓘ
    while (p->next != NULL
              && p->next->key < key)
        p = p->next;Ⓘ

    new->next = p->next;Ⓘ
    p->next = new;Ⓘ
}
```





Locating Functional Units

We do not require that the syntactic *functional units* in the program coincide with their semantic functional units. Therefore, 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.

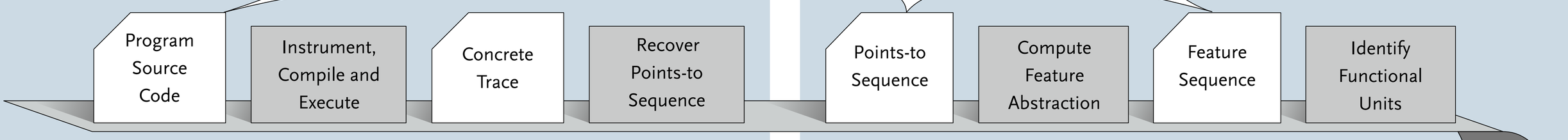
Our approach is based on the observation that program behaviour is, by nature, highly repetitive due to function calls and iterative constructs. We exploit this property to identify the functional units of a program by *repetition*.

Abstraction is necessary to expose the repetition such that we may generalize over code fragments with similar behavior. The result of applying the abstraction to a points-to graph G_x is a *feature vector* F_x.

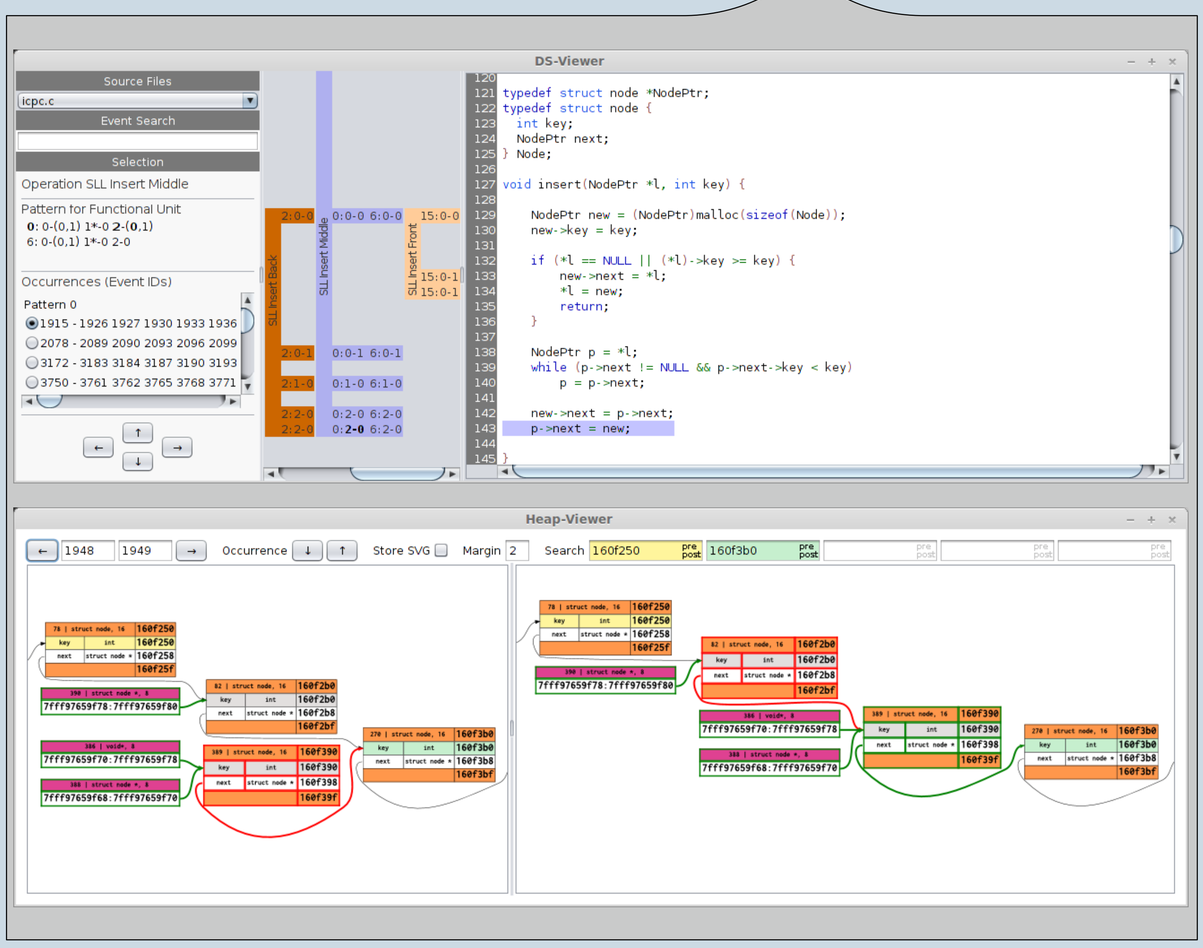
To search for *repeating patterns* in the feature sequence, we note that compression locates repetition. Thus, we employ a search algorithm to identify the set of patterns that best *compresses* the feature sequence.

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*.

Consider the specific invocation of `insert()` shown in the left points-to graph sequence. Other invocations of this operation which insert to the middle or end of the list will result in similar feature sequences, modulo the length of the traversal. Therefore, a suitable regular expression style pattern to recognize invocations of this operation could be $\mathbf{abc{*}de}$.

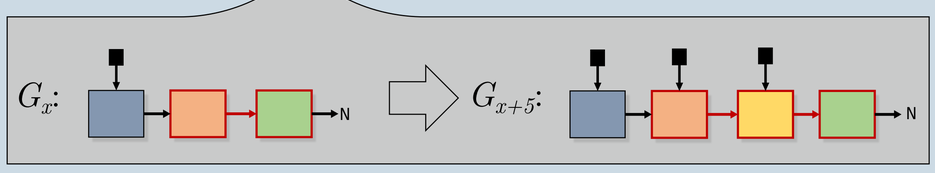


Analysis Output & GUI



A companion GUI overlays the results directly on the source code providing a useful visualization for the user. The pane adjacent to the source code summarizes the discovered data structure operations. The user may select an *occurrence* of an operation and step through the associated events. While stepping through, the points-to graphs of the current and previous events are displayed showing the concrete memory transformation. Each template consists of two graphs, one is matched to the points-to graph before the operation occurs and one is matched afterwards. The match of a template that recognizes inserts to the middle of an SLL is shown above (highlighted in red).

Identifying Data Structures & Operations



The best set of patterns 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. The potential operation (G_x, G_{x+5}) is shown above.

For each data structure operation of interest, a *template* is defined that will recognize the associated memory transformation.

Lastly, we use the set of operations that manipulated a data structure in order to name it. For example, if only inserts and removals to the front of an SLL were observed, then we would assign the name "SLL used as a stack".

Interfacing with Automated Verification

We are collaborating with partners at the University of York, UK and K.U. Leuven, Belgium to apply our tool in automated software verification. Our current goal is to automatically insert source code annotations suitable for proving memory safety properties in tools such as *VeriFast*. These can then be used to verify the program, or as a starting point for the verification engineer.

The annotations are written in *separation logic*, which allows reasoning about programs through memory partitioning, i.e., stating which parts of memory are unaffected by a code fragment. Our approach supports the automatic generation of candidate pre/post-conditions and loop invariants, as well as additional information that VeriFast requires to complete the proof.

Acknowledgements

ICPC'14, June 2, 2014

This work is partially supported by DFG grants LU 1748/2-1 and LU 1748/4-1

Lehrstuhl Softwaretechnik & Programmiersprachen
Otto-Friedrich Universität Bamberg
Fakultät WIAI

www.swt-bamberg.de/research/