ISSN 1744-1986

Technical Report N° 2008/ 10

Recovering Problem Structures from Execution Traces

Thein Than Tun
Yijun Yu
Robin Laney
Bashar Nuseibeh

30th June, 2008

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

Recovering Problem Structures from Execution Traces

Thein Than Tun Yijun Yu Robin Laney Bashar Nuseibeh
Department of Computing, The Open University, UK
{t.t.tun, y.yu, r.c.laney, b.nuseibeh}@open.ac.uk

Abstract

Software systems evolve in response to changes in stakeholder requirements. Lack of documentation about the original system can make it difficult to analyze and implement new requirements. Although automatic recovery of all requirements from an implementation is usually not possible, we suggest that the recovery of problem structures, which in turn inform the problem analysis of new requirements, is feasible and useful. In this paper, we propose the tool-supported approach to recover and maintain structures of problems, solutions, and their relationships, by recovering causal control and data dependencies between components. Extracting low-level program structures is done fully automatically, while higher-level descriptions of problem structures are obtained interactively. We validate our approach using a case study of a medium-sized open-source software system.

Keywords *Causality, Dynamic Data Dependency Analysis*

1 Introduction

Many software systems evolve to accommodate new stakeholder requirements. There are several Requirements Engineering (RE) approaches—such as the Problem Frames, KAOS, NFR, and i^* approaches [14, 29, 6, 33]—for capturing and analyzing user requirements early in the development. These approaches emphasize that, in the same way that programs have structures—called *solution* structures—the problems solved by the programs also have *problem* structures. In an implemented system, there are relationships between these two structures, and a good understanding of the problem structure is useful in analysis of its new requirements.

Although similar in their motivation to the UML use case and class models, the RE approach tend to focus on scoping and structuring the context of software systems and the intentions of stakeholders using techniques such problem diagrams [14] and goal models [29, 6, 33]. These techniques are helpful in describing requirements and specifications precisely, and in subjecting them to rigorous analyses [27, 8]. However, application of these RE approaches to the "brownfield" development of software systems has so far been limited, in part due to the difficulties of recovering appropriate models of existing software systems [2].

Recovery of architecture and design of software systems—in terms of artifacts such as class diagrams, sequence diagrams, component diagram and statecharts—have been well supported by reverse engineering techniques [5, 21]. Although automatically recovering all stakeholder requirements from an implementation is usually not possible, the recovery of the structures of implemented requirements is feasible and useful [36, 28]. In this paper, we describe an approach for recovering problem structures of the Problem Frames approach [14] from execution traces.

We use the term problem structure to refer to the problem context, upon which a program depends to satisfy a particular requirement. For example, a simple text editing program interacts with *domains*, such as the user, input device, the text object, and the display device. For the program to satisfy the requirement of editing, these domains must have appropriate properties and behavior. For instance, the user needs to provide correct commands in correct sequences, and the input and output devices must function properly. Furthermore, central to a problem structure is a notion of causality among these domains, demonstrating how the properties and behavior of the domains involved are sufficient in satisfying the requirement. Such explicit causal structures of a problem context is useful in scoping the problem and identifying areas of concerns where errors could arise [14].

The approach presented in this paper extracts problem structures from execution traces in three steps: (1) instrumenting and logging execution traces from requirement-driven test cases; (2) extracting causal dependencies from execution traces; and (3) reflecting the causal dependencies between functions into more abstract problem structures. We developed a tool-chain called `Galar` to perform steps (1) and (2) fully automatically, and step (3) interactively. `Galar`—based on the binary instrumentation frame-

work `Valgrind` [23] and `Fjalar` [11]— logs execution traces of C programs, containing necessary information for extracting the causal dependencies. It uses the relational programming tool `CrocoPat` and a simple awk script to implement a space efficient algorithm for extracting and reflecting causal dependencies in the execution traces into initial problem structures that can be refined interactively. Our approach is evaluated by applying it to extract problem structures of the open-source software Vim [1].

The rest of the paper is organized as follows. By way of putting the discussion into context, Section 2 gives a summary of related work. Section 3 presents a motivating example while outlining the challenges of extracting problem structures from code. Section 4 gives a detailed discussion of our approach, together with formal definitions of the key artifacts and the main algorithm used. Section 5 describes the implementing of our approach, which is evaluated on a medium-size open-source software system in Section 6. Concluding remarks are giving in Section 7.

2 Related Work

In iterative development of a software system, in the same way that the user requirements constrain the system behavior, the architecture of the existing system influences feasibility of implementing new requirements. Understanding the relationship between the requirements and architecture has been recogized as a long-standing research goal [24, 25]. Our work can be broadly classified as applying reverse engineering techniques to recover problem structures to be used in requirements engineering [35, 28].

Requirements Engineering Approaches There are a number of requirements engineering (RE) approaches that can be used to investigate problem structures. The goals-oriented approach KAOS refines high-level goals into subgoals and then into operational requirements [29]. Operational requirements are then assigned to agents in the solution space [18]. Using a similar notion of goals and goal-refinement, the NFR and i^* framework discuss how goals may contribute to achieving software quality [6, 33].

In this paper, we chose the language and techniques of the Problem Frames approach (PF) [14] to describe problem structures because (i) it provides mechanisms to consider solution components in the problem space [13], (ii) it allows known solution structures to influence problem decomposition [26, 12], (iii) it can recompose subproblems using a composition operator [16] and (iv) it allows the developers to reason about causality in resolving inconsistencies in requirements [17].

Although extensive research has been carried out in constructing problem structures from user requirements and analyzing them rigorously, little has been done so far to recover problem structures from code. Preliminary work carried out in [36, 28] shows the viability and utility of recovering problem structures, and in this work we present a tool-supported approach to recovering problem structures based on accurate and detailed causal models of the code.

Reverse Engineering of Behavioral Models from Execution Traces A general survey of reverse engineering has been given in [5, 21, 4], and we focus here on related work for recovering dynamic behaviors. Since state machines are typical representation of program behaviors, not surprisingly recent work has focused on extracting such design-level information [30, 9, 36]. In [30], a grammar inference-based approach was proposed to analyze user-provided accepted and rejected control event traces into state machines, and in [36] statecharts extraction is proposed based on static analysis of unstructured control flows. Both approaches rely on control-only information, which excludes crucial details of data for understanding various contexts in problem structures. In [9], context-enriched execution traces were used to extract label-transition systems (LTS), however, the contextual information used there requires the value of variables to be computed and summarized at the entry/exit of functions. This is not required by our memory address-based causality analysis.

Instrumentation Techniques Compiler-based static dataflow analysis techniques have been used in order to compute interprocedural program dependencies, e.g., in [32, 10, 31]. For the purpose of problem structure recovery, however, such techniques make conservative assumptions on all possible inputs instead of the traces of inputs related to requirements of a particular subproblem. Including data dependencies for all possible contexts is necessary for optimizing a program for unknown usages, however, it is not sufficient for understanding the behavior of the program in a particular problem context. On the other hand, compilers can still be used to produce dynamic traces for further analysis by instrumenting logging statements [32]. Aspect-oriented programming [15] and transformation systems [7] have also found their uses in instrumentation for the trace analysis. In this work, we chose to adapt the `Valgrind/Fjalar` dynamic supervisor framework [11] due to its maturity and efficiency in analyzing binary execution trace on the fly. In principle, similar instrumentation might be implemented by adapting a compiler when source code is available, as was done in a cache behavior analysis on Fortran programs [34].

3 Motivating Example

This section introduces a stack example, which we use to (i) demonstrate the challenges of recovering problem struc-

tures, and (ii) use it as a running example in the rest of the paper.

3.1 A Simple Stack

Two example implementations of a simple stack, based on an activity of a programming course, are listed in Figure 1. One implementation uses an array and the other uses a linked list to implement the stack. In other words, there are two possible solutions to the same stack problem. Given these implementations, developers may want to know the structure of the problem these programs are solving in order to, for instance, evolve these programs. For example, they may want to know the main components of the programs and the causal dependencies that exist between the components.

Array-based

```
// data structures
typedef char stackElementT;
typedef struct {
  stackElementT *contents;
  int maxSize;
  int top;
} stackT;
// operations
void StackInit(stackT *stackP,
               int maxSize) {
  stackElementT *newContents;
  newContents = (stackElementT *)
malloc(sizeof(stackElementT)*maxSize);
  if (newContents != NULL) {
    stackP->contents = newContents;
    stackP->maxSize = maxSize;
    stackP->top = -1;
  }
}
void StackDestroy(stackT *stackP){
  free(stackP->contents);
  stackP->contents = NULL;
  stackP->maxSize = 0;
  stackP->top = -1;
}
void StackPush(stackT *stackP,
          stackElementT element) {
  if (!StackIsFull(stackP))
   stackP->contents[++stackP->top]
       = element;
}
stackElementT StackPop(stackT *stackP){
  if (!StackIsEmpty(stackP))
   return stackP->contents
          [stackP->top--];
  exit(1);// Error
}
int StackIsEmpty(stackT *stackP) {
  return stackP->top < 0;
}
int StackIsFull(stackT *stackP) {
  return stackP->top
         >= stackP->maxSize - 1;
}
```

Linked list-based

```
typedef char stackElementT;
typedef struct stackTag {
  stackElementT element;
  struct stackTag *next;
} stackNodeT;
typedef struct {
  stackNodeT *top;
} stackT;
// operations
void StackInit(stackT *stackP) {
  stackP->top = NULL;
}
void StackDestroy(stackT *stackP) {
  // Nothing
}
void StackPush(stackT *stackP,
        stackElementT element) {
  stackNodeT * ptr = (stackNodeT*)
     malloc(sizeof(stackNodeT));
  ptr->element = element;
  ptr->next = stackP->top;
  stackP->top = ptr;
}
stackElementT StackPop(stackT *stackP) {
  if (!StackIsEmpty(stackP)) {
    stackElementT element =
      stackP->top->element;
    stackP->top = stackP->top->next;
    return element;
  }
  exit(1) // Error
}
int StackIsEmpty(stackT *stackP) {
  return stackP->top == NULL;
}
```

Figure 1. Two implementations of a stack from http://www.cs.bu.edu/teaching/c/stack/

3.2 Challenges

There are two specific challenges in reverse engineering a problem structure from program code: extraction and abstraction.

Extraction There are different kinds of information we can recover from code. For instance, we can recover the control structures programs, in terms of call graphs and abstract syntax trees. In such case, we obtain call dependencies between functions, but data dependencies between functions may be lost. On the other hand, we can extract data dependencies between parts of a program, but the control dependencies will be lost. The main challenge here is to extract accurate causal structures of programs.

Abstraction In programs of realistic sizes, the amount of details at the level of control and data dependencies are huge. Functions and data structures can be mapped at different granularity into more abstract structures using, for example, heuristics such as the naming conventions [22]. However, it is still a challenge to abstract detailed program structures into problem structure at a chosen level of granularity while preserving causal dependencies.

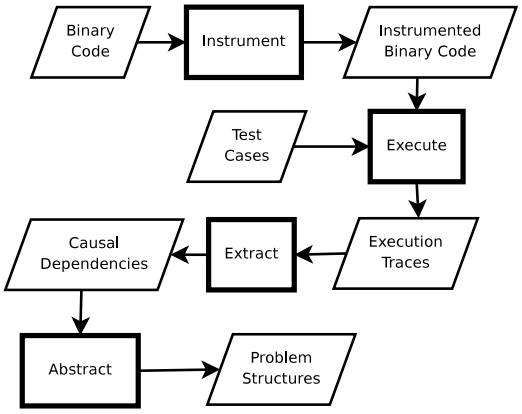


Figure 2. Schematic view of our approach

4 Our Approach

This section gives an overview of our approach before defining the target and source the reverse engineering, and as well as the step-by-step transformation of the source into the target. We also provide definitions of the main terminology, and the main algorithm of our our approach.

4.1 An Overview

As illustrated in Figure 2, the main idea of our approach is to:

- instrument binary code in order to obatin traces of single-threaded execution of the program containing information about when the execution has entered and exited certain functions, together with memory address

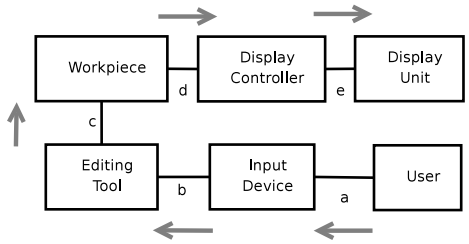
of global and static variables read and written by the functions

- identity all causal dependencies—namely true data and control dependencies—between functions, and
- reflect functions and their causal dependencies into initial problem structures, and refine them interactively.

4.2 Target of Our Reverse Engineering

The main target of our reverse engineering activity is a *problem structure*: a description of domains and their shared phenomena in the context of a problem. We give an intuitive definition of problem structures in Definition 1.

Definition 1 *A* problem structure *is a graph $G = \langle N, E \rangle$ where N is a set of domains, and E is a set of edges, each edge being associated with a set of phenomena. A domain is an abstraction of a component with observable behaviors, which is relevant to the problem at hand. Each shared phenomenon $e = (n_x, n_y)$ is a directed link from the domain n_x to n_y, and is read as the domain n_x controlling the phenomena e and the domain n_y observing the same phenomena. A shared phenomenon captures a set of causal relations between between two or more domains, denoting the fact that some observable behavior of a domain may cause other domain(s) to behave in a certain way.*



a : $U!\{user\ text\ \&\ command\}$
b : $ID!\{input\ signal\}$
c : $ET!\{marked\ up\ text\}$
d : $W!\{marked\ up\ text\}$
e : $DC!\{display\ text\}$

Figure 3. Problem Structure of a Simple Editing Problem

For example, Figure 3 shows the structure of a simple text editing problem, where the solid rectangles and lines denote the domains and shared phenomena respectively. The diagram shows a user who uses an input device to instruct a computer program to edit a lexical object such as a text document, which is displayed in a display device by display controller. In this case, the causality of the problem structure begins with commands of the user, and ends with the text being displayed by the display unit. Descriptions of the solid line such as $a : U!\{user\ text\ \&\ command\}$ denotes the fact that the domain User controls the phenomena $user\ text\ \&\ command$, which is observed by the Input Device domain because it is connected to User at a. When the user provies text and commands, the Input Device domain in turn generates input signals to the Editing Tool, which produce marked-up text for the Workpiece domain, which is extracted for display in the Display Unit by the display controller.

Therefore, the chain of causality in the diagram follows the direction of shaded arrows. The requirement, not shown in the diagram in Fig.3, is typically described as a relation between the domains at the beginning and end of a causal chain. For instace, a requirement in this problem context might be to "allow interactive editing by the user".

Although similar in some way to UML sequence and interaction diagrams, problem structures have a richer notion of causality: for instance, shared phenomena are not restricted to method or function calls between components.

4.3 Source of Our Reverse Engineering

The source of reverse engineering in this work is binary code and test cases. We instrument the binary code in order to obtain and analyze causality from execution traces for test cases related to the requirement of the problem we are interested. In this way, we can effectively filter out information irrelevant to analysis for a new requirement. The idea is similar to the way program slicing approaches are used to abstract programs [31].

We now define the term execution trace at the implementation level (Definition 2). For simplicity of presentation, we assume that each trace is generated by a sequentially executed program. For traces of multi-threaded programs, the execution trace annotated with the ID of multiple processors can be separated into a number of single-threaded traces for a similar analysis. Since we are extracting the computing semantics of causality, the synchronization among these processors can be ignored due to the fact that a correct parallel execution must respect the ordering of events imposed by data dependencies [10].

Definition 2 *A* trace *of a thread of execution σ is defined as a sequence $\langle \sigma_1, \sigma_2, \ldots \sigma_n \rangle$ where each element σ_i is a recorded event of a program execution. An event is either $Enter(f)$, $Exit(f)$, $Read(a)$ or $Write(a)$, where f is a function, a is a memory address. These events are often abbreviated as $E(f), X(f), R(a), W(a)$. $E(f)$ means that the execution has just entered the function f, $X(f)$ means that the execution has just exited the function f, $R(a)$ means that the data at the memory address a has been read, and*

$W(a)$ means that the data at the memory address a has been written to. These elements in the trace are totally ordered by their occurrence in the execution.

Conceptually, there are two stages in our transformation of the execution traces into problem diagrams: first, we will extract all direct causal dependencies (including both control and data dependencies) between functions, and second, abstracting those dependencies into problem structures.

4.4 Extracting Causal Dependencies

Given a sequence, a subsequence of events between $E(f)$ and $X(f)$ are called the scope of f, written as $Scope(f)$. For example, in the following trace of events,

$$\langle E(m), W(a), W(b), E(f), R(b), X(f), W(a), X(m)\rangle$$

$Scope(m)$ is the trace

$$\langle W(a), W(b), E(f), Scope(f), X(f), W(a)\rangle$$

where $Scope(f)$ is $\langle R(b)\rangle$. A causal relation between two events is a strict ordering where one event causes certain changes that the other depends on, and such ordering cannot be interchanged without altering the program behavior. For example, exchanging $W(b)$ in $Scope(m)$ with $R(b)$ in $Scope(f)$ may lead to a different program behavior. However, exchanging $W(a)$ with $W(b)$ does not alter the program behavior (notice that a and b are not the same). In an execution trace according to Defintion 2, there are two kinds of causality: data and control.

The data causality, or *true data dependency*, indicates that there is a subsequent read after a write event to a particular memory address. On the other hand, dependencies such as read after read, write after read, and write after write are known to be false data dependencies [10]. Such false data dependencies can be removed by introducing private copy of variables, without altering the program behavior [19]. The definitions of data dependency are given in Definition 3—5.

The definition of memory address data dependency (Definition 3) tells us which read event is dependent on which write event.

Definition 3 *For any given memory address a and two events σ_i and σ_j, where $i < j$, $\sigma_i = W(a)$ and $\sigma_j = R(a)$, if there is no σ_k such that $i < k < j$ and $\sigma_k = W(a)$, then σ_j is* causally dependent *on σ_i.*

Definition 3, however, does not tell us which functions these events belong to. Since the execution may have entered several functions before a given read or write event happens, we need the notion of closest containing function for a given memory access event (Definition 4).

Definition 4 *A containing function for a read or write event σ_j is a function f such that there is σ_i in $Scope(f)$. Furthermore, f is said to be* closer *to a containing function g if the $Scope(f)$ is contained in $Scope(g)$. The* closest *containing function f is the containing function with the closest scope.*

Input: A thread of execution trace σ (Defn. 2)
Output: Dependency relations: Dep (Defn. 3), Call(Defn. 6)

```
1  begin
2      writtenBy = an empty hashmap from a
       memory address to an event;
3      ccf = an empty hashmap from a memory address
4      to a function; // closest containing function
5      fStack is a stack of caller functions of the
       currently executing function;
6      foreach σ_j in σ do
7          ccf(σ_j) = fStack.top();
8          switch σ_j do
9              case R(a):
10                 σ_i = writtenBy(a);
11                 if σ_i exists then
12                     g = ccf(σ_i);
13                     f = fStack.top();
14                     set Dep(f, g, a) is true;
15                 end
16                 break;
17             case W(a):
18                 writtenBy(a) = σ_j;
19                 break;
20             case E(f):
21                 /* always the first event in the
                   execution trace */
22                 if !fStack.isEmpty() then
23                     set Call(fStack.top(), f) is true;
24                 end
25                 fStack.push(f);
26                 break;
27             case X(f):
28                 fStack.pop();
29                 break;
30             end
31         end
32     end
33 end
```

Algorithm 1: Extracting causal dependencies from an execution trace

Having defined the closest containing function for memory access events, we can now define data dependencies between functions (Definition 5).

Definition 5 *A function f depends on another function g on*

a memory address a, denoted by a predicate Dep(f, g, a), if there is a dependency from $\sigma_i = W(a)$ to $\sigma_j = R(a)$ and f, g are respectively the closest containing functions *of these events.*

Like true data dependencies, the call relations between functions are also causal in the sense that the execution of the caller function causes the execution of the callee function. Call dependencies are exposed in the execution trace as "call" relations between functions, as indicated by Definition 6.

Definition 6 *A function f* calls *a function g, denoted by a predicate $Call(f, g)$, if and only if $Scope(f)$ contains $Scope(g)$, and for any function $h \neq f$ if $Scope(h)$ contains $Scope(g)$ then $Scope(h)$ also contains $Scope(f)$.*

The definitions 2—6 give us a way to extract all causal dependencies between functions from the execution trace. Algorithm 1 illustrates a space efficient way of extracting a problem structure from an execution trace. The algorithm uses a hashtable `writtenBy` to maintain the mapping to memory addresses to write events (line 2), another hashtable `ccf` to maintain the mapping of the memory addresses to closet containing functions (lines 3-4), and a stack `fStack` to record the caller function(s) of the currently executing function (line 5).

The algorithm iterates over the events in the execution trace σ of Definition 2 (lines 6-32). If the event encountered is an $E(f)$ event, the current containting function (`ccf`) is pushed into the function stack `fStack` (lines 20-26); if it is an $X(f)$ event, the top of `fStack` is popped as closest containing function of the current event; and if it is an $E(f)$ event, a containment relation `Call` is recorded between the previous top of `fStack` and f (lines 27-29). While processing a $\sigma_j = W(a)$ event, the hashtable `writtenBy` is updated so that a is mapped to σ_j (lines 17-19). If a $R(a)$ event is encountered, the latest event that writes to the address a is retrieved from the `writtenBy` hashtable and reported as a data dependency on a between the containing functions of the write and read events (lines 9-16).

The algorithm is space efficient because it requires a single pass of all events in the execution trace. If carefully integrated with instrumentation frameworks, it does not require saving the trace to a file. The computation complexity is $O(mn)$ where n is the size of trace, and m is the size of memory consumption. The spatial complexity for the bookkeeping is $O(m)$ where each memory access may require a few book keeping bytes.

Having extracted all direct causal dependencies between functions, we can now abstract them into problem structures.

4.5 Abstracting Problem Structures

Definition 7 provides an intuitive way of relating dependency relationships to initial problem structures.

Definition 7 *A function is an atomic* domain *and a memory address is an atomic* phenomenon. *A domain is an abstraction of a group of functions. A phenomenon is an abstraction of a set of memory addresses. A phenomenon* shared *between two or more domains is a set of addresses that cause dependencies among functions in these domains. A phenomenon is* hidden *if it is shared among functions of the same domain, and not shared with functions in other domains.*

When abstracting problem structures from atomic domains and shared phenomena, we rely on the user heuristics about the `Containment` relations. The notion of containment is broader than the containing function of an event provided in Definition 4. With containment, we refer to several kinds relations such as the containment of function in another function, a module, a folder, or a file, as well as the containment of a variable in an array, a record, a scalar or a field. This is similar to the way naming conventions are used in software reflexion models [22].

5 Implementation of `Galar`

In this section we highlight our contribution in the implementation of the `Galar` tool-chain for obtaining execution traces and abstracting problem structures. In order to obtain the execution traces, it is necessary to instrument all read and write access of variables and entry and exit points for functions. In `Galar`, we instrument the executables using the open-source binary instrumentation framework for execution supervision `Valgrind/Fjalar` [23, 11].

The `Valgrind` framework allows us to intercept machine instructions during the dynamic execution, by simulating the computation on an execution engine that can dynamically perform additional tasks upon request. The `Fjalar` tool, based on `Valgrind`, calculates values of variables at the beginning and end of functions.

In order to obtain the execution traces as required by Definition 2, `Galar` extends `Fajalar` by bookkeeping relevant read and write events togeter with the memory addresses involved. Whenever possible, the addresses obtained from the execution are compared with those bookkept by `Fjalar` to relate them to the names of global or static variables. In this way, `Galar` obtains sufficient information to extract data dependencies between functions. Furthermore, implementation of the algorithm 1 is integrated with trace logging so that the extraction of data dependencies are done entirely online.

In order to improve the performace, we instrument only the variables and functions within the scope of each test case being run. This is done in two easy steps. First, we use the `Fjalar` options `-dump-var-list` and `-dump-ppt-list` to generate complete lists of global and static variables and all user-defined functions in the program. Second, we provide a subset of variables and functions to monitor through the `Fjalar` options `-var-list-file=varfile` and `-ppt-list-file=pptfile`. This greatly reduce the input size to the algorithm 1.

`Galar` implements the abstraction heuristics by describing the dependency relationships in RSF format [20], and applying the following Crocopat script [3].

```
Abstract(x, y):= Contain(x, y) | x = y;
AbstractCall(f, g) := Call(f1, g1) &
   Abstract(f, f1) & Abstract(g, g1);
AbstractDep(f, g, a) := Dep(f1, g1, a1) &
   Abstract(f, f1) & Abstract(g, g1) & Abstract(a, a1);
```

This step is semi-automatic in the sense that the elements of the `Containment` rules can be adjusted by the developer, using, for instance, logical modules or clustered modules.

The source code of the `Galar` tool-chain can be checked out from the following repository:

`http://computing-research.open.ac.uk/repos/datadep`.

5.1 The Running Example

Figure 4 shows the design of three test cases to validate that given a string of input characters, the stack produces the same string with characters in the reverse order of the input. We then run the test cases with three different input values: T1 (empty string input), T2 (single character input), T3 (three character input). Notice that the test cases for array based implementation initializes the stack by the length of the string being entered. We then instrumented the compiled program using `Galar` tool-chain and computed the data dependencies on the fly during the program executions. For given inputs with an empty string (T1), a single character string (T2) and a 3 character string (T3), we obtained three execution traces for each of the two programs. An example execution trace is given in Figure 5.

In Figure 5, each data dependency is represented by an entry beginning with (DataDep), which include the memory address (such as 0xBEA70624) and two different closest containing function of the respective write and read events (such as main() and StackInit()). Notice that some of the memory addresses are associated with names of global/static scalar variable, for instance, the address 0x80BE32C is associated with the variable stack.top). From this trace, we obtained a concrete causality structure from three relations, namely, `Contains, Call, DataDep`.

```
stackT stack;
char *traverse;
#if ARRAY
char str[101];
#endif
int main(void) {
  printf("Enter a string: ");
#if ARRAY
  StackInit(&stack, strlen(str));
#else
  StackInit(&stack);
#endif
  gets(str);  /* Read line. */
  for (traverse = str;*traverse != '\0';traverse++)
    StackPush(&stack, *traverse);
  printf("\nPopped characters are: ");
  while (!StackIsEmpty(&stack))
    printf("%c", StackPop(&stack));
  printf("\n");
  StackDestroy(&stack);
  return 0;
}
```

Figure 4. Design of Test Cases

```
[..main() - ENTER]
[..StackInit() - ENTER]
 DataDep: 0xBEA70624: : ..main(): ..StackInit()
 DataDep: 0xBEA70620: : ..main(): ..StackInit()
 DataDep: 0xBEA70620: : ..main(): ..StackInit()
 DataDep: 0xBEA70624: : ..main(): ..StackInit()
 DataDep: 0xBEA70620: : ..main(): ..StackInit()
 DataDep: 0xBEA70618: : ..main(): ..StackInit()
 DataDep: 0xBEA7061C: : ..main(): ..StackInit()
[..StackInit() - EXIT]
[..StackIsEmpty() - ENTER]
 DataDep: 0xBEA70620: : ..main(): ..StackIsEmpty()
 DataDep: 0x80BE32C: /stack.top: ..StackInit(): ..StackIsEmpty()
 DataDep: 0xBEA70618: : ..main(): ..StackIsEmpty()
 DataDep: 0xBEA7061C: : ..main(): ..StackIsEmpty()
[..StackIsEmpty() - EXIT]
[..StackDestroy() - ENTER]
 DataDep: 0xBEA70620: : ..main(): ..StackDestroy()
 DataDep: 0x80BE324: /stack.contents: ..StackInit(): ..StackDestroy()
 DataDep: 0x80C0684: : ..StackInit(): ..StackDestroy()
 DataDep: 0x80BD360: : ..StackInit(): ..StackDestroy()
 DataDep: 0x80C0684: : ..StackInit(): ..StackDestroy()
 DataDep: 0x80C0694: : ..StackInit(): ..StackDestroy()
 DataDep: 0xBEA70620: : ..main(): ..StackDestroy()
 DataDep: 0xBEA70620: : ..main(): ..StackDestroy()
 DataDep: 0xBEA70620: : ..main(): ..StackDestroy()
 DataDep: 0xBEA70618: : ..main(): ..StackDestroy()
 DataDep: 0xBEA7061C: : ..main(): ..StackDestroy()
[..StackDestroy() - EXIT]
[..main() - EXIT]
```

Figure 5. Data dependencies reported for an execution of test case T1

Figure 6 illustrates a detailed causality structure extracted from an execution trace of the array-based implemenation T1. In the figure, the `Call` relations are shown in solid arrows, whereas the `DataDep` relations are shown in dashed arrows. The labels on the dashed arrows indicate the memory address involved in the dependency.

After abstracting the exection traces, we obtained the problem structures for T1, T2 and T2 for both array-based and linked list-based implementation of the stack, as shown in the middle and right columns in Fig. 7. (For reasons of space and readibility, the data dependencies are represented by dotted arrows and call dependencies are represented by solid arrows, rather than the traditional plain lines. We also omitted the names of shared phenomena for the same reasons). A domain without an incoming arrow, such as `StackInit`, indicates an initiator of a causal

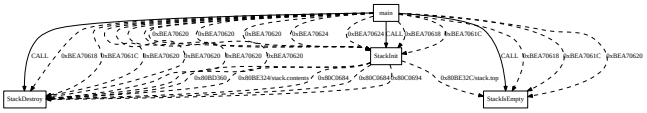


Figure 6. Causal Dependencies in the Execution Trace for T1

chain, whilst a domain without an outgoing arrow, such as `StackIsEmpty` and `StackDestroy`, indicates a terminator of a causal chain.

Using problem structures such as those shown in Fig. 7, the developer can perform certain problem analyses. For instance, it is interesting to observe that

1. execution tracesof T2 and T3 have same problem structures, perhaps indicating that the test cases have similar coverage;
2. an execution trace of the linked list-based stack is very much similar to an execution trace of an array-based stack, yet there is a difference in that `StackIsFull` and `StackDestroy` are not part of the linked list problem structures, perhaps because the program assumes that the linked list can grow infinitely, and that the linked list does not have to be destroyed after popping, and
3. `StackInit` is at the beinning of all causal chains, indicating that the program initializes the stack in all cases.

Examining the "concerns" raised above may lead the developers to identify potential causes for failure [14]. For instace, the developers should check the assumption that the linked list can grow infinitely is a reasonable assumption. In addition to identifing potential causes for failures, the developers can also check the certain concerns that have been addressed: for instace, both programs seem to be free of the initialisation concern.

6 Evaluation

In order to evaluate the performance and scalability of `Galar`, we applied the tool-chain to Vim [1]. First released in 1991, Vim is a popular open-source 'modal' text editing software that has been evolving continuously, and the recent release Vim 7.1 has approximately 323K lines of C code.

6.1 Experiment Design

The main aim of our experiment were to investigate the performance and scalability of the extraction and abstraction processes. In particular, we intended to find out

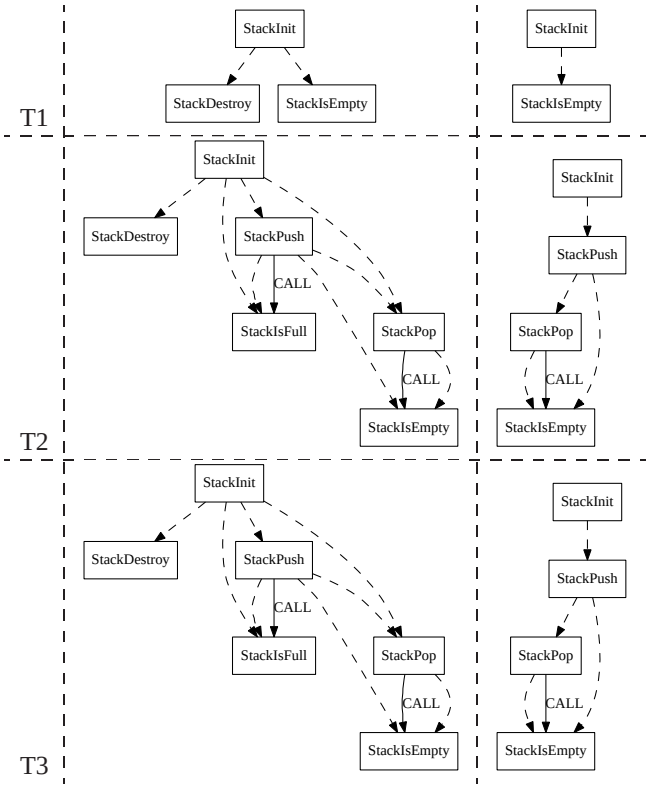


Figure 7. Problem Structures for T1, T2 and T3 for array-based linked list-based implementations.

1. the relation of the execution time to the complexity of causal dependencies extracted for each test case, and
2. the relation of the graph complexity of problem structures to the complexity of the extracted causal dependencies.

In order to compute the complexities of time, causal dependencies and problem structures through regression, we did the following. We first identified all global/static variables and functions in Vim. Then for each test case of varying functionality, we executed Vim to obtain a complete call graph of the test case. The resulting call graph was sliced into subgraphs of depths 0 to n, starting from the root function. The functions in these sliced callgraphs were then used to reinstrument and reran the test cases for each subgraph, while recording the execution time, the size of causal dependencies, and the McCabe complexity ($\#Nodes + \#Edges - 2$) of problem structures.

The experiments used four test cases covering various operations of Vim, such as the user I/O, disk I/O, and buffer. The test cases are (i) to display the version of Vim using the command (`vim --version`), (ii) to get Vim help

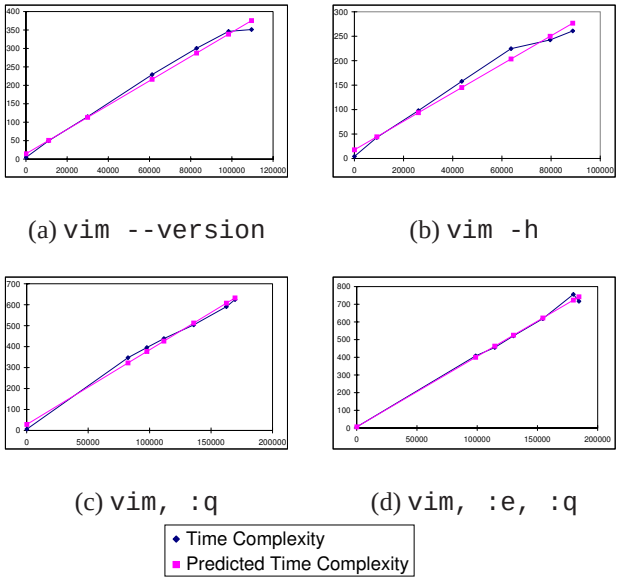


Figure 8. X Axes represent the McCabe graph complexity, Y Axes represent the time complexity.

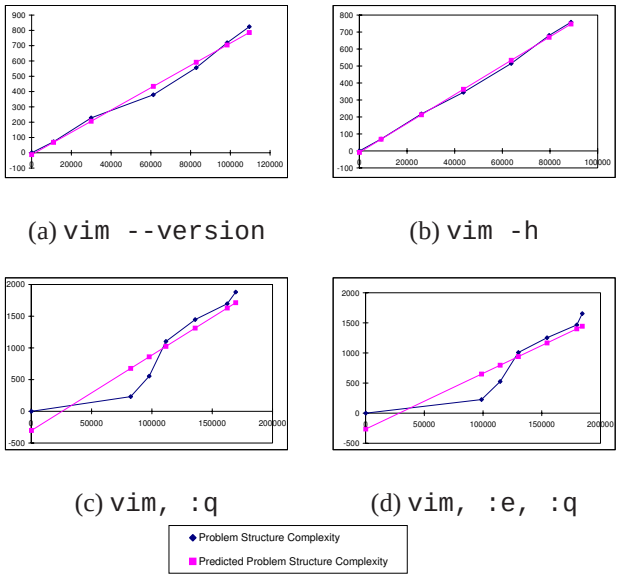


Figure 9. X Axes represent the input complexity, Y Axes represent the output (graph) complexity.

(`vim -h`), (iii) to start and quit Vim (`vim, :q`), and (iv) to start, open a file and quit (`vim, :e, :q`).

The experiments were carried on a Sony VAIO VGN-TX1XP Laptop (1.20GHz with 1GB RAM) running Ubuntu 8.04, and the regression analysis was done using the built-in tool in MS Excel 2003.

6.2 Results and Analysis

We did the above experiments for four selected test cases and for $n = 6$, which providing us with 7 data points for each test case. The main results are that the the relation of the execution time to the complexity of causal dependencies, and the relation of the graph complexity of problem structures to the complexity of the causal dependencies seem to be linear. Figure 8 shows that there is a linear relation between the the extracted causal dependencies and time complexities, with confidence level above 95%. The relations are quite smooth even when the dependency complexity has shot up to around 184K. Figure 9 shows the relation between the complexities of causal dependencies and problem structures remain linear as the former increases. These results suggest that the extraction and abstraction techniques of `Gala` are efficient.

7 Conclusions and future work

In this paper, we proposed a tool-supported approach to recover problem structures from execution traces, by extracting and abstracting causal control and data dependencies between components. Extracting low-level causal structures is done fully automatically, while higher-level descriptions of problem structures are obtained interactively: through abstraction based on the user heuristics. Our approach is shown to be efficient and scalable when applied to a medium-sized open-source software system. In future work, we intend to explore the following:

- Although we have been able to recover problem structures from execution traces, we have yet to identify *Problem Frames* from the code [14]. There is, of course, a limitation to the extent of design alternatives being recovered directly from code. Perhaps analysis of artefacts such as comments, user documentation and error reports using natural language processing techniques may complement this work.

- Despite the findings that `Galar` is scalable, there is an inevitable overhead imposed by `Valgrind/Fjarlar` framework. We believe that the performance and scalability of our our approach can be further improved. We are investigating more efficient way to obtain the execution trace using AOP and source-to-source compiler-based instrumentations techniques [15, 32].

Acknowledgements

We thank our colleagues at the Open University, in particular Michael Jackson, for helpful comments and suggestions. This research is funded by the EPSRC, UK.

References

[1] http://www.vim.org/.

[2] J. Aranda, S. Easterbrook, and G. Wilson. Requirements in the wild: How small companies do it. In *Proc. of RE*, pages 39–48, 2007.

[3] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE TSE*, 31(2):137–149, 2005.

[4] G. Canfora and M. D. Penta. New frontiers of reverse engineering. In *Proc. of FOSE*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.

[5] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[6] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[7] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[8] C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *SIGSOFT '06/FSE-14*, pages 197–207, New York, NY, USA, 2006. ACM.

[9] L. M. Duarte, J. Kramer, and S. Uchitel. Towards faithful model extraction based on contexts. In *Proc. of FASE*, pages 101–115, 2008.

[10] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[11] P. Guo. A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs. Master's thesis, MIT, 2006.

[12] J. Hall, L. Rapanotti, and M. Jackson. Problem oriented software engineering: Solving the package router control problem. *IEEE TSE*, 34(2):226–241, 2008.

[13] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *Proc. of RE*, pages 137–144. IEEE Computer Society, 2002.

[14] M. Jackson. *Problem Frames*. ACM Press & Addison Wesley, 2001.

[15] G. Kiczales and E. Hilsdale. Aspect-oriented programming. In *Proc. of ESEC/FSE*, page 313, 2001.

[16] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *Proc. of RE*, pages 122–131. IEEE Computer Society, 2004.

[17] R. C. Laney, T. T. Tun, M. Jackson, and B. Nuseibeh. Composing features by managing inconsistent requirements. In *Proc. of ICFI*, pages 129–144, 2007.

[18] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proc. of ICSE*, pages 83–93, New York, NY, USA, 2002. ACM Press.

[19] D. Maydan, S. Amarasinghe, and M. Lam. Array-data flow analysis and its use in array privatization. *Proc. of PoPL*, pages 2–15, 1993.

[20] H. Müller and K. Klashinsky. Rigi: a system for programming-in-the-large. *Proc. of ICSE*, pages 80–86, 1988.

[21] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE - Future of SE Track*, pages 47–60, 2000.

[22] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE TSE*, 27(4):364–380, 2001.

[23] N. Nethercote and J. Seward. Valgrind A Program Supervision Framework. *Elec. Notes in Theo. Comp. Sci.*, 89(2):44–66, 2003.

[24] B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, 2001.

[25] B. Nuseibeh and S. M. Easterbrook. Requirements engineering: a roadmap. In *Proc. of FOSE*, pages 35–46, 2000.

[26] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *Proc. of RE*, pages 80–89, 2004.

[27] R. Seater and D. Jackson. Requirement progression in problem frames applied to a proton therapy system. In *Proc. of RE*, pages 166–175, Washington, DC, USA, 2006. IEEE Computer Society.

[28] T. T. Tun, Y. Yu, R. Laney, and B. Nuseibeh. Recovering problem structures to support the evolution of software systems. Technical report, The Open University, 2008.

[29] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proc. of RE, 2001*, pages 249–262, 2001. TY - CONF.

[30] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proc. of WCRE*, pages 209–218, 2007.

[31] M. Weiser. Program slicing. *Proc. of ICSE*, pages 439–449, 1981.

[32] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, et al. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

[33] E. S. K. Yu and J. Mylopoulos. Understanding "why" in software process modelling, analysis, and design. In *Proc. of ICSE*, pages 159–168, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[34] Y. Yu, K. Beyls, and E. H. D'Hollander. Visualizing the impact of the cache on program execution. In *Proc. of IV*, pages 336–341, 2001.

[35] Y. Yu, J. Mylopoulos, Y. Wang, S. Liaskos, A. Lapouchnian, Y. Zou, M. Litoiu, and J. C. S. do Prado Leite. Retr: Reverse engineering to requirements. In *Proc. of WCRE*, page 234, 2005.

[36] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse engineering goal models from legacy code. In *Proc. of RE*, pages 363–372, 2005.