

Static Trace Extraction

Thomas Eisenbarth, Rainer Koschke, and Gunther Vogel
University of Stuttgart, Germany
{eisenbarth, koschke, vogel}@informatik.uni-stuttgart.de

Abstract

A trace is a record of the execution of a computer program, showing the sequence of operations executed. Dynamic traces are obtained by executing the program and depend upon the input. Static traces, on the other hand, describe potential sequences of operations extracted statically from the source code. Static traces offer the advantage that they do not depend upon input data.

This paper describes a new automatic technique to extract static traces for individual stack and heap objects. The extracted static traces can be used in many ways, such as protocol recovery and validation in particular and program understanding in general.

1. Introduction

During design, a system architect decomposes a larger software system into smaller and more manageable components. A component has an interface, which specifies its external properties. Two aspects of interfaces can be distinguished: *syntactic* and *semantic* interfaces. The *syntactic interface* describes the syntactic conventions between the component itself and its clients, such as names of routines, number and types of parameters, etc. Consider Figure 1 as an example of a typical syntactic interface description. Most programming languages require specifying syntactic conventions, and compilers report errors if a violation of a syntactic convention is encountered. Yet, the syntactic information alone is not sufficient to use the component correctly. If, for instance, an actual sequence of operations offered by a component violates the given restrictions, a failure may occur at run time. Such failures are often hard to find because they may become visible long after the actual fault happened.

The *semantic interface* describes the semantic conventions between the component and its clients—for instance, effects of routine calls and sequencing constraints as well as pre- and postconditions and invariants of the component. Most programming languages provide no means to specify

semantic conventions. Far too often, the semantic part is only described by informal comments or not documented at all. Reconsider Figure 1 as an example of a typical interface description. Given this informal description, it is not clear how the component is to be used correctly. For instance, the comments do not specify whether the initialization is optional or required before other routines of `Stack` can be called.

Contributions. In this paper, we will focus on sequencing constraints of operations as one important aspect of the semantic description of an interface, similarly to [1, 9]. We will show how operation sequences can be extracted from source code completely automatically. We introduce *object process graphs* as a finite description of all possible sequences. Object process graphs are extracted for individual stack and heap objects.

Use of object process graphs. Protocols of components specify the legal operation sequences. To validate operation sequences of a program statically, it is necessary to extract object process graphs first [9]. Additionally, object process graphs are necessary input to our new semi-automatic protocol recovery for components, a technique to recover protocols from existing code by unifying the actual usages of a component [3].

```
typedef ... Stack;
void init (Stack *stack);
// Initialization
void push (Stack *stack, Item i);
// Pushes item i onto stack
Item pop (Stack *stack);
// Returns and removes top-most element
Item top (Stack *stack);
// Yields top-most element
int is_empty (Stack *stack);
// True iff stack is empty
void release(Stack *stack);
// Destructor
```

Figure 1. Example syntactic interface.

Overview. Section 2 describes related research. Section 3 introduces object process graphs. Section 4 explains what kind of objects are traced and how they are modeled. In Section 5, the generic algorithm for extracting individual

object process graphs is described, and Section 6 describes how points-to information is integrated.

2. Related Research

Protocol validation. The aim of protocol validation is to check that each trace conforms to the specified protocol. Protocols are typically checked at run time. However, to be on the safe side, one has to validate protocols statically. Static protocol validation has to check that every static trace is either infeasible or is covered by the protocol specification. Protocol validation can only be done semi-automatically since many questions related to protocol validation are generally undecidable. However, it is still useful for large systems to at least identify the potential mismatches between static traces and the specified protocol automatically. The user can then focus on the mismatches to decide whether the apparently illegal static traces actually do not conform to the protocol.

Object process graphs describe all potential static traces for a given object and are extracted from source code. Both an object process graph and a protocol describe a language. Thus, for such a validation, one basically has to show that the language described by the object process graph is a subset of the language described by the protocol. Unfortunately, verifying this property is only possible for regular languages in general. Consequently, different authors have proposed to use finite state automata to specify protocols [1, 9].

These protocols express the sequencing constraints on a component's operations only. Constraints on the data passed to the components, for instance, are not part of sequencing constraints. For example, it cannot be expressed with finite state automata that the element that is currently being retrieved from a container component must have been added before.

To automatically prove that the language described by an object is a subset of the language described by the protocol, one can use known algorithms for finite state automata [7]. An alternative approach was proposed by Oleander and Osterweil, who use a data flow framework in which state transitions are propagated through the control-flow graph (CFG) [9]. The advantage of their approach is that it does not only check universally quantified but also existentially quantified constraints.

Oleander and Osterweil have only shown how to validate sequencing constraints for operations that do not use objects as arguments. In other words, the operations are considered parameterless routines. Given these kinds of operations, the CFG is sufficient. Yet, if operations relate to objects, the validation technique by Oleander and Osterweil cannot be applied. Our technique allows to extract object process graphs per object based on data flow information.

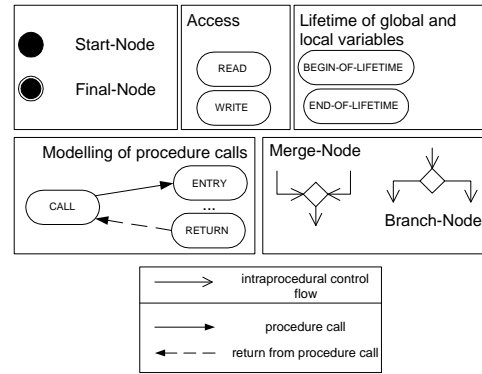


Figure 2. Nodes and edges of process graphs.

These object process graphs could be used as input to Oleander and Osterweil's technique to validate the operation sequences of the object against a protocol.

Protocol recovery. To validate a protocol, one needs the protocol in the first place. For many components, the protocol has not been specified. For these components, protocols may be derived through dynamic and static analyses.

ISVis [6, 5] is a system that allows one to identify interaction patterns in program event traces that can help an analyst construct design-level behavioral models from the low-level behavior of a system. The source code is instrumented to track interesting events. The resulting event traces can be analyzed post-mortem using the ISVis graphical views. As for all dynamic analyses, the results are only valid for the particular scenarios that generate the program event traces. Moreover, the patterns of interaction need to be identified by the human viewer.

Cook and Wolf [2] describe a method to automatically infer recurring patterns of behavior from a stream of dynamically collected events. They recast the problem of protocol recovery as *grammar inference*, which attempts to derive a regular grammar from token sequences.

Heiber [3] has shown how to unify extracted object process graphs for objects that are instances of the same component to recover the protocol for this component. The semi-automatic method uses well-known algorithms for finite state automata and additional user-defined graph transformations.

3. Object Processes

This section introduces object processes that describe the set of static traces relative to a statically detectable object. The set of dynamic traces for all run-time objects, which are represented by a specific statically detectable object, are a subset of the set of static traces of the latter.

Objects are instances of components that export a user-defined type. For instance, if the component is an abstract data type, a client of the component may declare variables of this type or may allocate a heap object of this type.

The purpose of object processes is to describe all potential sequences of operations on a given object. Object processes are represented by process graphs, which are basically projections of CFGs for a specific object. Figure 2 depicts a legend of the elements of process graphs. The nodes of process graphs represent operations on objects, branches and merges of control flow, and procedure calls. Operations on objects are (partial) reads and writes to the object as well as passing an object as actual parameter to a routine. If the routine to which the object is passed is not part of the interface of the component, an object process graph will be used to represent the operations of the called routine on the object. Otherwise, the routine is considered *atomic* (also known as *primitive*) and its implementation is considered hidden.

Entry and *return* nodes indicate the entry of a procedure from a procedure call and the return of control to a caller, respectively. Special nodes denote the begin and end of lifetime of local and global stack variables. Control flow starts at start nodes and ends at final nodes.

Edges represent the possible flow of control between two nodes. Edges may optionally be annotated with *true* or *false* if the control flow depends upon a true or false predicate of interest, respectively. Such conditional edges may only start at branch nodes.

Figure 3 shows a first example. The program in Figure 3(a) consists of the functions `main` and `P`. Both functions use and access a global `Stack` object. Figure 3(b) shows the corresponding object process. The lifetime of the global variable begins at the start of the program (i.e., call to `main`) and ends when the program terminates (i.e., exit of `main`). In `main`, routine `P` is called (the index is used to distinguish different call sites of `P`), which either initializes `stack` or calls itself recursively. Upon return of the recursive call, `push` is applied to `stack`. When `P` exits, `main` checks whether `stack` is empty and if so, `pop` is applied to `stack`.

Note that one could add many other variables of type `Stack` and many other statements to the given program and the resulting object process graph for variable `stack` would look alike as long as these additions do not affect `stack`. Note also that object process graphs are completely different from program slices [4]. Object process graphs represent the order of operations on one object, whereas program slices contain all statements that affect or are affected by a statement. For instance, if an operation `top` is applied twice on an object directly one after the other, the second call to `top` would not show up in the forward slice starting at the first `top` because there is no data dependency

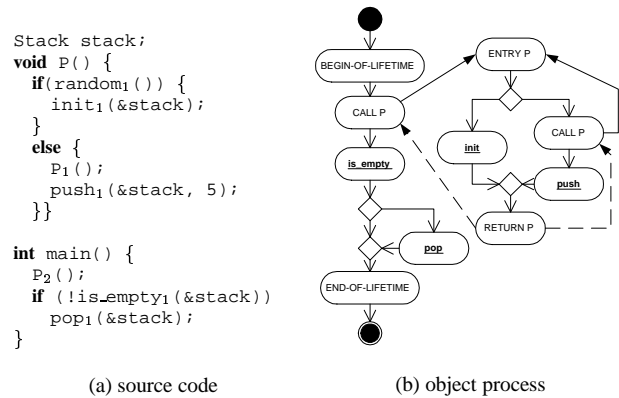


Figure 3. Example object process graph.

in between. On the other hand, the usages of the value returned by the first `top` would be in the slice, but not in the object process graph.

```

Stack *Create () {
  Stack *cs = (Stack *)malloc1 ();
  init (cs);
  return cs;
}

Stack *P0 () {
  Stack *ps;
  if (random1 ())
    return Create1 ();
  else {
    ps = P1 ();
    push (ps, 5);
    return ps;
  }
}

void swap (Stack *sp1, Stack *sp2) {
  Item i1, i2;
  if (!is_empty(sp1) && !is_empty(sp2)) {
    i1 = pop (sp1); i2 = pop (sp2);
    push (sp1, i2); push (sp2, i1);
  }
}

int main () {
  Stack *(*fp)(), *ms;
  Stack stack;
  init (&stack);
  if (random2 ())
    fp = Create;
  else
    fp = P;
  ms = fp ();
  /* Create2 () or P2 () */
  swap1 (ms, &stack);
  ms = &stack;
  swap2 (ms, &stack);
}

```

Figure 4. Source program.

4 Statically Detectable Objects

An object is created at run time either implicitly as part of a new activation record for a routine call in case of stack variables or by way of a heap allocation routine in case of heap objects. Because of recursion and loops we cannot statically know the set of all objects in the general case. What we can identify statically are the declarations of the stack variables and the allocator calls for heap objects—named *constructors* in the following. A potentially infinite set of dynamic objects corresponds to one such *static object*

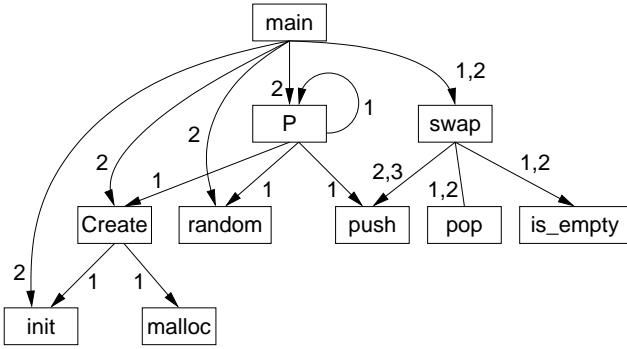


Figure 5. Call graph of program in Figure 4.

that is identified by a certain static constructor, and the behavior of the static object must cover the potential behavior of all its associated run-time objects.

4.1 Call Paths

We will distinguish stack from heap objects. Stack objects are identified by their declaration, heap objects by their static allocation site. Yet, there may be different paths in the call graph to this allocation site. On each path, a different heap object will be created. To further disambiguate heap objects allocated at the same allocation site, a heap object is represented as a pair of an allocation site and a call path leading to this allocation site.

Figure 4 shows a program, whose call multigraph is shown in Figure 5; the indices in the source identify the call sites. Ignoring the recursive call in P for a moment, we can identify two different paths in the call multigraph from main to malloc . Hence, the corresponding heap objects would be: $O_1 = (\text{main} \rightarrow \text{Create}_2 \rightarrow \text{malloc}_1)$ and $O_2 = (\text{main} \rightarrow P_2 \rightarrow \text{Create}_1 \rightarrow \text{malloc}_1)$. Beyond the two heap objects, there is a local variable stack in main that constitutes an object, O_3 . Yet, local and global variables need not be disambiguated with a call path.

Recursive call paths cannot be described by finite sequences of call sites. Instead, we identify objects created in recursive cycles by regular path expressions, which permit a finite description of call paths of arbitrary length.

The following grammar rules define path expressions:

1. $R ::= \text{call-site}$
2. $R ::= \text{call-site} \rightarrow R$
3. $R ::= (R|R)$
4. $R ::= (R)^*$

The first rule says that a routine call was executed to construct the object. The second rule describes that a call site leads to a constructor along path expression R . The third rule represents a choice between two call paths. The fourth

rule states that a recursive cycle R could be entered and executed zero to infinite times.

Due to the recursive call in P in Figure 5, another object can be described by the call path: $O_4 = (\text{main} \rightarrow P_2 \rightarrow P_1^* \rightarrow \text{Create}_1 \rightarrow \text{malloc}_1)$. This path expression subsumes the aforementioned path expression for $O_2 = (\text{main} \rightarrow P_2 \rightarrow \text{Create}_1 \rightarrow \text{malloc}_1)$. Hence, it is sufficient to distinguish the three heap objects O_1, O_3, O_4 for this example.

4.2 Static Objects and Points-To Information

To detect all operations where a given object may be involved in the presence of pointer expressions, points-to information is necessary. Points-to analyses detect all potential aliases for objects. Points-to analyses differ in their context- and flow-sensitivity. Context-sensitive analyses distinguish different points-to relations at different call sites, whereas context-insensitive points-to analyses unite all possible calling contexts. Flow-sensitive analyses regard the flow of control, whereas flow-insensitive analyses do not care about the order of pointer assignments.

Because many languages (for instance, C) allow a programmer to circumvent the type system, a points-to analysis for general programs in these languages cannot safely rely on type information. For this reason, we model the points-to relation by way of an *abstract storage*. The abstract storage consists of a set of *abstract blocks* that represent contiguous pieces of memory. Three different kinds of abstract blocks are distinguished: *local blocks*, *heap blocks*, and *non-local blocks*. Local blocks represent non-static local stack variables and formal parameters, non-local blocks represent blocks reachable via formal input parameters (from the viewpoint of the callee), and heap blocks are memory blocks allocated by the given routine either locally through malloc and similar operating system calls or through a (transitively) called user-defined routine. Heap blocks are additionally associated with call paths as already described in the previous section. Global variables and static local variables are treated as parameters, i.e., as if they were passed indirectly: An implicit formal parameter is added that points to the global variable or static local variable, respectively.

Blocks are the physical representation of an object. Each block represents one statically detectable object specific to one points-to context. Because multiple different points-to relations may hold at a call site in a context-sensitive points-to analysis, an object may correspond to multiple blocks.

5. Extracting Object Processes

This section describes the algorithm that extracts object processes for a given criterion. The criterion specifies the

```

procedure Process_Extraction (Configuration, CFG_Node) is
begin
  if not Already_Marked (CFG_Node, Configuration) then
    Mark (CFG_Node, Configuration);

    if Is_Relevant (CFG_Node, Configuration) then
      Generate_Process_Node (CFG_Node);

      case Type (CFG_Node) is
        when End_of_Lifetime =>
          -- destructor; for deallocator see discussion
          return;
        when Call => -- forward traversal
          for each Callee in
            Callees (CFG_Node, Configuration)
          loop
            if Needs_Visit_Callee
              (Configuration, CFG_Node, Callee)
              and not Primitive (Callee)
            then Process_Extraction
              (Translate_fw
               (Configuration, CFG_Node, Callee),
               Callee);
            end if;
          end loop;
        when Return => -- backward traversal
          for each Call_Path in
            Callers (Configuration.Call_Path)
          loop
            Process_Extraction
              (Translate_bw
               (Configuration, CFG_Node, Call_Path),
               last (Call_Path));
          end loop;
        end case;

      for each S in Successors (CFG_Node)
        in Intraprocedural_CFG
      loop -- intraprocedural control-flow tracing
        Process_Extraction (Configuration, S);
      end loop;
    end if;
  end Process_Extraction;

```

Figure 6. Object process extraction.

object whose object process is to be extracted. The object is represented as a set of blocks as described in Section 4.

The algorithm is based on a context-sensitive traversal of the interprocedural and the intraprocedural CFG of the source program annotated with context- and flow-sensitive points-to information. The algorithm is independent from the underlying points-to analysis. It will even work if only context-insensitive points-to information is available (although its results will be weakened). The algorithm assumes that the underlying points-to analysis has resolved calls via function pointers.

The algorithm has two phases. In the first phase, a corresponding object process node is added for each relevant CFG node. A CFG node is relevant if it starts or ends the lifetime of the selected object, if it references the relevant object (as established by the points-to information), or if it is a call that leads to a relevant CFG node. In the second phase, these object process nodes are connected by control flow edges. Additional merge and branch nodes are added in this phase if necessary.

5.1 Phase 1: Creating Object Process Nodes

Figure 6 shows the context-sensitive traversal of the intra- and interprocedural CFG that creates object process nodes for relevant CFG nodes.

The two input parameters to the algorithm are as follows:

CFG_Node denotes the currently processed node of the CFG.

Configuration records the aliases of the relevant object in the context of the current *CFG_Node*: $Configuration = (blocks \Rightarrow B, call_path \Rightarrow C)$ where B is the set of blocks that represent the same object in different points-to contexts and C is a call path in case of a heap object.

The analysis starts with an initial call to

$Process_Extraction((b_c, cp_c), entry_point_c)$

at the specified *entry_point* that denotes the constructor, c , of the object. Local and global variables are constructed by declaration points, which are represented in CFGs by *begin_of_lifetime* nodes; heap objects are constructed by calls to allocators. The initial configuration contains the blocks b_c constructed by c and the call path cp_c to c (excluding the constructor itself) if the constructor is an allocator that creates a heap object; otherwise, cp_c is irrelevant.

Generally, one will call *Process_Extraction* separately for each block representing the object in a different points-to context, and would then obtain a set of object process graphs that depend on the points-to context. If one is interested in several points-to contexts at once, one can call *Process_Extraction* with a set of blocks instead. The algorithm handles both cases.

It might be necessary to enter the same routine several times due to the context-sensitivity of the points-to information, but only once for the same configuration.

When a CFG node is reached that is relevant for the behavior of the selected object, an object process node is created. A CFG node is relevant if it references (defines or uses) the object as determined by the points-to information. The transitive relation *reference* also subsumes that the object is constructed or destructed. Points-to analysis reveals the set of all abstract blocks that are (transitively) referenced by the operation. If a block of the current configuration is member of this set, the node is relevant.

$$Is_Relevant(CFG_Node, conf) = references(CFG_Node) \cap conf.blocks \neq \emptyset$$

Destructors. When a destructor is reached, the extraction for the current *CFG_Node* ends. In case of stack variables, the *end_of_lifetime* node for the object is the unique program point where an object ceases to exist. As opposed

to stack variables, heap objects may have multiple destructors: calls to deallocation routines, like `free` in C/Unix, and the final return node of the program.

Unfortunately, one cannot safely determine destructors for heap objects in general. Calls to `free` will be handled as normal calls and, thus, appear in the object process graph, however. For stack variables as well as heap objects, the extraction will resume in the recursion of *Process_Extraction* with a previous node as there may be multiple paths to the same destructor.

Calls. Calls need to be followed if the callee references the selected object. This information can be determined by mapping the blocks representing the object in the caller to the blocks representing the object in the callee. If at least one block has an alias in the callee that is actually (transitively) referenced, the call is relevant:

$$\text{Needs_Visit}_{\text{callee}}(\text{conf}, \text{cs}, \text{callee}) = \text{references}(\text{callee}) \cap$$

$$\text{Translate_Blocks}_{fw}(\text{conf.blocks}, \text{cs}, \text{callee}) \neq \emptyset$$

where function *Translate_Blocks_{fw}* is a function that maps blocks from caller to callee at a given call site (details of this mapping follow in Section 6.2).

A call site might call more than one routine if the call is through a function pointer. Function *Calles* returns all potentially called routines at the current call site depending upon the current points-to context.

The user can specify primitive operations, i.e., operations that are to be considered atomic. For instance, routines in the provided interface of a component will likely be considered atomic. Primitive routines will not be entered.

When entering the called routine, the configuration at the call site must be transformed to the callee's configuration. For this transformation, blocks representing the object in the caller's context need to be mapped onto the blocks of the callee, which is achieved by *Translate_{fw}* (cf. Section 6.2).

Returns. For objects passed from callers to callees (and possibly back), a forward traversal following interprocedural call edges is sufficient. Only if the lifetime of the object starts in the callee and then the object is passed to the caller at a return statement, the algorithm has to follow interprocedural control flow from callees to callers. This can only happen for heap objects created by the callee. If the object were a local variable, the program would be erroneous since local variables cease to exist upon return of the call. We exclude such incorrect programs.

Handling return statements is similar to call statements, yet complicated by the fact that the call path needs to be considered in case of heap objects. When the traversal continues at a caller after a return node of the callee, the configuration at the return node must be transformed to the caller's configuration analogously to the traversal from caller to callee. Additionally, the call path must be adjusted so that it reflects the new current position (in the context

of the caller, the caller is immediately before executing the call). Details of function *Translate_{bw}* that achieves this mapping follow in Section 6.2.

Function *Callers* determines the call paths that are used to identify the call sites where the analysis has to resume with help of the regular path expression of the relevant heap object (including call sites that may call the current routine via function pointers):

$$\text{Callers}(cp) = \begin{cases} \{\alpha a\} & \text{if } cp = \alpha a b \\ \{\alpha a_1, \dots, \alpha a_n\} & \text{if } cp = \alpha(a_1 | \dots | a_n) b \\ \{\alpha, \alpha \beta a\} & \text{if } cp = \alpha(\beta a)^* b \end{cases}$$

where α and β are possibly empty regular expressions and a , a_1 , and a_2 are call sites; b is a call site or a regular expression $b = (c)^*$ where c is a call site. The first expression above simply states that we will visit the caller if there is just one. Then, in case of alternatives, we will visit every alternative caller. The last expression states that in case of cycles, we need to visit the caller from which the cycle is entered (thus omitting the cycle altogether) and the last caller in the cycle. Note that is sufficient to handle the cycle once, so that the $*$ may be omitted.

Function *last*(cp) used in the recursive call of *Process_Extraction* yields the last call site in the path expression cp that is to be visited next.

For instance, when extracting the object process graph for object $O_4 = (\text{main} \rightarrow P_2 \rightarrow P_1^* \rightarrow \text{Create}_1 \rightarrow \text{malloc}_1)$ of the program in Figure 4, we need to ascertain the callers at the return statement of *Create* where the analysis is to proceed. The working call path at the return statement is $(\text{main} \rightarrow P_2 \rightarrow P_1^* \rightarrow \text{Create}_1)$ since the *malloc* is omitted in the initial call to *Process_Extraction*. There are two possible callers according to the call graph. Yet, only the call site *Create₁* is relevant with respect to the call path of O_4 as returned by function *last*. The new call paths are then $(\text{main} \rightarrow P_2)$ and $(\text{main} \rightarrow P_2 \rightarrow P_1)$ according to the last rule of *Callers*. Due to the two call paths, routine P will be entered twice. Visiting P with $(\text{main} \rightarrow P_2)$, the next call path at the return statement in P is $(\text{main} \rightarrow P_2)$ and call site P_2 will be visited next. Visiting P with $(\text{main} \rightarrow P_2 \rightarrow P_1)$, the next call path at the return statement in P is $(\text{main} \rightarrow P_2)$. Yet, P has previously been visited with this call path and, hence, need not be visited again.

As opposed to forward traversal, it is not necessary to check whether the returned object is actually used by the caller. The caller is entered at any rate. In the rather unlikely event that the object is not referenced by the caller, the caller itself does not return the object and the traversal of the caller is a dead-end, i.e., transitive callers do not need to be analyzed and no further node needs to be created.

Note that we need to traverse every relevant caller in case of backward interprocedural traversal—be it a primitive or a client routine—to finally reach the operations initiated by the client. If the operations taking place from the allocator in the component to the first operation in the client should be omitted, additional status information can be added to the algorithm to suppress all operations in the component’s code. To keep the description simple, we omit this minor detail here.

5.2. Phase 2: Edges and Branch and Merge Nodes

To preserve that operations on objects are conditional, i.e., that there is a path that circumvents the operation, predicates need to be added in the object process graph even when they do not refer to the relevant object. Yet, branch nodes are only inserted when other nodes of the object process graph (other than the initial node representing the object’s constructor) are control dependent on the predicate. We also add the predicates on which the operation is transitively control dependent so that all conditions that must hold to execute an operation can be traced back to the original program. On demand, cascading predicate nodes may be collapsed in the representation analogously to removing epsilon edges from nondeterministic state automata.

Once all object process nodes have been created they need to be connected through interprocedural and intraprocedural control-flow edges. The connection has to reflect the original control flow.

Interprocedural edges are added from call nodes in the object process graph to the corresponding entry node of the callee and from the return node of the callee back to the call node, depending on the kind of parameter passing. If the caller passes the object to the callee as value parameter, only a call edge needs to be added. The traversal in *Process_Extraction* has already extracted the object process graph for the corresponding formal parameter in the context of the callee. When *Process_Extraction* for the callee returns, the extraction of the object process for the actual parameter continues and yields an object process subgraph that is independent from the object process subgraph for the formal parameter of the callee to which the actual parameter has been passed by value. The call node in the whole object process then acts as a kind of dispatcher of the object’s value into two independent object process subgraphs. The operations in the two subgraphs are applied to two physically different objects that share the same operation sequence prefix.

If a heap block is returned to the caller, a return edge is added from the callee to the caller. Likewise, in case of call-by-reference or copy-in/copy-out, the return node created for the callee’s return node and the call node representing the call site in the object process graph will be connected by a return edge to represent that all operations in the context

of the caller after the call take place after the operations in the callee.

An intraprocedural control-flow edge is added from object process node S to node E if and only if there is a path from the CFG node corresponding to S to the CFG node corresponding to E in the intraprocedural CFG and if there is no other CFG node on this path that has a corresponding object process node. Additionally, conditional edges may be annotated with *true* or *false* accordingly.

Merge nodes represent the merging of two or more intraprocedural control flows. They are optional and allow to describe processes by graphs where each node (except branch nodes and merge nodes) has one intraprocedural control-flow successor and one intraprocedural control-flow predecessor.

Merge nodes are introduced to the process graph when all other nodes were already generated. For each node of the process graph with more than one intraprocedural predecessor, one merge node is introduced. The edges from the predecessor are re-linked to the merge node and a new edge that leads from the merge node to the original node is introduced.

5.3. Complexity

In general, each node of the CFG must be processed n times, where n is the number of configurations at this node. In the worst case, each CFG node references the object. Then, the resulting object process graph is N times as large as the original CFG, where N is the maximum number of configurations per node. However, empirical data by Wilson suggest that there are less than two configurations per routine on average [11], and for normal programs, only few CFG nodes will reference the object.

Our own experimental results for *concepts* [8] (8 KLOC of C, 143 routines, 74 calls to `malloc`) suggest that object processes are fundamentally smaller than the original CFG: 1,320 object processes were extracted (770 for heap and 550 for stack objects). The average number of nodes of object process graphs is 209 (minimum 5, maximum 850). The results were computed in 397 seconds on an AMD Athlon XP 1700+ processor running Linux. The real costs for trace extraction incur for the points-to analysis.

6. Extraction in the Presence of Pointers

Our technique to extract object processes statically is based on the CFG. Yet, in case of instances of components, control-flow information is not sufficient. Points-To analysis is required to obtain information about indirectly accessed memory objects. This section introduces the points-to information used for extracting object processes. Based

on this information, the functions used by the generic algorithm in Section 5 are refined.

6.1. Integrating Points-To Information

To maximize the precision, we chose Wilson’s points-to analysis, which is both context- and flow-sensitive [11]. However, our technique is independent from the underlying points-to analysis. This section describes the concepts necessary to understand how objects can be traced statically in the presence of pointers.

Wilson’s pointer analysis is based on *partial transfer functions (PTF)*, which summarize the context-sensitive effects of a routine. PTFs are partial functions because they do not describe the effect of the routine call under all circumstances, but only for those that actually occur in the program.

In our example of Figure 4, each routine has exactly one PTF, except *swap*. For the latter, two calling contexts and therefore two PTFs can be distinguished: One in which the two formal pointer arguments point to different blocks, and one in which they point to the same block.

Each PTF possesses a set of abstract blocks to represent accessed memory. By definition, no two abstract blocks of one PTF are aliases, and each abstract block is associated with exactly one PTF. At run time, each allocated memory block can be associated with one abstract memory block of a specific PTF when the function accesses the block under the conditions of the PTF. The storage abstraction allows a PTF to be re-used for equivalent calling contexts, such that a routine body does not need to be re-analyzed for other, equivalent call sites.

A PTF is defined as a mapping from an input topology to an output topology, which represent the points-to relation before and after the call, respectively:

$$PTF(F) : Topology_{in} \rightarrow Topology_{out}$$

where $Topology_{in}$ is an abstract description of the points-to relation at a call site for F just before the call and $Topology_{out}$ describes the points-to relation after the call. The transition from input to output topology summarizes the effect of the call to F .

Topologies are modeled as graphs to represent the points-to relation: $Topology = Graph(N, E)$ where the set of nodes N are blocks and the set of edges $E \subseteq N \times N$ represents the points-to relation: (n_1, n_2) specifies that n_1 points to n_2 .

The two different PTFs for *swap* are shown in Figure 7. The input topology in Figure 7(a) describes the non-aliasing situation, whereas the input topology in Figure 7(b) shows that the two formal input parameters $*sp1$ and $*sp2$ are aliases. Because routine *swap* does not change its in-

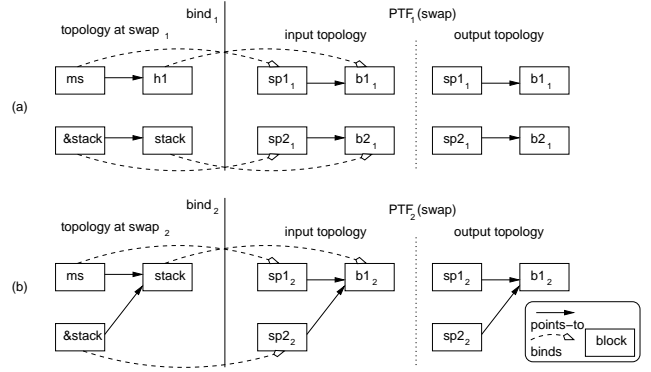


Figure 7. Points-to relations for *swap*.

put values, the output topologies are identical to the input topologies.

A PTF can be applied by mapping the blocks of the caller at the call site (the topology at the call site) to the blocks of the input topology of the callee’s PTF and by making the transition from input to output topology. Part of the representation of points-to information is a binding that describes how the blocks of the topology at the call site map to blocks of the input topology of the callee’s PTF:

$$Binding : Blocks \rightarrow Blocks$$

Every call site is associated with bindings that relate the topology at the call site to the input topologies of all PTFs for the callees. A binding may map different blocks in the call-site topology to the same block in the callee’s input topology, but not vice versa.

The bindings, $bind_1$ and $bind_2$, for the two calls from procedure *main* to *swap* of the example program in Figure 4 are shown in Figure 7: $bind_1 = \{(ms, sp1_1), (h1, b1_1), (stack, b2_1)\}$ at call site $swap_1$ and $bind_2 = \{(ms, sp1_2), (stack, b1_2)\}$ at call site $swap_2$, where $*sp1_2$ and $b1_2$ are aliases.

Figure 8 summarizes the conceptual model that connects the concepts from the three different domains of object processes, points-to information (shaded box), and programming language constructs.

6.2 Refinement of Translations for Pointers

When a called routine is entered in the context-sensitive traversal to extract object processes (cf. Figure 6), the configuration at the call site needs to be mapped to the configuration of the callee; similarly so for backward traversal from a callee to a caller in cases where heap blocks are returned to a caller. This section refines the two functions $Translate_{fw}$ and $Translate_{bw}$ that achieve these mappings.

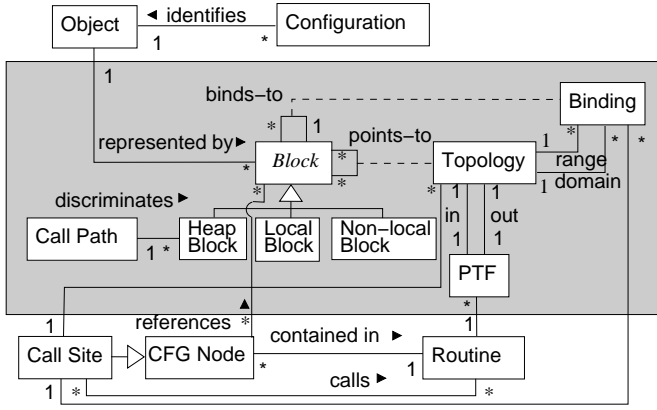


Figure 8. Conceptual model.

Blocks are used to represent an object in different points-to relations. When entering a called routine, the blocks representing the object in the context of the caller need to be mapped onto blocks representing the object in the callee, which may be achieved through the binding function identified by the caller's and callee's topologies:

$$\text{Translate_Blocks}_{fw}(blocks, cs, callee) = \bigcup_{\substack{b \in blocks \\ bind \in Bindings(cs, callee)}} bind(b)$$

where *Bindings* identifies all bindings between a call site and a callee (attributes correspond to the conceptual model in Figure 8):

$$\begin{aligned} Bindings(call_site, callee) = & \{ bind : Block \rightarrow Blocks \mid \\ & bind \in call_site.bindings \wedge \\ & \exists (ptf \in callee.PTFs) bind.range = ptf.in \end{aligned}$$

For instance, when extracting the object process graph for object O_3 of the program in Figure 4, namely, local variable *stack*, we need to map *stack* to the input parameter of *swap* at call site $swap_1$. At this call site (cf. Figure 7), there is only one binding, $bind_1$, that binds *stack* to the non-local block $b2_1$.

From the viewpoint of the callee, it does not matter whether input parameters represent stack variables or heap objects. That is why all input parameters may be represented alike as non-local blocks. This way, the PTF of the callee is independent from the type of the input block and may, hence, be re-used for different kinds of blocks. Because heap blocks of the caller will appear as non-local blocks in the callee and because call paths are only relevant

for heap blocks, call paths may be left out in the transformation from the caller's configuration to the callee's configuration:

$$\begin{aligned} \text{Translate}_{fw}(c, cs, callee) = & \\ & (\text{Translate_Blocks}_{fw}(c.blocks, cs, callee), \\ & \text{undefined}) \end{aligned}$$

Consequently, $\text{Translate}_{fw}(\{stack\}, swap_1, swap) = (\{b2_1\}, \perp)$, resulting in the object process graph on the left-hand side of Figure 9(c). At call site $swap_2$, we obtain a different configuration because *stack* binds at a different block $b1_2$ that belongs to another PTF. Because of the new configuration, *swap* needs to be re-analyzed, resulting in the object process graph on the right-hand side of Figure 9(c). Because of the alias situation, all operations on $*sp1$ and $*sp2$ apply to the same object, and an object process graph different from the first object process graph results. Due to this alias situation, *pop* is executed twice on the same object without checking in between whether the *stack* is empty, as can be seen on the right-hand side of Figure 9(c). The object process graph for O_3 clearly reveals this discrepancy to the likely protocol. If this problem is not fixed, a run-time error will occur when the *stack* has exactly one element. Another issue revealed by the object process graphs in Figure 9 is that the *release* operation is not called for any of the objects O_1 , O_3 , and O_4 .

In case of backward traversal, the blocks of the callee need to be mapped onto blocks of the caller. Additionally, the call path needs to be adjusted because a heap object is traced. The new call path was already determined by function *Caller* and is given as argument *cp* (cf. Section 5.1).

$$\begin{aligned} \text{Translate}_{bw}(c, return, cp) = & \\ & \left(\bigcup_{\substack{b \in c.blocks \\ bind \in Bindings(last(cp), return.contained_in)}} bind^{-1}(b), \right. \\ & \left. cp \right) \end{aligned}$$

Because different blocks of the caller may be mapped on the same block of the callee, function $bind^{-1}$ actually returns a set of blocks in the general case. However, in case of returned heap blocks, the binding is indeed invertible and function $bind^{-1}$ yields exactly one block. For the technical details of the binding see [11].

7. Conclusions

This paper introduced object processes that describe all potential operation sequences for individual stack and heap variables. Object process graphs may be used for protocol

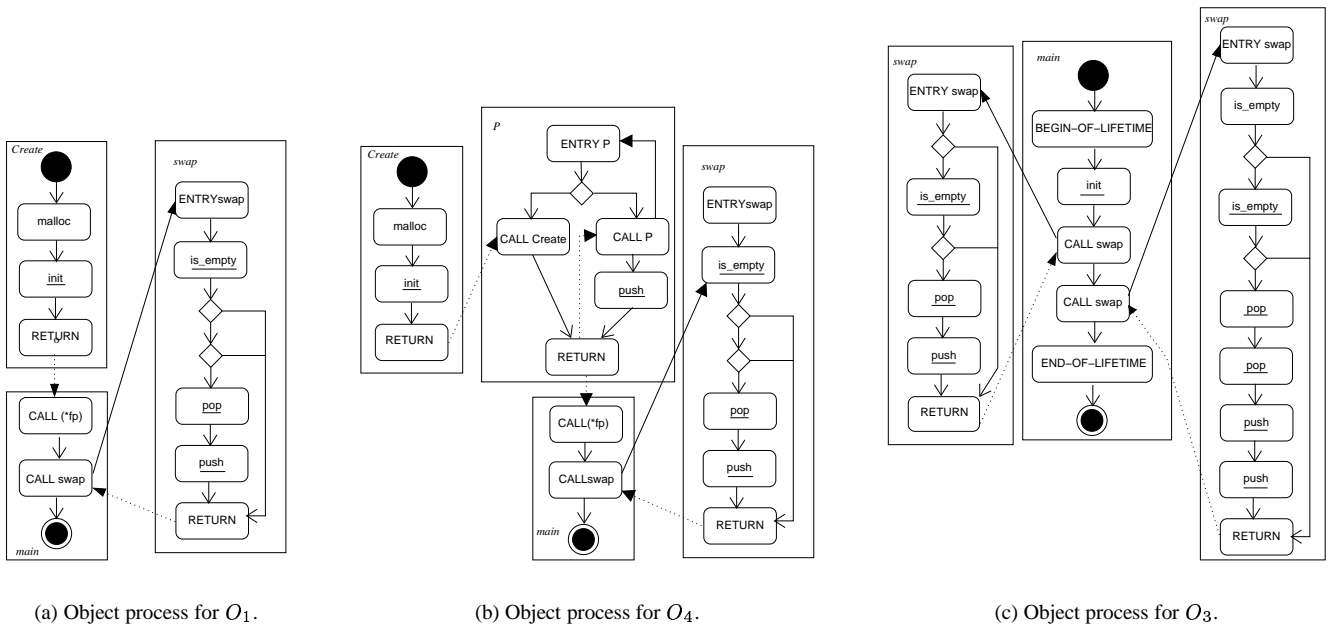


Figure 9. Process graphs for stack objects of Figure 4.

recovery and validation and program understanding in general.

We described an algorithm to extract object process graphs that integrates points-to information to obtain precise object process graphs in the presence of pointers. Our preliminary empirical studies have shown that the algorithm is efficient and yields object process graphs that are considerably smaller than the original CFG.

Like every static analysis, the extraction of object process graphs has its limitations. The limits are set by the accuracy of the available points-to information. In the general case, aliasing is statically undecidable [10]. For instance, one cannot statically know whether two pointers relate to the same heap block as part of a recursive data structure. Consequently, in many cases, only may-alias information is available. However, because we use object process graphs in a semi-automatic method for protocol recovery and validation, the user can be asked to validate the results.

References

- [1] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *Proceedings of the 8th International Symposium on Foundations of Software Engineering for Twenty-first Century Applications*, pages 50–59, 2000.
- [2] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.
- [3] T. Heiber. Semi-automatische Herleitung von Komponentenprotokollen aus statischen Verwendungsmustern (semi-automatic recovery of component protocols from static usage patterns). Diplomarbeit, 2001.
- [4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.
- [5] D. F. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1997.
- [6] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the International Conference on Software Engineering*. ACM, 1997.
- [7] R. Koschke and Y. Zhang. Component recovery, protocol recovery and validation in bauhaus. In J. Ebert, B. Kullbach, and F. Lehner, editors, *3rd Reengineering Workshop, Bad Honnef, Germany*, volume 1/2002 of *Fachberichte Informatik*, pages 73–76. University of Koblenz-Landau, 2001.
- [8] C. Lindig. Concepts 0.3e. Available at <http://www.gaertner.de/~lindig/software/>, 1999.
- [9] K. Olender and L. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, Jan. 1992.
- [10] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, Sept. 1994.
- [11] R. Wilson. *Efficient, Context-Sensitive Pointer Analysis*. PhD thesis, Department of Electrical Engineering, Stanford University, Dec. 1997.