Dynamic Protocol Recovery

Jochen Quante, Rainer Koschke
University of Bremen, Germany
http://www.informatik.uni-bremen.de/st/
{quante,koschke}@informatik.uni-bremen.de

Abstract

Dynamic protocol recovery tries to recover a component's sequencing constraints by means of dynamic analysis. This problem has been tackled by several automaton learning approaches in the past. These approaches are based on the sequence of component method invocations only.

We introduce a new dynamic protocol recovery technique based on object process graphs. These graphs contain information about loops and the context in which methods are being called. We describe the transformation of a set of these graphs to a protocol automaton. The additional input, compared to the sole sequence of method calls, results in a more detailed protocol.

In a case study, we compare the resulting protocol automata of our approach to those of several existing automaton learning approaches.

1 Introduction

Modern static bug finders and security vulnerability detectors perform sophisticated checks. These tools are based on advanced analyses, such as global control and data flow analyses, model checking, or abstract interpretation. Examples for such tools are FindBugs, PMD, Lint, Grammatec/CodeSonar, Coverity/Prevent, and Polyspace.

Many of these tools find generic defects, such as potential null pointer dereferences or buffer-overflow problems. They may be used to limit the effects of weak programming languages (e. g., no index range checks at runtime) or frequent programming errors. Although useful, such tools do not help in finding application-specific defects, where a component is not used according to its specification. More advanced tools may detect such problems by allowing an analyst to create customized checks. Engler et al. [5], for instance, developed a technique where checkers can be specified as finite state automata (FSA). These FSA specify the allowable sequences of operations of a component—its protocol—in terms of a regular language. Engler's technique checks code by traversing the control flow graph and symbolically executing the operations. The technique has been successfully transferred to industry by way of Engler's spin-off Coverity. Coverity runs its checkers regularly on large open-source projects, such as GNU/Linux. These checks have discovered thousands of defects in GNU/Linux, including several security alerts. Similar techniques are used by GrammaTech's CodeSonar, too.

To write application-specific checkers, an analyst needs to know the protocol of a component in the first place. However, in many cases, only the syntactic interface of a component is defined as this is usually supported and enforced by the programming language. Semantic constraints are often undefined. For example, an `init` method might have to be called before any other method may be called. Such sequencing constraints, which we call the *protocol*, are often not specified. In some cases, they are informally included in the documentation, but this makes it impossible to check compliance of applications with those protocols automatically. If the protocol was available in a machine-readable way, adherence could be checked, and this could help to make software more reliable and less erroneous. It is therefore desirable to be able to reconstruct the protocol of a component.

Contributions. This paper addresses the issue of protocol recovery. We describe a novel technique using object process graphs, a finite representation of the sequences of operations for particular objects extracted from source code or via dynamic analysis. We compare the new technique quantitatively to other existing automaton learning approaches, which are also using dynamic analysis. For this comparison, we propose a novel similarity measure of the resulting FSA.

Overview. The remainder of this paper is organized as follows. Section 2 describes related research

in protocol recovery. Section 3 describes the concepts of object process graphs (OPG), and Section 4 explains the technique used to transform a set of OPGs to a protocol automaton. Section 5 introduces a new metric for automaton comparison. The case study in Section 6 compares our approach to existing protocol recovery techniques based on automaton learning.

2 Related Research

Protocol recovery can be tackled by static or dynamic analyses, or by a combination of both. The static approach has the advantage of finding all potentially possible calling sequences, while a dynamic approach can build only upon those sequences that really occurred in a given set of program runs. A static approach is always based on conservative assumptions, which may result in infeasible paths [4], while this cannot happen in the dynamic case. In this paper, we focus on dynamic approaches to protocol recovery. A comparison of static and dynamic trace extraction can be found in another paper of ours [14].

Dynamic protocol recovery has been subject to prior research. Researchers have mostly focused on automaton learning in this area: Program traces, i. e., sequences of invocations of a component's methods, are fed into a learner which produces an automaton that accepts the given traces – and more [19, 1, 15]. Constructing an automaton that accepts exactly the given set of traces (*prefix tree acceptor*, *PTA*) is simple but not useful. This automaton only represents the concrete sequencing information of the regarded applications that are using the component, although other usages might be allowed as well. Therefore, generalizations are necessary. Most automaton learning techniques apply different heuristics to transform the PTA to a more general form, thus reducing the number of states and transitions. On the other hand, when generalizing too much, the automaton becomes useless as well (*overgeneralization*). In the extreme, the protocol automaton could be reduced to a single state with transitions on all possible events that lead back to this state, which allows any sequence of operations. Therefore, the challenge here is to apply the right amount of generalization to get a protocol automaton that is most useful and meaningful for a particular purpose. For example, one goal may be to get a protocol that is as close to the real specification as possible.

k-tails method. Biermann et al. [3] introduced the k-tails method. It starts with the PTA and then merges states that are indistinguishable in the set of accepted output strings up to a given length k. Steven P. Reiss [16] uses an extended k-tails algorithm for learning an FSA that represents a set of traces. His approach is not originally intended for protocol recovery, but for compressing a trace. Nevertheless, the resulting FSA can as well be regarded as an interface's protocol when the trace consists of interface interactions only.

Successor method. A straight-forward approach is to represent each method by one state. Transitions between states indicate legal sequences of method calls. Richetin et al. call this the "successor method" [17].

Whaley. As pointed out by Whaley et al. [20], the successor method has some drawbacks: Firstly, only knowing the last method cannot capture the proper behavior of the object in many cases, because their sequencing constraints are often more complex. Secondly, there may be methods that are state preserving; including these in the model destroys its accuracy, since state-preserving methods may often be called in any state. As a consequence, Whaley et al. propose to use multiple FSA per component. Each FSA describes the protocol of only a subset of methods ("model slicing"), e. g., one that implements an interface or accesses a particular field. They also propose to ignore state-preserving methods during the construction of the automaton and to just annotate state-preserving methods to those states. However, potentially important information is lost this way.

sk-strings approach. Raman and Patrick [15] modified the k-tails method for stochastic automata. They merge states that are indistinguishable for their top s percent of the most probable k-strings. A k-string does not need to end in an accepting state when it has length k. The result is a *probabilistic* FSA (PFSA) that is annotated with transition frequencies. Glenn Ammons [1] uses this approach to infer an automaton that represents the protocol. In a postprocessing step, a *corer* removes infrequently traversed edges from the PFSA. Ammons applied this technique to XWindows programs and found several errors in the use of XWindows components.

Other approaches. In all these approaches, states represent methods or sequences of them. A different idea is to introduce automaton states for different object states or abstractions thereof. Xie et al. [21] follow this approach. By construction, the resulting state machines are different from the other approaches' and thus hardly comparable to them.

Lo and Khoo [11] propose a protocol recovery architecture called SMArTIC, which consists of four consecutive steps: First, erroneous traces are filtered out by detecting common behavior. Next, traces are divided

into groups of "similar" traces (clusters). Then, protocol automata are generated for each group. Using a PFSA specification miner is proposed for this step. Last, the automata are merged into a single one. The authors claim that this architecture improves accuracy, robustness and scalability.

Sekar et al. [19] use a similar technique to learn state automata. They take into consideration the static *program state* which can be extracted from the call stack. The resulting automata are used for detecting anomalous program behavior. This is closely related to violations of a given protocol, and the idea could be regarded as a lightweight DOPG variant.

Salah et al. [18] apply dynamic analysis to extract representative usage scenarios for a component. Their analysis is also based on method invocation sequences. It computes *canonical sets* of such sequences, using a trace similarity metric. The results are combined and generalized into regular expressions using common subsequence detection. This approach tries to find typical usage scenarios, while we try to detect all possible usage scenarios.

Differences to our work. Since these approaches are solely based on the sequence of method invocations of the investigated component, they cannot distinguish between loops and repeated invocation of a method. Also, automaton learning requires negative examples to prevent the resulting automaton from overgeneralizing, but these are never generated by real program runs. On the other hand, generalization is necessary when recovering a protocol, because traces are only samples of all possible method invocation sequences. The problem here is to find the right compromise between generalization and specialization.

Our approach is based on object process graphs. Compared to the automaton learning approaches' input, which is just the call sequence information of a component, a dynamic object process graph (DOPG) contains more information: It describes the overall control flow of an application with respect to a single instance of a component (i. e., one object). In particular, a DOPG contains exactly those parts of the control flow graph that are relevant for this object in terms of control flow. In the following sections, we describe how DOPGs can be used for protocol recovery.

3 Concepts

A *protocol* in the sense of this paper describes the sequencing constraints that are imposed on a component's methods: It tells us in which order these methods may be called. The sequencing protocol is part of the *interface* of a component.

Protocol validation aims at checking whether all actual sequences of operations conform to the protocol. All actual sequences of operations form a language; likewise, a protocol can be considered a language. Hence, protocol validation needs to check whether one language is a subset of another language. This test is in general only possible for regular languages. Consequently, regular languages or finite state automata (FSA) are often the notion of choice for protocol validation [12]. Therefore, and to be comparable to other protocol recovery approaches, we also choose FSA to represent a protocol. In such protocol automata, automaton states represent program states, and transitions correspond to operations on a component.

We represent the actual behavior extracted from a program for individual objects as object process graphs. An object process graph (OPG) is a projection of an *interprocedural control flow graph (CFG)* specific to one object. It contains only those parts of the CFG that are relevant for the given object with respect to control flow. An OPG is defined as a graph $OPG := (N, E)$ where each node $n \in N$ and each edge $e \in E$ can be of one of the following types:

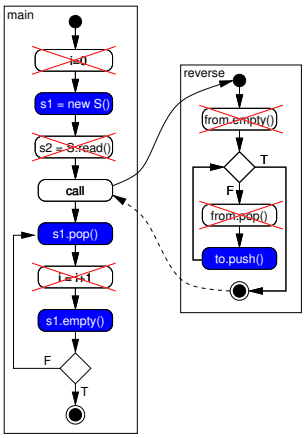
$$\begin{aligned} T_N(n) &\in \{\texttt{start}, \texttt{create}, \texttt{access}, \texttt{decision}, \texttt{final}, \\ &\qquad \texttt{call}, \texttt{entry}, \texttt{return}, \texttt{atomic\_call}\} \\ T_E(e) &\in \{\texttt{call}, \texttt{return}, \texttt{seq}, \texttt{true}, \texttt{false}\} \end{aligned}$$

Each node in the OPG represents a location in the program. The `start` node indicates the entry point. A `create` node denotes the creation of an object by way of a variable declaration or allocation.

An `access` node represents access of an object's attribute. A `decision` node models a point where control flow can take two different ways, depending on a boolean value that was calculated in the previous operation or call. Interprocedural control flow is modeled by `call` nodes, which lead to the `entry` node of the method being called, and `return` nodes leading back to the corresponding `call` node. *Atomic* functions are functions that belong to the interface of the regarded component. These nodes are of special interest for protocol recovery, because they will be transformed to transitions in the protocol automaton (along with `access` nodes). Calls to such functions are of type `atomic_call`. A `final` node indicates the end of the program, i. e., end of the life of the object. Edges represent control flow between these locations, which can either be unconditional (`seq`) or conditional (`true`, `false`). Interprocedural control flow is represented by `call` and `return` edges which connect a `call` node with an `entry` node or a `return` node with a `call` node, respectively.

```
void main() {
  int i = 0;
  S s1 = new S();
  s2 = S.read();
  reverse(s2, s1);
  do {
    s1.pop();
    i = i + 1;
  } while (!s1.empty());
}

reverse(S from, S to) {
  while (!from.empty())
    to.push(from.pop());
}
```



(a) Source code (b) OPG for s1

Figure 1. Sample source code and CFG. Removing the crossed-out nodes results in the OPG for object s1.

Figure 1 shows an example for an OPG along with the corresponding application source code. The graph shows the OPG for stack object `s1` immediately before removing those nodes of the CFG that are not relevant (crossed out). Dark gray nodes represent calls of atomic methods, i. e., methods of the investigated stack class, and diamonds represent decision nodes.

OPGs can be extracted statically or dynamically. While static OPG extraction requires global control and data flow analysis [4], its dynamic counterpart requires program instrumentation for data collection and transformations on the collected data [13, 14]. In this paper, we focus on the use of dynamically extracted OPGs (*DOPGs*) for protocol recovery.

4 OPG-Based Protocol Recovery

Object process graphs contain information about the sequence in which operations of the regarded object are being called or may be called. They also contain information about loops, which are as well important for a protocol. This makes OPGs a potentially good basis for protocol recovery.

The idea for OPG-based protocol recovery is to transform a set of OPGs to a single protocol automaton. The different input OPGs represent different usages of instances of the same component. The foundations of this idea have been described in detail for static OPGs by Heiber [7] and Haak [6]. This approach involves the following steps, which are independent of whether the OPGs are extracted dynamically or statically:

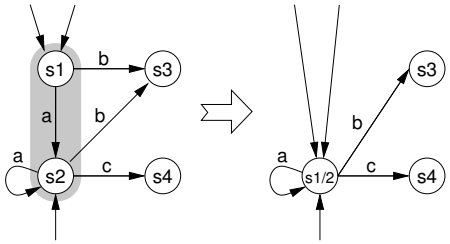


Figure 2. Simplifying transformation example. States s1 and s2 will be merged because all transitions from s1 (a and b) lead to the same states as those from s2.

1. **Eliminate recursion** for each OPG. The resulting graphs contain only `create`, `atomic_call` and `access` nodes and can be regarded as nondeterministic finite-state automata (*NFA*). This step will be explained in more detail in Section 4.1.

2. **Merge** all these NFA, accomplished by merging their start nodes. The result will usually be a highly redundant automaton, because certain steps in using a component are always the same. This redundancy will be reduced in the next steps.

3. **NFA to DFA.** This step uses the subset construction by Hopcroft et al. [8] to get a deterministic finite-state automaton (*DFA*) that is equivalent to the given non-deterministic one, i. e., it accepts exactly the same language. In our case, the algorithm has the effect that commonalities close to the `start` node are unified.

4. **Minimization.** Next, the automaton is minimized with respect to the number of states, using another algorithm by Hopcroft et al. [8]. This algorithm finds all groups of states that can be distinguished by some input string. If two states cannot be distinguished, they are merged to a single state. This step tends to unify parts of the automaton that are close to the accepting nodes, since these naturally form one group.

5. **Additional transformations.** Optionally, further simplifying transformations may be applied, depending on the degree of generalization that is desired. For example, if the exact number of calls to the same method in a sequence is not considered interesting, this could be reduced to a simple loop.

Two different but related transformations are used in our case study as the 5th step:

(a) Merge two states $s1$ and $s2$ if the transitions emerging from $s1$ are a subset of $s2$'s, considering transition event and target state. This means that $s2$ must have a transition to itself, labeled with at least the same events that lead from $s1$ to $s2$. Figure 2 shows an example for such a case. Practically, this means that we do not care about the minimum number of invocations of a method m before looping calls to m. For more complex components, this transformation should be extended to work on sequences of different method invocations as well (e. g., by common substring detection).

(b) The second transformation we apply was introduced by Angluin [2] for inferring *zero-reversible languages*: If a state has two or more incoming transitions with the same label, the source states of these transitions are merged into one state. For the resulting automaton, it is always clear without ambiguity which state was the previous one, given the last input symbol. This is a further generalization of the previous transformation.

Depending on the type of transformation applied in step 5, previous transformations might have to be repeated. Transformations in this step are intended to be generalizing and thus change the language described by the FSA.

Since steps 3 and 4 use well known techniques, we refer to the literature for a detailed explanation. The recursion elimination step will be discussed in detail in the next section.

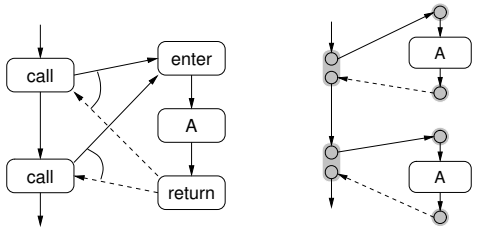
4.1 Recursion Elimination

Regular expressions or, equivalently, finite state automata are the notion of choice for protocol validation. In contrast to that, OPGs are capable of describing a context-free grammar. They contain function calls and therefore allow recursion, which cannot be represented by regular expressions. Hence, to get from an OPG to an FSA, the first step is to eliminate recursion.

An intuitive way for eliminating recursion from an OPG is described by Haak [6]. Call nodes are split into two separate nodes, one connecting the incoming edge with the call edge and the other connecting the return edge with the outgoing edge. (In case of function pointers or dynamic binding, multiple call and return edges occur.) For each invocation of a method, a copy of that method's subgraph is inserted into the graph (*inlining*). The only exception is the recursion case, when the existing copy is used. This way, the mapping between call and return edges is maintained – otherwise, one would not know which return edge to take, resulting in unnecessary overgeneralization.

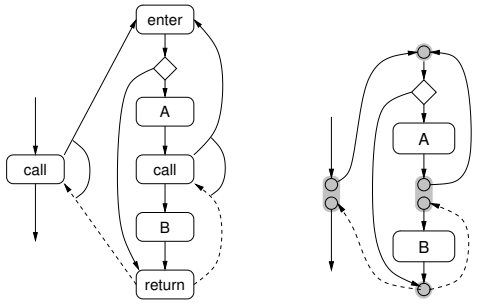
Figure 3 shows how the repeated invocation of the same method leads to copies of the method in the resulting graph. If the invoked method was not copied, this would result in a loop, and then any number of invocations would be allowed. In Figure 4, a recursive function is resolved. Applying this transformation, the correspondence between call and return and the guarantee that there are as many calls as returns are lost. Note that the former return node has two outgoing edges. In the OPG, the choice of which return edge to take was unambiguously given by the corresponding call edge. In the resulting graph, the choice has become non-deterministic. This has the consequence that instead of A^nB^n, the resulting automaton accepts A^mB^n which is more general.

The new intermediate nodes (resulting from `call`, `entry`, and `return` nodes) are then removed from the graph, along with all other nodes that are *not* atomic method calls, attribute accesses or the `start` or `final` node. Finally, the graph is further transformed such



(a) original OPG (b) method copied

Figure 3. OPG to NFA transformation: Multiple invocations lead to copies.



(a) original OPG (b) no recursion

Figure 4. Recursion elimination example. The original OPG creates sequences A^nB^n, the resulting NFA accepts A^mB^n ($m, n \geq 0$).

that edges (instead of nodes) are labeled with the atomic method names or read/write accesses. This can be accomplished by moving each node's label to all incoming edges. The result can be regarded as a nondeterministic finite-state automaton (NFA): Nodes become states, and edges become transitions. End nodes are transformed to accepting states.

4.2 Generalization

A certain amount of generalization is intrinsic for the DOPG based protocol recovery approach. Sources of generalization include:

- Loops for which the loop body always contains only one relevant method call or attribute access. This is not represented in the DOPG. Figure 5 shows a `for` loop that could cause this problem. The `read` method is called once for each object in the array. If `x` contains the regarded object only once, `read` is always called only once, but the loop still remains in the DOPG. In order to better handle this case, the number of invocations can be counted.
- Multiple invocations of the same method, when different operations are relevant for the object. This may happen for example for invocations with different parameters or a different environment. This generalization could be prevented by adding context sensitivity in creating distinct copies for different method invocations, but this could lead to an explosion of the graph's size.
- Recursion elimination, as discussed in Section 4.1.
- Additional simplifying transformations (see Section 4). The first transformation (a) that we use in our experiments has two effects: For multiple method invocation, we lose the information whether a certain minimum number of invocations is required, and by the subset criterion, we allow additional transitions that were not possible before. Transformation (b) loses even more information, allowing further additional transitions.

```
for (int i = 0;
     i < x.length;
     i ++) {
  x[i].read();
}
```

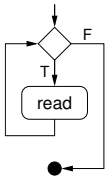


Figure 5. Overgeneralization caused by loops (when `x` contains the object exactly once).

All in all, there are many sources of generalization in our protocol recovery process. The amount can be influenced by choosing more or less additional transformations in step 5, whereas the other transformations (steps 1–4) are mandatory. This makes the approach adjustable to the amount of generalization that is desired, depending on the intended use of the resulting protocol.

5 Comparing Protocol Automata

In order to compare results of the different protocol recovery techniques, it is necessary to compare the languages that are accepted by the recovered protocol automata. Exact comparison of languages is often not possible because they are infinite when loops are present. Therefore, alternative comparison measures that are based on the finite state automaton or regular expression representation have to be applied.

Lo et al. [10] compare two protocol automata A and B with a heuristic approach. The similarity between A and B is measured in terms of their generated sentences. Automaton A is used to generate random sentences, which automaton B tries to accept. The share of accepted words is then regarded as the precision of A with respect to B, or as the recall of B with respect to A. Since this approach is random based, it does not allow an exact comparison. The frequency at which the different words occur is unknown in an FSA and cannot be considered. However, this approach delivers a first approximation of similarity.

We propose a new measure for automaton similarity that is inspired by Levenshtein's measure for string distance [9]. Our measure can be regarded as a kind of edit distance between two automata. The idea is to count the number of edge additions and deletions that are necessary to get from one automaton to the other. To do this counting in a defined (but not necessarily minimal) way, we first need a unified representation of both, which naturally is the union of the two automata. Each of the original automata can then be found in the union automaton by following the transitions in the original and the union automaton in parallel, which corresponds to the product automaton construction [8]. The product automaton describes the intersection of both automata, i. e., their common language. The distance between each of the two automata and their union is calculated based on the information from product automaton calculation. Finally, the two distances are combined, resulting in the difference between the two original automata.

The measure is calculated as follows. Let $T(X)$ denote the number of transitions in automaton X.

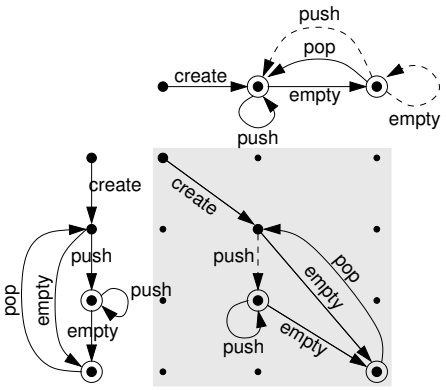


Figure 6. Example for automaton difference calculation. The top automaton is U (dashed transitions are never taken), the left one is A, and the central one is P_A (dashed transition inserted).

1. Input: two deterministic and minimal automata A and B.
2. Calculate the deterministic and minimal union U of A and B (see steps 2–4 of the recovery algorithm).
3. By calculating the product automata P_X of X and U (for $X \in \{A, B\}$), i. e., following the transitions in both automata in parallel, count
   - the number of transitions n_X in U that are never taken. Such unused edges correspond to deletions.
   - the number of transitions t_X that are a self-loop in U, but a non-self-loop transition in P_X or vice versa. When loops are unrolled this way, this corresponds to insertions.
4. The distance between $X \in \{A, B\}$ and U is
$$d_U(X) := \frac{n_X + t_X}{T(P_X) + T(U)} \quad (1)$$
5. The difference between A and B is
$$d(A, B) := \frac{d_U(A) + d_U(B)}{2} \quad (2)$$

This metric is non-negative and symmetric. The result is a value between zero and one, indicating the degree of differences between the two automata: Zero indicates identity, and one indicates that they are completely disjunctive.

Figure 6 shows an example for distance calculation between an automaton A (left) and the union U (top) of A and another automaton B (B is not shown). The product automaton P_A is located in the center. Construction of P_A starts at the start nodes of A and U. From here, `create` is accepted by both automata, so this is part of the product automaton. From the next state, `push` is accepted by both automata. For U, this leads back to the same state (self-loop), which is not the case for A, giving $t_A = 1$. Two edges of U are never used, which means that $n_A = 2$. This results in a difference between A and U of $d_U(A) = \frac{2+1}{6+6} = 0.25$.

6 Case Study

In this case study, we apply the different protocol recovery techniques that were described in Sections 2 and 4 with the same input data and compare the resulting protocol automata. In particular, we want to address the following questions:

- How close do the different approaches come to the real specification?
- How do the DOPG based protocols compare to other approaches' results? Are they more general, equivalent, or more specialized?
- Which of the two introduced simplification steps are useful for DOPG based protocol recovery?
- How does the automaton difference metric compare to Lo's metric?

6.1 Setup and Subject Systems

For this experiment, we choose several Java and C applications and different components they use. We concentrate on standard components: `java.util.ArrayList` and `Stack` for Java, and the file or socket handle for C. Our subject systems are:

- ArgoUML[1], a UML modeling tool (Java)
- ANTLR[2], a parser generator (Java)
- J[3], a powerful text editor (Java)
- sqlite3[4], an SQL database engine (C)
- rhapsody[5] and ircII[6], two console IRC clients (C)

These systems are first instrumented to produce the necessary traces. Then the instrumented systems are executed with typical usage scenarios. From the resulting trace files, we extract one object trace for each

[1] `http://argouml.tigris.org/`
[2] `http://www.antlr.org/`
[3] `http://armedbear-j.sourceforge.net`
[4] `http://www.sqlite.org/`
[5] `http://rhapsody.sourceforge.net/`
[6] `http://www.eterna.com.au/ircii/`

No.	System	Size	Component	AP	Inst.	AMC
1	ArgoUML	264	`ArrayList`	66	346	13,817
2	ANTLR	38	`ArrayList`	2	29	1,291
3	J	158	`ArrayList`	13	37	36,426
4	ArgoUML	264	`Stack`	2	7	352
5	J	158	`Stack`	2	13	943
6	sqlite3	60	file handle	1	2	253
7	rhapsody	18	socket	2	4	8,412
	+ ircII	+49				

Table 1. Characteristics of the components as used in the respective subject system: Size (KLOC), number of allocation points (AP), number of investigated instances, and number of atomic method calls (AMC).

instance of the regarded component. These object traces are then used to construct the corresponding DOPGs [13], which are the basis for our protocol recovery approach. The automaton learning approaches require a simpler trace that contains just the sequence of component method calls. This can be extracted from the object trace. Based on these representations, protocol recovery is performed with different approaches.

Table 1 shows which components of which applications are investigated, how many instances occur in each case, and how often the component's methods are called in total. It also shows the number of distinct allocation points. An allocation point is the location in the source code where an instance of the regarded class is created, i. e., the location of the `new` statement or `malloc` call. For the socket component, we combine rhapsody's and ircII's traces to get more information about the general usage of a socket.

The following protocol recovery approaches and tools are applied (see Section 2 for details):

- *Tree:* Minimized prefix tree acceptor.
- *Successor:* Successor method plus minimization.
- *k-tails:* Reiss' tool from his Bloom package.
- *sk-strings:* Raman's tool (also used by Ammons).
- *DOPG:* DOPG based protocol recovery steps 1-4.
- *DOPG-A:* DOPG plus transformation 5a.
- *DOPG-B:* DOPG plus transformation 5b.

This results in 7 automata that are to be compared. Unfortunately, there is no official specification for the regarded components available. Therefore, we created the real specification of the protocol as we would expect it. This real specification is represented as an 8th automaton and used as a reference. For all these automata, the number of states and transitions is measured to get an impression of their sizes.

Each of these automata is then compared to the specification. For this comparison, the specification is reduced to those methods that are really used in the respective traces. It is reasonable to do so because methods that are not called are not visible for any dynamic analysis. The comparison is performed by applying both introduced similarity measures: random-based precision/recall approximation, and automaton difference. Also, the correlation between these two measures is calculated.

6.2 Results

Tables 2 and 3 show the results of our study. This section discusses them in detail.

Automaton sizes. In Table 2, the sizes of the different resulting protocol automata in terms of number of states and transitions are displayed. The sizes of the specification and prefix tree automata are given for comparison. The table shows that all approaches reach a good compression ratio compared to the PTA. However, sizes still differ in a wide range. In five out of seven cases, the DOPG based automaton is the largest. With additional simplifications, specially with transformation 5(b) (see Section 4), the DOPG based automata can be reduced to a size comparable to the other approaches. Only for case 3, the DOPG-B automaton is still very large. In summary, DOPG based protocols remain more specialized than the others.

Automaton/language similarity. Table 3 shows the results for language and automaton comparison of each recovered protocol automaton with the specification. For each case, the first and second columns contain the precision and recall as calculated with Lo's method (100,000 samples). The third column displays the automaton's difference to the specification automaton. The last line shows the correlation between precision and distance (prec.) and recall and distance (rec.). Let us now discuss the different cases in detail.

- *No. 1:* k-tails is closest to the specification and also delivers a 100% recall automaton. The successor method results in a higher precision, but its recall is not higher than the PTA's. Both are better in quality than the DOPG based protocols, but those are still better than PFSA.
- *No. 2:* DOPG has the highest precision, while DOPG-B has the highest recall and the lowest difference to the specification.
- *No. 3:* Precision and recall for DOPG-B are both quite good. k-tails has a higher recall at the price

No.	Spec.	Tree	Succ.	k-tails	PFSA	DOPG	DOPG-A	DOPG-B
1	4:31	6k:6k*	19:64	3:19	42:126	233:687	232:684	20:50
2	4:17	259:273	12:18	8:12	20:40	43:61	36:46	20:28
3	3:10	34k:34k*	6:14	2:6	13:42	475:760	469:754	307:465
4	4:8	334:334	7:9	4:6	8:10	27:38	26:37	6:8
5	4:7	871:872	5:7	3:5	21:25	4:7	3:5	3:5
6	4:8	241:241	8:14	4:8	8:14	24:54	22:49	3:8
7	4:12	8k:8k*	13:33	9:19	49:73	34:80	23:50	14:30

Table 2. Protocol automaton sizes (states:transitions) for the different approaches. Specification and prefix tree sizes are given for comparison. The automata marked with a * have not been minimized.

No.	1			2			3			4			5			6			7		
Tree	100	32	n/a	100	15	48	100	28	n/a	100	43	49	100	21	58	100	0	32	100	32	n/a
Succ.	87	33	50	88	15	52	83	34	32	100	44	23	75	23	18	100	1	17	89	33	50
k-tails	42	100	5	75	17	46	75	100	5	50	48	31	67	100	32	100	100	0	16	100	10
PFSA	3	29	51	93	16	52	4	25	42	100	44	24	100	21	44	100	1	17	3	29	51
DOPG	37	33	57	95	15	53	79	36	62	100	43	34	89	57	24	75	1	33	63	33	57
DOPG-A	37	32	57	80	20	43	79	36	62	100	43	34	67	100	4	75	1	32	63	33	57
DOPG-B	17	38	60	83	22	32	85	65	58	40	47	30	67	100	4	58	100	6	29	35	60
prec./rec.	-15	-96		-26	-95		-18	-77		-12	-82		47	-72		-18	-89		-47	-98	

Table 3. Automaton/language similarities when compared to the real protocol automaton: precision, recall and difference. Some values could not be computed due to memory limitations (n/a). The last line shows the correlation between precision/recall and automaton difference (all values in percent).

of a lower precision. However, the DOPG based automaton is about factor 100 larger than the k-tails result.

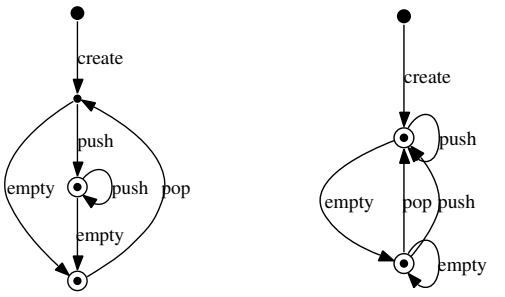
- *No. 5:* PFSA reaches the highest precision but the lowest recall (same as PTA). k-tails and PFSA-A/B have the same precision and recall; they result in different automata of the same size. This is indicated by the measured difference to the specification.
- *No. 6:* k-tails results exactly in the specification. Most other approaches have a very high precision and very low recall, except DOPG-B which has a lower precision but 100% recall.
- *No. 7:* The successor method has the highest precision, while k-tails delivers the highest recall. DOPG is still better than PFSA in this case.

All in all, the results are diverse: In some cases, DOPG provides the best quality protocol automata, but in other cases, other approaches have better results. k-tails is often close to the specification, but also often generalizes too much. An interesting observation is that in many cases, the approaches reduce precision but do not increase recall. This means that generalization in these cases affects only automaton size, but not the accepted language.

There is no correlation between automaton difference and precision, but between measured automaton difference and recall. The correlation is between -0.72 and -0.98. This means that the more the difference to the specification increases, the fewer allowable sequences of operations are accepted by the automaton.

DOPG variants. Regarding the differences between the different DOPG variants, there are a few more things to observe. With an increasing degree of generalization, recall increases and precision drops. However, this is not true in all cases. The difference between DOPG and DOPG-A is small in most cases, and the languages accepted by the respective automata do not differ much. Only the transformation to DOPG-B reduces the automaton's size significantly in all cases, also affecting precision and recall (increased recall, reduced precision). In summary, DOPG-A does not have a great effect and can therefore be omitted in practice.

Example. Figure 7 shows the protocol automata for scenario 5 that result from the successor and DOPG algorithms. The DOPG automaton allows `push` and `empty` after an `empty`, which are both not allowed in the successor one. The language of the successor automaton is completely accepted by the DOPG automaton. Both automata show that there can never be a `pop` without a preceeding `empty`.



(a) Successor (b) DOPG

Figure 7. Two protocol automata for system/component no. 5 (constructor calls omitted). The successor automaton accepts a subset of the DOPG automaton's language.

7 Conclusion

We have described a novel dynamic protocol recovery approach. Using dynamic object process graphs as the basis, a protocol automaton can be derived by a sequence of transformations. Compared to the traditional prefix tree acceptor, DOPGs have the advantage of containing information about loops and context.

Our case study showed that the approach is applicable in practice. It also showed that the resulting automata are usually more detailed than the other approaches' results. They are closer to the application, which may be helpful for program understanding. On the other hand, this is not necessarily an advantage when the goal is recovery of the specification. However, when incorporating simplifying transformations, the resulting protocols were often better than the other protocol recovery approaches' results.

We also introduced a new metric for comparing automata. In our case study, this metric indicated differences between automata where other metrics did not find any difference. Also, we detected a high correlation between this metric and the measured recall as defined by Lo and Khoo.

All in all, DOPG based protocol recovery is a promising approach. It provides a concrete detailed protocol, which may be adjusted to the desired level of generalization by choosing appropriate simplifying transformations.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th Symp. on Principles of Prog. Languages (POPL)*, pages 4–16, 2002.

[2] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.

[3] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE Trans. on Comp.*, 21:591–597, 1972.

[4] T. Eisenbarth, R. Koschke, and G. Vogel. Static object trace extraction for programs with pointers. *Journal of Systems and Software*, 77(3):263–284, Sep 2005.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th Symp. on Operating System Design and Impl.*, pages 1–16, 2000.

[6] D. Haak. Werkzeuggestützte Herleitung von Protokollen. Diploma thesis, Univ. Stuttgart, CS, 2004.

[7] T. Heiber. Semi-automatic protocol recovery. Diploma thesis, Univ. Stuttgart, Computer Science, 2000.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, second edition, 2001.

[9] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, Feb. 1966.

[10] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Proc. of 13th WCRE*, pages 51–60, 2006.

[11] D. Lo and S.-C. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proc. of 14th FSE*, pages 265–275, 2006.

[12] K. M. Olender and L. J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, Jan. 1992.

[13] J. Quante and R. Koschke. Dynamic object process graphs. In *Proc. of 10th CSMR*, pages 81–90, 2006.

[14] J. Quante and R. Koschke. Dynamic object process graphs. *Journal of Systems and Software*, 2007. Accepted for publication.

[15] A. Raman and J. Patrick. The sk-strings method for inferring PFSA. In *Proc. Workshop Automata Induction, Gramm. Inference and Lang. Acquisition*, 1997.

[16] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of 23rd ICSE*, pages 221–230, 2001.

[17] M. Richetin and F. Vernadet. Regular inference for syntactic pattern recognition: A case study. In *Proc. 7th Intl. Conf. on Pattern Rec.*, pages 1370–1372, 1984.

[18] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *Proc. of ICSM*, pages 155–164, 2005.

[19] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proc. of Symposium on Security and Privacy*, pages 144–155, 2001.

[20] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. of Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[21] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems*, pages 39–46, 2004.