Abstract—A protocol defines the sequencing constraints for the operations that can be applied to an object. Quante introduced a protocol recovery technique that is able to extract protocols from existing software by means of dynamic analysis. This approach represents the behavior as object process graphs (OPG). OPGs are a projection of the control flow graph reduced to the operations relevant to an individual object. The protocol is inferred from a set of OPGs. The extraction was designed to handle sequential programs only.

As multi-core architectures and, hence, multi-threading becomes more and more common in nowadays programming, it is necessary to extend reverse engineering techniques for multi-threaded programs.

In this paper, we extend Quante’s approach to protocol reconstruction for programs with multiple threads. We are formalizing this process using concepts from automata theory, namely, product and shuffle automata. We present a naïve approach to combine these concepts and a combined approach. Our evaluation for realistic Java programs demonstrates the scalability of the combined approach and the combinatorial explosion of the naïve approach.

Keywords—Protocol recovery, dynamic analysis, object process graphs, automata theory, product automata, shuffle automata.

I. INTRODUCTION

A protocol describes the allowable sequences of operations an interface offers. Protocols need to be known for validation purposes [1] as well as program comprehension [2]. Far too often, the protocol is not sufficiently specified and, hence, it needs to be reconstructed from the program. Protocol recovery is the process of reconstructing the protocol of a component’s interface.

Many researchers have tackled protocol recovery using static or dynamic analysis. Among these is the technique by Quante and Koschke [3]–[7]. This technique executes the program observing operations at the level of individual objects. A dynamic object process graph is built from the object-based trace. An object process graph is a projection of the original control flow graph where all operations irrelevant to the behavior of an object are omitted.

The technique has several advantages over previous techniques such as automated machine learning algorithms analyzing traces. Previous approaches are solely based on the sequence of method invocations of the investigated component. They cannot distinguish between loops and repeated invocations 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.

A dynamic object process graph (DOPG) contains more information than traditional outputs of automaton learning approaches: 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 their original work, Quante and Koschke have excluded the problem of collaborating threads that jointly work on a common set of objects. In the presence of threads that apply operations onto the same object in an interleaved manner, the DOPG of their approach will be wrong. Because there is a current trend toward multi-threaded programs, this issue should be fixed.

Contributions. This paper gives a correct modelling of DOPGs when multiple threads collaborate for the behavior of the same object. We first describe a naïve solution using several advanced concepts of automata theory. Measurements show that the naïve solution does not scale to realistic programs. Therefore, we provide an improved solution, for which we demonstrate scalability.

Overview. The remainder of this paper is organized as follows. Section II describes related research and Section III explains the original approach and its problem. Section IV describes a solution, evaluated in Section V. Section VI finally, concludes.

II. RELATED RESEARCH

A protocol in the sense of this paper describes the sequencing constraints that are imposed on a component’s operations: It tells us in which order these operations may be applied. The sequencing protocol is part of the interface...
of a component. Client code using the interface must obey to the protocol.

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. The states of such protocol automata represent program states, and transitions correspond to operations on a component. As a side note, modelling parallel programs with sequential FSA implies that all transitions are mutually exclusive because no two events can co-occur at the very same moment in FSA.

Protocol recovery is the attempt to infer the protocol from a given program. It can be approached by looking at the interface’s implementation, for instance, at the encoded preconditions and paths leading to code that throws exceptions (white-box recovery). Alternatively, it can be derived from actual usages of the interface (black-box recovery). Under the assumption, the actual usages are all correct, a correct protocol may be inferred. In cases where the actual usages considered are not representative for all possible behaviors, the derived protocol may be incomplete. Despite of potential incorrectness and incompleteness, the recovered protocol may be a starting point that can be validated and completed by a human. It can also be used to generate test cases in order to further complete the protocol.

Protocol recovery techniques can be further distinguished by the type of event they consider. Early techniques consider procedural calls. While that may be sufficient for procedural languages, the behavior of object-oriented systems depends generally on the state of the receiver object of a method call. Object-based protocol recovery distinguishes events not only by the method being called but also by the receiver object.

In most object-based approaches, states are constituted through sequences of method calls. A different idea is to introduce states for different object states (in terms of their attributes) or abstractions thereof.

Protocol recovery can be tackled by static or dynamic analyses, or by a combination of both (called hybrid). The static approach has the advantage of finding all potentially possible calling sequences, that is, yields sound results if based on conservative assumptions. Due to inherently undecidable problems about infeasible paths, however, static analysis is generally imprecise. A dynamic approach can build only upon those sequences that really occurred in a given set of program runs, thus, yields precise results. On the other hand, because it is generally impossible to consider all possible inputs, dynamic analysis offers incomplete results in general, that is, is not sound. 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.

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 that produces an automaton accepting the given traces – and possibly more. Constructing an automaton that accepts exactly the given set of traces (prefix tree acceptor, PTA) is simple but not useful. This automaton represents only 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.

In this spectrum, the technique by Quante can be considered a hybrid (static and dynamic) object-based protocol recovery technique. It offers several advantages over purely dynamic approaches. Since the latter are solely based on the sequence of method invocations observed at runtime, they cannot distinguish between loops and repeated invocation of a method. Likewise in the presence of multiple threads, an observed program trace may represent only one particular interleaving of the execution of the threads, which further increases the chances of incompleteness. Quante’s techniques leverages the static control flow graph and does not have these problems. 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. Quante’s technique 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. Thus, there is a very close connection between the behavior
represented by an DOPG and the original program, which further helps in understanding and validation. Quante et al. compared this technique quantitatively to other approaches [5] and demonstrated the usefulness of its results in program comprehension tasks [2]. The focus of this paper is to extend this technique to multithreaded programs.

III. ORIGINAL APPROACH TO PROTOCOL RECOVERY

This section introduces the terminology and concepts relevant to protocol recovery based on DOPGs. It describes the original technique for extracting DOPGs and recovering protocols from DOPGs. Finally it gives an example for a multi-threaded program where the original technique produces an incorrect DOPG.

A. Concepts

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 typed graph

\[ \text{OPG} := (N, E, T_N, T_E) \]

where each node \( n \in N \) and each edge \( e \in E \) can be of one of the following types:

- \( T_N : N \rightarrow \{ \text{start}, \text{create}, \text{access}, \text{decision}, \text{call}, \text{entry}, \text{return}, \text{atomic\_call} \} \)
- \( T_E : E \rightarrow \{ \text{call}, \text{return}, \text{seq}, \text{true}, \text{false} \} \)

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 paths, 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. An atomic_call is a call to operations that belong to the interface of the regarded component. These nodes of type atomic_call and access are referred to as atomic operations. A final node indicates the end of the program, that is, end of the life of the object. Edges represent intraprocedural control flow between these locations, which can either be unconditional (seq) or conditional (true, false).

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 [18], its dynamic counterpart requires program instrumentation for data collection and transformations on the collected data [4], [7]. In this paper, we focus on the use of dynamically extracted OPGs (DOPGs) for protocol recovery.

B. DOPG Extraction and Protocol Recovery

The goal is to recover the protocol of a component, for instance, a class in Java. The idea is to infer the protocol from actual usages of that class from correct programs. A class is used at runtime by creating an object of that class and using this object in various ways. The first step in the approach is, hence, to extract DOPGs for all objects of that class by using different test cases so that different legal usage
scenarios are covered. Then a protocol graph is inferred from the DOPGs for each object and test case. Finally, the individual protocol graphs are combined into one to obtain the final protocol.

The dataflow in the creation of the DOPG is shown in Figure 2. The observed events (operations involving objects and evaluations of predicates) of the running program are externalized as a trace. The trace is then filtered for the relevant objects and a raw DOPG is created and then transformed by simplification rules into the final DOPG. Part of this transformation is a projection. The projection removes all nodes and edges of the raw graph not reachable from the allocation point of the object.

Each DOPG is then transformed into a finite state automaton representing the protocol as sketched in Figure 2a. The DOPG has the semantics of a stack machine. Finite state automata are more amenable to automated protocol conformance validation due to the decidability properties of finite automata. For this reason, each DOPG is transformed into a protocol graph. First, if there are multiple invocations of the same method, the DOPG for the called method contains paths for each calling context. For better precision, these different paths are disentangled by inlining the called method – if non-recursive – at the call site. After that, if there are multiple invocations of the same method, the DOPG for the called method is transformed by simplification rules into the final DOPG. The result is a protocol graph for each DOPG. These protocol graphs are finally merged into a single automaton, which is then made deterministic and minimized through a series of automaton transformations.

More details can be found in the original paper [6].

C. Problem for Multi-Threading

Although the original technique has a thread-safe implementation and provides correct DOPGs for programs with multiple threads whose sets of objects they apply operations on are disjoint, the technique was not designed to work for multi-threaded programs in general. If multiple threads apply operations on the same object in an interleaved manner, the resulting DOPG will be wrong. This problem can be exemplified by the example Java program in Figure 3 whose DOPG is shown in Figure 4b. The original technique creates the protocol shown in Figure 4b, while the correct protocol should be in fact the one in Figure 4c. The wrongly extracted protocol does not contain the call to connect(), which is necessary before any read() can be called. However, the implementation does obey to the correct protocol.

The wrongly extracted protocol contains only those nodes reachable from the create node of the thread who created the object. The projection as part of the DOPG creation is responsible for this problem. Omitting the projection is not a solution either as can be demonstrated with Figure 5 where a DOPG and its extracted protocol can be seen. One needs to take into account the global sequence of operations among all threads and not just the individual sequence of single threads. The next section will explain how.

IV. IMPROVED EXTRACTION

This section describes an extension to the original protocol recovery that works also for multiple threads sharing objects. The dataflow of this extraction technique is shown in Figure 2b. As Figure 2b shows, the steps to extract a raw graph are the same as in the original approach except that a global trace is created additionally. The projection, however, that was the root cause of the confusion will be omitted in the graph transformations applied to obtain the DOPG from the raw graph. The new steps will be discussed in more detail in the following.

A. Inter-Thread Sequence Graph

The technique is based on an inter-thread sequence graph (ITSG), which describes the interleaved operations of threads globally.

Given an OPG = (N, E), where N are the nodes and E are the edges of the OPG. An ITSG for the OPG is a graph (N_{ITS}, E_{ITS}), where N_{ITS} ⊆ N and E_{ITS} ⊆ N_{ITS} × N_{ITS}. The edges of an ITSG model the path among the nodes of the OPG in the order of their actual execution at runtime across all threads. They are derived from a global trace as follows: If there are two immediately subsequent relevant events e_1 and e_2 in the trace, an ITSG edge (n_1, n_2) will be added to E_{ITS} where nodes n_1 and n_2 are the corresponding OPG nodes of e_1 and e_2, respectively.

N_{ITS} contains all nodes of N except for call nodes not representing calls to atomic operations since method calls are modelled differently in the ITSG: a call node in the DOPG leads from the call site to the called method and via its return node back to the call node. If the same modelling was used in the ITSG, the call would appear to be a loop because the ITSG does not differentiate between edge types. Instead the return edge in the DOPG will appear in the ITSG as an edge to the immediate successor of the call node in the actual control flow.

B. Integration of Global Trace

The steps of the protocol extraction are very similar to the original approach. Again, the inlining is done as far as recursion allows. At the end, the recursion is eliminated. In between, however, we add a new step that leverages the global trace to obtain a correct protocol graph.

As opposed to the original approach, a global trace is kept of all operations consisting of tuples (threadID, nodeID, objAddr) ∈ N × String × String, where threadID is the unique identifier of a thread, nodeID is an
identifier representing the operation in the OPG, and \texttt{objAddr} is the identifier of the object involved in the operation.

The additional step scans the global trace until it finds an entry that represents the \textit{create} operation for the given object \(o\) whose protocol is to be extracted. The initial node of the ITSG is created for that event. After that it continues scanning for entries \((t, n, o)\) and adds an edge \((l, n')\) to the ITSG whenever it finds an operation relevant to the object of interest \(o\), where \(t\) denotes the thread that executed the operation as documented in the global trace, \(l\) is the last node added to the ITSG, and \(n'\) is the node in the ITSG that corresponds to the DOPG whose node ID is \(n\) taken from the global trace.

The ITSG for our running example is shown in Figure 7.

C. Protocol Graph Extraction

From the ITSG we can extract the protocol by applying the same transformations originally applied to the DOPG. The ITSG is reduced to the atomic operations and then transformed into a deterministic finite state automaton. This step is formalized here.

An ITSG path \(p \in N_{ITS}^*\) is a path in the ITSG \((N_{ITS}, E_{ITS})\), that is, a finite sequence of nodes \(p = n_1 \cdots n_m\) where

- \((n_i, n_{i+1}) \in E_{ITS}\) for \(i = 1, \ldots, m - 1\)
- \(T_N(n_1) = \text{start}\)
- \(T_N(n_m) = \text{final}\)

The set of all ITSG paths is \(P_{ITS} \subseteq N_{ITS}^*\). By applying a projection of ITSG paths onto atomic operations the protocol can be extracted for \(P_{ITS}\).

A \textit{projection} eliminates all symbols in the alphabet \(\Sigma\) that are not in a given alphabet \(\overline{\Sigma}\). In general, the projection is
In our case $\Sigma = N$ and $\Sigma$ is the set of the atomic operations $N_a := \{ n \in N \mid T_N(n) \in \{ \text{atomic\_call, access} \} \}$. The atomic-call sequence $\pi_{N_a}(p)$ of an ITSG path $p$ can be obtained by the projection $\pi_{N_a} : N_{\text{ITSG}}^* \rightarrow N_a^*$. If one wants to obtain an intra-thread protocol, that is, the atomic operations for a thread $t$, one uses the projection $\pi_{N_t \cap N_a}(p)$ where $N_t$ denotes the set of nodes in the ITSG associated with thread $t$.

The protocol $L_P \subseteq N_a^*$, finally, is obtained from the set of all ITSG paths of $P_{\text{ITSG}}$ as follows: $L_P := \{ l(\pi_{N_a}(p)) \mid p \in P_{\text{ITSG}} \}$ where $l$ is a labeling function $l : N_a^* \rightarrow \text{String}^*$ that returns the sequences of names of the sequence of operations given as argument. The protocol is represented as a finite state automaton accepting $L_P$.

D. Infeasible Paths

The protocol obtained as described in the previous section may contain infeasible paths when projected for a given thread; that is $\exists t \in N, p \in P_{\text{ITSG}} : l(\pi_{N_t \cap N_a}(p)) \notin L_t$ where $L_t$ is the intra-thread protocol of thread $t$.

The problem can be illustrated with Figure 8 showing a DOPG for two threads $A$ and $B$. This DOPG is similar to the one in Figure 7 somewhat simplified by removing superflu-
ous nodes. Furthermore this time both threads execute three atomic operations each. Let \( N_\alpha = \{ a_1, a_2, a_3, b_1, b_2, b_3 \} \) be the atomic operations labeled with the identity function, that is, \( l = \text{id} : N_\alpha \to N_\alpha, n \mapsto n \). The protocol \( L_A \subseteq N_\alpha^* \) within thread \( A \) can be described by the regular expression \( L_A = L(\{a_1a_2a_3\}^*) \).

Let us assume that the program execution yields a global trace \( p = n_1s_1n_Ea_1a_2b_1a_3b_2b_3a_1b_2a_2a_3b_3b_F \in N_\text{TS} \) (omitting the condition nodes for simplicity). The operations within threads correspond to the order in the DOPG (e.g., \( a_1 \to a_2 \to a_3 \to \ldots \)). A projection of the global trace for thread \( A \), thus, yields a valid path in the DOPG \( \pi_{N_A}(p) = n_1s_1n_Ea_1a_2a_3a_1a_2a_3 \).

If we reconstruct an ITSG from \( p \), we obtain the dashed ITSG edges in Figure 8 including the sequence \((n_E, a_1), (b_2, a_3), (a_3, b_3), (a_1, b_2), (b_3, b_F) \in E_{\text{TS}} \). Consequently, there is an ITSG path \( p' = n_1s_1n_Ea_1a_2a_3b_3b_F \in P_{\text{TS}} \).

E. Avoiding Infeasible Paths

We need to avoid infeasible paths in the protocol by taking the intra-thread protocols \( L_1, \ldots, L_m \) of the individual threads into account in addition to the ITSG. One could restrict the set of ITSG paths so that none of the intra-thread protocols is violated: \( P'_{\text{TS}} := \{ p \in P_{\text{TS}} \mid \forall i \in \{1, \ldots, m\} : l(\pi_{N_i \cap N_A}(p)) \in L_i \} \). Then we construct an automaton only for \( P'_{\text{TS}} \).

Computing this set is difficult, however. A better approach is to filter infeasible paths from the automaton constructed for \( P_{\text{TS}} \). \( P_{\text{TS}} \) as well as the feasible intra-thread paths can be expressed as finite state automata, and we can, hence, use transformations from automata theory. One possibility is to use so-called shuffle automata to model the feasible intra-thread interleavings. Next, we introduce a shuffle for words, which is then extended to languages.

Given two words \( w_1, w_2 \in \Sigma^* \) for an alphabet \( \Sigma \), their shuffle \( w_1 \Delta w_2 \) is the set of all words that can be obtained by interleaving symbols of \( w_1 \) and \( w_2 \) (\( a,b \in \Sigma \)):

\[
\begin{align*}
  w_1 \Delta \epsilon &= \{ w_1 \} \\
  \epsilon \Delta w_2 &= \{ w_2 \} \\
  w_1 a \Delta w_2 b &= (w_1 a \Delta w_2) b \cup (w_1 \Delta w_2 b) a
\end{align*}
\]

Accordingly, the shuffle of two languages \( L_1, L_2 \subseteq \Sigma^* \) is the union of all shuffles of all word pairs:

\[
L_1 \Delta L_2 := \bigcup_{w_1 \in L_1, w_2 \in L_2} w_1 \Delta w_2
\]

It can be shown that the shuffle operation is associative and commutative and \( L_1 \Delta L_2 \) is a regular language if \( L_1 \) and \( L_2 \) are regular languages over a common alphabet \( \Sigma \).

Multiple intra-thread protocols \( L_1, \ldots, L_n \subseteq \Sigma^* \) can be combined by their shuffle \( L_\Delta := L_1 \Delta \cdots \Delta L_n \). \( L_\Delta \) describes all possible operation interleavings of all threads – whether they occur in practice or not. The idea now is to intersect the protocol \( L_{\text{TS}} \) obtained from the ITSG for a given object and the shuffle of all intra-thread protocols \( L_1, \ldots, L_n \). Thus, we compute the intersection of \( L_\Delta \) and \( L_{\text{TS}} \): \( L_P = (L_1 \Delta \cdots \Delta L_n) \cap L_{\text{TS}} \). The elements in the intersection are those paths obtained from the ITSG that are also feasible from the thread-interleaving point of view.

F. The Na"ive Computation

A straightforward solution to computing \( L_P \) is to generate the shuffle and then to intersect it with the automaton for \( L_{\text{TS}} \).

Shuffle Automaton: The shuffle automaton is determined as follows. Let \( A_1 = (S_1, \Sigma, \delta_1, s_{01}, F_1) \) and \( A_2 = (S_2, \Sigma, \delta_2, s_{02}, F_2) \) be two finite state automata, where \( S_1, S_2 \) are their set of states, \( \Sigma \) is the alphabet, \( \delta_1, \delta_2 \) are their transition functions, \( s_{01} \in S_1, s_{02} \in S_2 \) are their start states, and \( F_1, F_2 \) their set of final states, respectively. The shuffle automaton is a nondeterministic finite automaton \( A_\Delta \) as follows:

\[
(S_\Delta, \Sigma, \delta_\Delta, s_{0\Delta}, F_\Delta) := (S_1 \times S_2, \Sigma, \delta_\Delta, (s_{01}, s_{02}), F_1 \times F_2)
\]

where \( \delta_\Delta \) is a function from \( S_\Delta \times \Sigma \) to \( \mathcal{P}(S_{\Delta}) \) defined as follows:

\[
\delta_\Delta((s_1, s_2), a) := \{ (s'_1, s_2) \mid s'_1 \in \delta_1(s_1, a) \} \cup \{ (s_1, s'_2) \mid s'_2 \in \delta_2(s_2, a) \}
\]

The two intra-thread automata derived from Figure 8 are shown in Figure 9a. Their shuffle automaton is shown in Figure 9b.

Intersection of Two Automata: The intersection of two automata is their Cartesian product, also known as product automaton. The product automaton can be thought of as an automaton that runs two automata simultaneously and accepts a word if both single automata are in their final state at the end of the word. More precisely, let \( A_1 \) and \( A_2 \) be two finite state automata as above. The product automaton \( A_\cap \) of \( A_1 \) and \( A_2 \), written \( A_1 \cap A_2 \), is the tuple \( (S_\cap, \Sigma, \delta_\cap, s_{0\cap}, F_\cap) := (S_1 \times S_2, \Sigma, \delta_1 \times \delta_2, (s_{01}, s_{02}), F_1 \times F_2) \) where \( \delta_\cap \) is given by

\[
\delta_\cap((s_1, s_2), a) := \delta_1(s_1, a) \times \delta_2(s_2, a)
\]

and \( a \in \Sigma \).
Figure 9: Automata combined into protocol; the start states use the UML notation for start states; the final states are highlighted by thicker lines

The difference between the shuffle and the product automaton is the transition function, which simulates a concurrent rather than parallel execution.

Using the concepts of shuffle and product automaton, we can compute \( L_P = (L_1 \Delta \cdot \cdots \Delta L_n) \cap L_{ITS} \) using their respective automata as follows: \( (A(L_1) \Delta \cdots \Delta A(L_n)) \cap A(L_{ITS}) \) where \( A(L) \) denotes the automaton for a language \( L \).

The protocol automaton for the shuffle automaton in Figure 9b and the automaton for the ITSG – shown in Figure 9c – for our running example in Figure 8 is shown in Figure 9d. We notice that some of the paths in Figure 9d are dead ends and could be removed by further minimization. The product automaton describes the regular language \( \{ (s_1, \ldots, s_n, s_{ITS}) | s_i \in \delta_i(s_i, a), s_{ITS} \in \delta_{ITS}(s_{ITS}, a) \} \)

- the initial state \( s_0P = (s_01, \ldots, s_{0n}, s_{0ITS}) \)
- the final states \( F_P = \times \cdots \times F_n \times F_{ITS} \)

This automaton can be obtained by a simple term rewriting of the shuffle and product automata. Let \( A_0 \) be the automaton as defined in this section using the combined approach. Let \( A_{\cap} = (A(L_1) \Delta \cdots \Delta A(L_n)) \cap A(L_{ITS}) \).

To simplify the term rewriting, we consider only two intra-thread protocols, but the following equation can be easily extended to more than two due to the closeness of shuffle automata. That is, we have \( A_{\cap} = (A_1 \Delta A_2) \cap A_{ITS} \) with \( A_1 = (S_1, \Sigma, \delta, s_01, F_1) \) and \( A_2 = (S_2, \Sigma, \delta, s_02, F_2) \) (let \( s_{\Delta} = (s_1, s_2) \))

\[
\begin{align*}
\delta_{\cap}(s_{\Delta}, s_{ITS}), a) &= \delta_{\Delta}(s_{\Delta}, a) \times \delta_{ITS}(s_{ITS}, a) \\
&= \delta_{\Delta}(s_1, s_2, a) \times \delta_{ITS}(s_{ITS}, a) \\
&= \left( \{ (s', s_2) \mid s' \in \delta_1(s_1, a) \} \cup \{ (s_1, s') \mid s' \in \delta_2(s_2, a) \} \right) \\
&\quad \times \delta_{ITS}(s_{ITS}, a) \\
&= \left( \{ (s'_1, s_2, s_{ITS}) \mid s'_1 \in \delta_1(s_1, a), s_{ITS} \in \delta_{ITS}(s_{ITS}, a) \} \right) \\
&\quad \cup \{ (s_1, s'_2, s_{ITS}) \mid s'_2 \in \delta_2(s_2, a), s_{ITS} \in \delta_{ITS}(s_{ITS}, a) \} \\
&= \delta_{\cap}(s_1, s_2, s_{ITS}, a) \\
\end{align*}
\]

That is, the two automata, \( A_p \) and \( A_{\cap} \) are isomorphic and consequently their language is identical. The automaton is a simulation of \( A_{ITS} \) where all infeasible paths are cut. The rewritten term avoids the explicit computation of the expensive shuffle automaton.

V. EVALUATION

This section compares the new approach to the original one. It reports on the extra costs that incur for the global trace. Furthermore it compares the naive and the combined calculation of the protocol graph.
### Table I: Subject systems; KLOC excludes blank lines and comments

<table>
<thead>
<tr>
<th>System</th>
<th>Domain</th>
<th>Version</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>UML tool</td>
<td>0.28.beta5</td>
<td>173</td>
</tr>
<tr>
<td>FindBugs</td>
<td>Bug Finder</td>
<td>1.3.6</td>
<td>100</td>
</tr>
<tr>
<td>J</td>
<td>Editor</td>
<td>0.21.0</td>
<td>140</td>
</tr>
</tbody>
</table>

### Table II: Runtime resources for FindBugs

<table>
<thead>
<tr>
<th>Test case</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td>Original code</td>
<td>28.68</td>
<td>306.22</td>
<td>21.98</td>
</tr>
<tr>
<td>DOPG (online)</td>
<td>65.75</td>
<td>708.28</td>
<td>417.72</td>
</tr>
<tr>
<td>G. trace</td>
<td>417.72</td>
<td>417.72</td>
<td>140.0</td>
</tr>
<tr>
<td>sum = Δt; DOPG</td>
<td>384.21</td>
<td>1.64</td>
<td>1.64</td>
</tr>
<tr>
<td>G. trace: MB</td>
<td>306.22</td>
<td>306.22</td>
<td>140.0</td>
</tr>
</tbody>
</table>

### A. Overhead for Global Trace

To compare the overhead of extracting the DOPG and global trace, we measure three runtime aspects: time for (1) the original program $t_{\text{original}}$, (2) the online extraction of the DOPG $t_{\text{DOPG}}$, and (3) capturing and storing the global trace $t_{\text{G. trace}}$. Aspect (2) contains the time needed for event capturing, filtering, graph construction, and graph transformation in Figure 2a to generate the DOPG. All steps – but event capturing – can be performed both online – while the program is running – and offline – post-mortem to the program execution. According to the evaluation by Quante et al., both alternatives are comparable in terms of runtime overhead, but the online approach requires considerably less disk space. The DOPG is a relatively small graph bounded by the size of the control flow graph of the program, whereas the length of the trace depends upon the execution time and can be virtually infinite. Creating the DOPG online avoids capturing and storing a global trace. Our approach needs the complete global trace to derive the protocol graph from the DOPG.

For the runtime measurements, we chose a program that does not require user input because latency of user input is non-deterministic and it is difficult to make sure that the user behaves exactly the same for different program runs. For this reason, we chose the program FindBugs that offers a batch mode. In the batch mode, only one thread is used so that we can fairly compare the two approaches.

We applied FindBugs to search for bugs in three different smaller Java projects. All other parameters were kept the same across the Java projects. This gives us three different data points for the runtime overhead. Table I reports the results. The dynamic analysis to generate the DOPG slows down the original program by a factor $\Delta_1$ in the range of 13 and 52. Similar results were also reported by Quante et al. Obtaining the global trace further increases runtime consistently by roughly 70% ($\Delta_1/\Delta_2$). Likewise, the number of events created per second (shown in the last row) is very consistent across the test cases.

### B. Naïve versus combined approach

To compare the naïve and combined approach, we selected the programs with multiple threads listed in Table I. Here we used FindBugs in an interactive mode with a GUI where multiple threads are used. The scenarios for the evaluation considered and the classes whose protocols were extracted are described in Table III.

As Table IV shows, determining the combined automaton is always much faster than calculating the shuffle and product automata – often by several orders of magnitudes. In addition to the scenarios reported in Table III, we ran test cases where the naïve approach did not yield any result after 5 hours, while the combined approach terminated in a few seconds. The bottleneck of the naïve approach is the computation of the protocol automaton, which depends upon the size of the shuffle automaton – which in turn depends by construction upon the number of shuffled threads. The table section on runtime for each pair of shuffle combination clearly indicates an increase of runtime with the number of threads combined. It should be noted that the table contains the number of threads sharing a common object. The actual number of threads is higher.

### VI. Conclusion

In this paper, we described an extension to Quante’s protocol reconstruction that correctly extracts protocols from programs with multiple threads. Since multi-threading programs are becoming omnipresent, this extension is highly relevant. The approach is mathematically based on shuffle and product automata. We gave a formulation of the problem that avoids creating the shuffle automaton explicitly. Our case studies show that our formulation scales very well as opposed to a naïve approach that creates the shuffle automaton explicitly. A disadvantage over the original approach by Quante is that our approach needs a global trace, which requires additional disk space linearly to the program execution and increases runtime of about 70% in comparison to the original approach.

To avoid creating the global trace, we are thinking about a way to create the ITSG online, too. One principle problem to
be solved here is to capture which thread created an edge in the ITSG online. If there is a thread executing nodes in the control flow graph leading only to irrelevant operations and another thread executing the same nodes leading to relevant operations, we obtain paths from irrelevant nodes to relevant operations so that paths from the irrelevant thread suddenly lead to relevant operations.

Table III: Scenarios and considered classes

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start ArgouML</td>
<td>Start ArgouML</td>
<td>Start ArgoUML</td>
</tr>
<tr>
<td>Create two classes</td>
<td>Create UMLStateDiagram</td>
<td>Create two classes</td>
</tr>
<tr>
<td>Create one association</td>
<td>Create UMLStateDiagram</td>
<td>Create UMLStateDiagram with 3 nodes</td>
</tr>
<tr>
<td>Save project to file</td>
<td>Save project</td>
<td>Save project to file</td>
</tr>
<tr>
<td>5. Exit ArgoUML</td>
<td>5. Exit ArgoUML</td>
<td>5. Exit ArgoUML</td>
</tr>
</tbody>
</table>

Table IV: Runtime resources

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td># threads</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>glob. trace MB</td>
<td>30</td>
<td>45</td>
<td>60</td>
<td>682</td>
<td>335</td>
</tr>
<tr>
<td>glob. trace mio. entries</td>
<td>2.6</td>
<td>3.9</td>
<td>5.4</td>
<td>58.7</td>
<td>28.8</td>
</tr>
<tr>
<td>step</td>
<td>time in milliseconds</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inline expansion (ms)</td>
<td>117</td>
<td>65</td>
<td>105</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>trace reading</td>
<td>20,201</td>
<td>29,544</td>
<td>41,704</td>
<td>206,289</td>
<td>236,116</td>
</tr>
<tr>
<td>ITSG const.</td>
<td>351</td>
<td>196</td>
<td>1,434</td>
<td>55</td>
<td>90</td>
</tr>
<tr>
<td>ITSG</td>
<td>20,669</td>
<td>29,805</td>
<td>43,243</td>
<td>206,359</td>
<td>236,241</td>
</tr>
<tr>
<td>threads extraction</td>
<td>71</td>
<td>47</td>
<td>68</td>
<td>12</td>
<td>21</td>
</tr>
<tr>
<td>intra-thread protocol aut.</td>
<td>370</td>
<td>214</td>
<td>425</td>
<td>51</td>
<td>90</td>
</tr>
<tr>
<td>combined calculation</td>
<td>441</td>
<td>261</td>
<td>493</td>
<td>63</td>
<td>111</td>
</tr>
</tbody>
</table>

be solved here is to capture which thread created an edge in the ITSG online. If there is a thread executing nodes in the control flow graph leading only to irrelevant operations and another thread executing the same nodes leading to relevant operations, we obtain paths from irrelevant nodes over irrelevant nodes to relevant operations so that paths from the irrelevant thread suddenly lead to relevant operations.