

# Online Construction of Dynamic Object Process Graphs

Jochen Quante

University of Bremen, Germany  
quante@informatik.uni-bremen.de

## Abstract

*A dynamic object process graph is a view on the control flow graph from the perspective of a single object. It has been shown that such a graph can be a useful starting point for many reverse engineering tasks, such as program understanding, protocol recovery, and feature location. In a previous paper, we described how dynamic object process graphs can be constructed from detailed program traces. However, the size of such traces is immense. Moreover, the instrumentation was described only for procedural languages. Object-oriented features such as exceptions require special treatment. Also, techniques like multithreading and reflection were not handled by our previous approach. This prevented it from being applicable to real-life systems.*

*In this paper, we introduce an alternative way of collecting the necessary data. By constructing an intermediate graph online, i.e., concurrently with the running program, the trace size problem is eliminated. Additional optimizations for both trace logging and online graph creation are discussed. In a case study, we show that running times are acceptable even for larger Java programs when these optimizations are applied. Additionally, we discuss some special issues arising from analyzing Java instead of C.*

## 1 Introduction

A dynamic object process graph describes the control flow of a program with respect to a single object. It contains only those parts of the control flow graph that are relevant (in terms of control flow) for a given object. In a previous paper [7], we showed that there are several useful applications of dynamic object process graphs in software reengineering. They can provide valuable insight into design and implementation of an application if the object is chosen carefully. For example, the socket is a good candidate object for a chat client, since communication is the main concern

of such an application. Also, dynamic object process graphs can be a good starting point for protocol recovery.

In the same paper, we also showed how such graphs can be constructed by tracing the execution of a program and then analyzing the resulting trace. However, due to the large amount of trace data and the high overhead of tracing to a file, we were only able to experiment with quite small programs (about 50 KLOC of C code). Also, the analysis was limited to single-threaded C programs. Since real-world applications are generally much larger, a more efficient way of collecting the necessary data is needed to make dynamic object process graph based analyses applicable in practice.

**Contributions.** In this paper, we show how raw object process graphs, which are the basis for a dynamic object process graph, can be constructed online efficiently, and how this even works for multithreaded and object-oriented programs. In a case study, we compare the CPU and hard disk resources required by the different approaches. By choosing real-life applications as subject systems, we show that both approaches are feasible in practice and therefore make dynamic object process graphs available for a wide range of systems. Furthermore, online construction enables additional applications.

**Overview.** The remainder of this paper is organized as follows. Section 2 defines what dynamic object process graphs are. Section 3 contains a short description of the previous approach. It shows up its problems and discusses possible optimizations. Section 4 introduces the online approach. In Section 5, we discuss some issues for Java tracing, and Section 6 is the report from our case study. In Section 7, we discuss related work.

## 2 Dynamic Object Process Graphs

Let us first define what a dynamic object process graph is. An object process graph is basically a projection of an interprocedural control flow graph (CFG).

Therefore we will first give a brief definition of a CFG.

An intraprocedural control flow graph is a graph with nodes  $N$  and edges  $E$ :

$$CFG := (N, E) \text{ with } E \subseteq N \times N$$

Each node  $n \in N$  and each edge  $e \in E$  can be of one of the following types:

$$T_N(n) \in \{\text{start, create, read, write, decision, call, entry, return, atomic\_call, end}\}$$

$$T_E(e) \in \{\text{call, return, seq, true, false}\}$$

Each node in the OPG represents a location in the program. The **start** node indicates the entry point of the OPG. A **create** node denotes the creation of an object – this must always be the first node after the **start** node, because an OPG needs an associated object. A **read/write** node represents a read or write access to the object’s attribute(s). A **decision** node marks a point where control flow can take two different ways, depending on the boolean value that was calculated in the previous operation or call. A **call** node always leads to a method **entry** node, and a **return** node leads back to the **call** node. The **call** node represents the method call site, while **entry** is the entry point of the called method. *Atomic* functions are functions that belong to the interface of the regarded object. Their internals may be hidden in an OPG, replacing their **call** nodes by **atomic\_call** nodes. Nodes of type **end** represent the end of the object’s lifetime.

Edges represent control flow between these locations. Control flow can either be unconditional (**seq**) or conditional (**true**, **false**). Conditional control flow edges may follow only a **decision** node, and there must be exactly one outgoing **true** edge and one outgoing **false** edge per decision node. 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.

A **dynamic object process graph** for a given object is a subgraph of the CFG. It is defined by the following construction:

1. Take all operations that relate to this object as initial nodes.
2. Add all nodes and edges on direct paths between operations on the given object that are really taken during execution. “Direct path” means that there may not be any unnecessary function calls or loops on the path.
3. Remove all nodes and subgraphs that do not contain relevant information, such as decision nodes

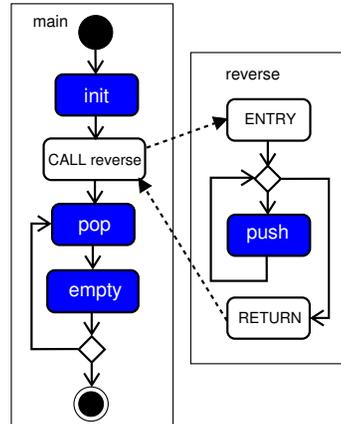


Figure 1. Dynamic object process graph.

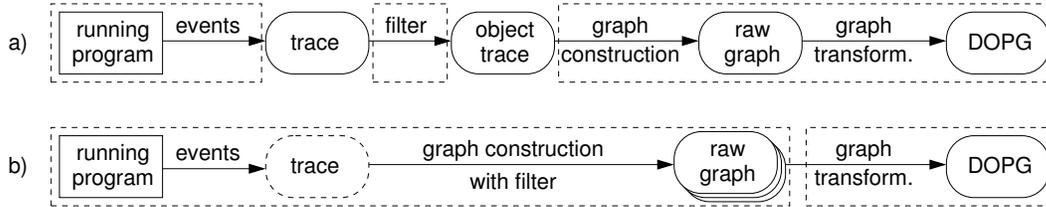
with just one exit, operations on other objects, and subgraphs which are not relevant for control flow. Connect the dangling edge pairs for each removed node or subgraph.

The resulting graph contains only the part of the CFG that is relevant for the given object with respect to control flow. Data flow is not regarded here. Figure 1 shows an example for a simplified dynamic object process graph for a stack object. Dark gray nodes represent calls to methods of the investigated stack class here, and diamonds represent decision nodes. For a more detailed introduction to dynamic object process graphs, see [7].

### 3 Offline Construction

Our previous approach [7] constructs the dynamic object process graph by instrumenting the program, collecting traces for all objects during execution and then evaluating these traces. Figure 2 summarizes this process. We call this approach “offline”, because the events are recorded and will only be processed after the application has terminated. The next paragraphs sketch the way the offline approach works.

**Instrumentation.** We instrument every location in the program that corresponds to a node in the CFG which is either relevant for any object that shall be traced for, or which has several predecessors or successors. Additionally, **entry** and **return** nodes are instrumented to capture function calls. Each instrumented location is labelled with a unique id. The added code leads to tracing of the passed location ids with object information written to a file. **true** and **false** are recorded as nodes, which eliminates the need to also trace edges.



**Figure 2. Construction of dynamic object process graph (DOPG) a) offline, b) online. In the online approach, the trace is not explicitly represented. Dashed rectangles represent a process.**

**Object trace extraction.** The resulting files contain the trace for the entire program and all objects that have been used during execution. Therefore, in this step, an *object trace* is extracted for every single object that we are interested in. This is done by filtering out all irrelevant method invocations. A method invocation is *relevant* if it either contains an operation on the regarded object or contains a relevant method invocation.

**Raw graph generation.** In the next step, all object traces for objects that have the same static allocation point (which means they are created at the same location in the source code) are used to construct a “raw” graph that contains all information needed to derive the dynamic object process graph from it. The raw graph differs from the final graph in having only one edge type (`true` and `false` are represented through nodes) and in containing a lot of additional nodes. This graph is necessary because later on, other paths to existing nodes may be added, and to capture control flow correctly, all the intermediate nodes must be kept in this step. Construction of this graph is quite straightforward. Nodes are added to the graph as they appear in the trace and connected to the previously read node. Nodes and edges that have already been constructed are just followed in the graph when they appear again. The raw graph’s nodes are therefore equivalent to the visited instrumentation points, and the edges indicate the order in which these points were visited.

**Dynamic object process graph generation.** On this raw graph, graph transformations are applied. This includes removing unnecessary nodes and edges, joining all return nodes of a function to one node, and removing subgraphs that are irrelevant for control flow. This can only be done after all traces have been integrated into the raw graph. After removing parts of the graph that are not reachable from the allocation point and parts that have no path to an operation on the object, the result is a dynamic object process graph.

**Drawbacks.** This approach involves several resource intensive steps:

- Traces become very large quickly, because a lot of

information has to be recorded. For example, for each condition node of the control flow graph that is passed during execution, we have to remember which branch has been taken. Tracing produces hundreds of megabytes within minutes. This implies a high I/O overhead.

- Individual object traces have to be extracted, and only then, the dynamic object process graphs can be constructed from that. In this step, the trace file has to be read again and again (for each individual object). Due to the size of the trace, this also takes quite some time (see Section 6).
- During the construction of the dynamic object process graph for each object, the object traces have to be read again. They contain a lot of redundant information, caused by cycles and repeated function calls.

**Optimizations.** A possible solution to the amount-of-data problem is instrumenting only those locations that we know can possibly be relevant for a given object. However, this would have to be decided statically, which would partly eliminate the advantages of dynamic analysis. In particular, it is often not possible to decide if a given node can potentially be in the object trace or not. This will specially happen when pointers are involved, or dynamic binding or reflection in the object-oriented world. In contrast, it is safe to regard all parts of the program as potentially relevant for an object trace, so selective instrumentation is not the best option.

Another solution is to compress the trace data. Reiss et al. [8] list a lot of possibilities to compress a dynamic trace. We use the following practices to compress our traces:

- Use the shortest possible identifiers for locations,
- use numbers to identify method signatures, since those can be quite long for Java programs (including package and parameters),
- use single characters to distinguish the node type.

<b>Graph</b>		<b>ThreadInfo</b>	
Nodes	Set of nodes contained in this graph; edges are directly kept within the nodes	PreviousNode	the node that was last added or visited in the graph by this thread
<b>Node</b>		AllNewEdges	set of all edges that have been preliminarily added to the graph within this thread
Type	type of the node ( $\in Type_N(N)$ )	RoutineStack	stack of RoutineInfo, which contains data that must be remembered per function on the call stack
ID	unique location identifier	<b>RoutineInfo</b>	
Succs	successors of this node, each along with a unique id for the corresponding edge	EntryPred	predecessor of this entry node (usually a call node)
<b>GraphView</b>		UseFlag	true if the function is already relevant for the object of this graph
Object	the object this graph is tracing	NewEdges	set of edges that have been preliminarily added to the graph within this func.
AllocID	unique id of the static allocation point		
Edges	edges of the CFG that are relevant for this object, i.e. which have been permanently added		
ThreadInfos	maps Thread to ThreadInfo		

**Figure 3. Data structures.**

These optimizations reduce the average amount of data required to store an event to 12 bytes. The runtime reduction compared to the original trace file format was only about 14%, while trace file size could be compressed to 15% of the original size. The amount of data could be reduced even further by using a binary format or applying online compression to the data. However, this would increase the runtime overhead.

The next paragraph describes an online processing approach to tackle the problem of handling large amounts of trace data.

## 4 Online Construction

A better approach for tackling the trace size problem would be to limit the trace to objects of the relevant class dynamically. The *relevant class* is the class whose instances the analyst is interested in, that is, whose instances are to be traced.

After returning from a called function, it is known if this invocation of the function was relevant for the regarded object or not. If the call was relevant, the trace of the called function has to be remembered, else it can be thrown away. The problem with this approach is that large amounts of trace data – in the extreme, the entire trace – have to be remembered before it can be decided if that data is needed or not, which can easily fill up all memory.

Therefore, this approach has to be complemented by a different representation of trace data. Loops have to be represented in an efficient way. Since we are interested in the “raw graph” as described above anyway, why not directly construct those graphs online instead of remembering all the trace events? The number of

nodes in this graph is limited by the number of nodes in the CFG, while the number of trace events is not limited. Therefore, it is possible to remember executed parts of the graph temporarily. Only when the program terminates, the resulting raw graphs need to be written to file, which eliminates the I/O overhead during execution. By allowing to just record the graphs for objects of certain classes, the number of graphs that must be constructed simultaneously is limited as well, which prevents explosion of additionally required memory. However, this approach is probably not adequate when lots of instances of a class are created.

**Algorithm sketch.** The algorithm could basically work as follows:

- As events occur, construct the corresponding graph. When a method is left, eliminate that call from the graph again. This way, always keep only all methods of the current call stack in this graph. It represents the path from the main routine to the current node. We call this graph the “current stack graph”.
- Whenever a new object of a relevant class is instantiated, create a copy of the current stack graph for this object.
- Apply each other event to the current stack graph and to all copies. This means that an edge is inserted between the previously visited node and the node that corresponds to the event’s location.

For the copies, remove method calls (the same way as for the construction of the current stack graph) only if the method invocation is not relevant for the object of this copy. As construction

goes on, this will discriminate the different graphs from each other.

The overall process is shown in Figure 2. The trace itself does not have an explicit representation. Events are directly integrated into the raw graphs (one for each object of interest). The basic instrumentation and the transformation from raw graph to dynamic object process graph remains the same as in our earlier approach.

**Data structures.** To allow an efficient implementation of this idea, we decided to have just one graph that contains all nodes and edges that were visited in the entire program run. The raw graphs for our objects and the current stack graph are then just views (i.e. subgraphs, called *GraphViews*) of the complete graph. We will not have any nodes without edges, so it is sufficient to keep track of the edges only. This can be done very efficiently by using edge numbers and a bit set implementation. Multithreading must be considered in the data structures because multiple threads may be constructing the same graph at different locations at the same time. The overall data structures used are shown in Figure 3. Additionally, we have to keep a mapping from objects to GraphViews.

**Algorithm.** With these data structures, the algorithm in Figures 4 and 5 can be applied for raw graph construction. Figure 4 shows the main loop for event processing. New GraphViews, based on the current stack GraphView, are created as new relevant objects are created. Nodes that are corresponding to the unique instrumentation location ids of the events are added to the different GraphViews. Here, special care must be taken of operations on relevant objects, which are only applied to the GraphView that belongs to that object.

Figure 5 describes in detail how a node is added to a GraphView. Note that adding a node may also lead to removal of preliminarily added nodes. This is true when a return event has been received, and the method invocation turns out to be irrelevant. To make sure that this information is available, on method entry a new RoutineInfo is created which collects the necessary information for this method. Edges that are not yet visible in this GraphView are preliminarily added, until we know whether the method invocation is relevant or not. Only when it turns out to be relevant, those nodes are permanently added to the GraphView. While NewEdges keeps track of edges that are not contained in the GraphView and that have not been preliminarily added in one of the calling methods, AllNewEdges contains all edges that have been preliminarily added in any methods on the call stack. This allows a very fast check and update of those sets.

Figure 6 shows an example in the process of con-

Start with an empty GraphView (“current stack GraphView”) and an empty graph.

```

foreach incoming event do
  if event is the creation of an object of the
  relevant class then
    create a deep copy of the current stack
    GraphView for this object.
  Create/lookup node n for event in the
  graph, according to its location id.
  if event is an operation on a relevant
  object (including creation of the object)
  then
    addNode(n, GraphView for this object)
  else
    foreach GraphView g do
      addNode(n, gr)

```

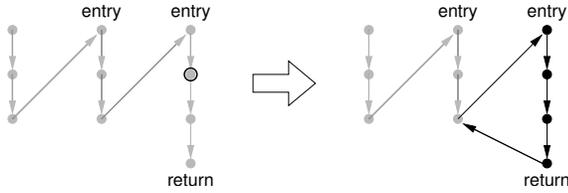
**Figure 4. Main algorithm.**

```

if ThreadInfos contains a ThreadInfo for the
current thread then
  use this ThreadInfo ti
else
  create new ThreadInfo ti
  add ti to ThreadInfos
if  $Type_N(n) = \text{entry}$  then
  ri := new RoutineInfo
  ri.EntryPred := ti.PreviousNode
  ti.RoutineStack.push(ri)
else
  ri := ti.RoutineStack.top()
Edge e := (ti.PreviousNode, n)
if  $e \notin \text{Edges}$  and  $e \notin \text{ti.AllNewEdges}$  then
  ti.AllNewEdges := ti.AllNewEdges  $\cup$  {e}
  ri.NewEdges := ri.NewEdges  $\cup$  {e}
if  $Type_N(n) = \text{return}$  then
  ri := ti.RoutineStack.pop()
  ti.AllNewEdges.removeAll(ri.NewEdges)
  if UseFlag is set then
    Edges := Edges  $\cup$  ri.NewEdges
  else
    ti.PreviousNode := ri.EntryPred
  UseFlag := UseFlag | ri.UseFlag
else if  $Type_N(n) = \text{operation}$  and operation
relates to the object for this GraphView then
  UseFlag := true
  ti.PreviousNode := n

```

**Figure 5. addNode: Adding to a GraphView.**



**Figure 6. Example: A relevant method invocation leads to permanent addition of edges to the GraphView.**

structuring a GraphView. On the left, edges have been preliminarily added to AllNewEdges and NewEdges for each routine (gray). Then, an operation on the GraphView’s object is encountered (black circle). This leads to setting the UseFlag, which in turn has the effect that the NewEdges are permanently (black) added to the GraphView when the routine is left (return node). Because the UseFlag is propagated up the call stack, all callers’ edges will also be permanently added to the GraphView’s Edges set later. If there had not been a relevant node within the routine, all edges from NewEdges would have been removed from AllNewEdges.

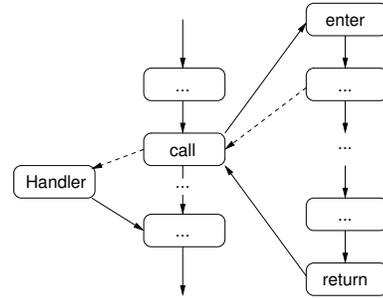
## 5 Java/OO Issues

In our previous study, we analyzed C programs and used graph transformations on an intermediate representation (based on abstract syntax trees) for instrumenting the subject systems. For our new case study, we focused on Java programs. The instrumentation is performed on byte code level using the ASM framework<sup>1</sup>. The advantage of this approach is that a system can be analyzed even when the source code is not or not completely available. Furthermore, instrumenting byte code is very fast and easy to do. On the other hand, this instrumentation is not reusable for other languages, and the connection to source code is only available on the source line level.

When dealing with Java, a few additional issues have to be considered, compared to simple (i.e. small and single-threaded) C programs. Some other issues just become more important.

**Object addresses.** Addresses of objects are not accessible in Java. This raises the question of how to get a unique identifier of an object, which is necessary for a full textual trace. However, using the `System.identityHashCode()` function delivers sufficiently disjunctive values.

<sup>1</sup><http://asm.objectweb.org/>



**Figure 7. Exception handling: Dashed edges denote exceptional control flow. The right method is left by an exceptional return edge, the exception is then caught by an exception handler within the calling method.**

**Accessing new objects.** New objects may only be accessed after the `java.lang.Object` constructor has been invoked. Before that, it is not possible to access the object. Therefore, we must distinguish between the point of creation of the object and the point where the object is accessible for the first time. New objects can be identified only at the latter point. For our implementation, this means that the object identity can only be assigned to the GraphView after the constructor has been visited. Therefore, calls of system constructors have to be treated accordingly.

**Multithreading.** In contrast to C, multithreading is very easy to do in Java. Also, even without explicitly starting new threads, any Java program – and surely any Java program with a GUI – always has several threads running in parallel. Therefore, a dynamic analysis on Java applications must take care of multithreading. For the offline approach, this was realized by writing one trace file per thread, which turned out to be faster than additionally writing a thread id into a single trace file. The online approach handles multiple threads through use of the `ThreadInfo` structure. In any way, tracing must be synchronized to work correctly, which additionally slows it down.

**Reflection.** In Java, classes and their methods can be accessed dynamically. *Reflection* allows to access a class based on its name, create instances of it and call its methods. Since reflection is widely used, we also have to consider this technique. Our byte code instrumentation recognizes calls to `java.lang.reflection` methods and handles them accordingly.

**Exceptions.** Exceptions are widely used for error handling in Java. They can heavily influence control flow. Exceptions lead to branches without an explicit condition and provide an alternative return mechanism for methods. Since control flow must be reconstructed

completely and correctly in the dynamic object process graphs, special care has to be taken for exceptions.

Our instrumentation adds default exception handlers around all method bodies that catch all `RuntimeExceptions` and all exceptions that the method is declared to throw. The occurrence of the exception is logged, and the exception is re-raised. Existing exception handlers are just prepended with an event logging call. This way, all exceptions are noticed and included in the trace.

The trace representation also requires an extension of the control flow graph for exceptions. Two additional edge types have to be introduced (see figure 7 for an example):

1. *exception*: An exception occurred and lead to this transition to an exception handler.
2. *exceptional return*: An exception occurred and caused premature exit from this method, which caused control to get back to the call node. From there, the exception may either be caught by a corresponding exception handler, or it is thrown further up the call stack.

**System classes.** Similar to library functions which cannot easily be instrumented for C code, the same is true for Java API classes. Parts of the API appear to be protected from being modified. The Java Virtual Machine produces all kinds of error messages when trying to modify certain classes. Therefore, we did not instrument classes belonging to the standard Java runtime environment at all. Consequences arising from this are discussed in Callbacks.

**Callbacks.** Callbacks are probably even more important in Java than in C. For example, most event handling in Java's GUI framework (AWT/Swing) is done through callbacks. These are usually realized by providing an object of a class that implements a given interface. This leads to calls of application code from system routines – and since system routines cannot be completely traced, this will cause incomplete and potentially misleading traces. Therefore, analysis must be robust against such effects. When there is an `entry` node without a `call` node, an artificial `call` node has to be created.

**Other OO features.** Other object-oriented techniques like dynamic binding, inheritance and polymorphism can be handled by our tracing approach. Since method entries are noticed, this tells us which method has really been entered with dynamic binding. Polymorphism is handled by taking into consideration the entire signature of a method. The default behavior concerning inheritance is to only regard instances of a given class, i.e. instances of subclasses are ignored.

However, invocation of superclass methods is of course recognized correctly.

All the above items are considered in our implementation. The next section reports on its application to different systems.

## 6 Case Study

In this case study, we compare online and offline construction of raw graphs. The raw graph is the common data structure that is produced by both approaches, so further processing is identical. The case study examines the instrumentation overhead in terms of running time and the number of trace events that occur within each program run. Also, the size of the resulting trace files is measured for the offline approach. Our goal is to show that construction of dynamic object process graphs is feasible using these methods even for larger programs. For being able to compare online and offline approach with Java programs, we reimplemented the offline approach for Java.

**Subject systems.** As subject systems, we investigate several Java programs of different size, tracing for potentially representative objects within typical use cases. Table 1 shows some size measures of the investigated systems.

**ArgoUML**<sup>2</sup> is a widely-used open-source UML modeling tool which supports all standard UML 1.4 diagrams. Graph models for the different diagrams must be a central concern for this tool. Therefore, as a first use case, we use the construction of a class diagram with `ClassDiagramGraphModel` as the relevant class. A class diagram for the observer pattern is drawn, consisting of four classes, one note, one aggregation, two inheritance relations and six methods. The result is saved, and the application is quit. The second use case is the construction of a sequence diagram (relevant class: `SequenceDiagramGraphModel`). Here, we create a new sequence diagram which consists of three actors and three synchronous interactions. The result is also saved. In the third use test case, we are interested in objects of the `Project` class. The actions performed include loading existing projects and creating new projects.

**J**<sup>3</sup> is a text editor with different editing modes, XML support, compiler/debugger, mail client and lisp interpreter. Sending emails is not a basic functionality for a text editor, therefore classes related to this feature were left out from instrumentation for this exper-

---

<sup>2</sup><http://argouml.tigris.org/>

<sup>3</sup><http://armedbear-j.sourceforge.net/>

	LOC	Classes	Methods	Bytecode size	Instrumented	Ratio	Time	inserted calls
ArgoUML	264K	4,285	32,263	17,943 KB	25,133 KB	+40.0%	13.9s	436,766
J	158K	1,277	9,081	5,782 KB	9,670 KB	+67.3%	7.1s	197,516
JHotDraw	71K	398	3,422	1,650 KB	2,258 KB	+36.9%	2.5s	41,037

**Table 1. Subject system properties and static instrumentation overhead.** Instrumented **byte code size**, Ratio of instrumented byte code size to original bytecode size, CPU Time needed for instrumentation.

Test case	number of		mio events		CPU time [s]					trace size [MB]
	threads	obj.	offline	online	original	offline	+filter	=sum	online	
<i>ArgoUML:</i>										
- ClassDiag.	12	1	23.2	25.8	19.0	130.4	29.8	160.2	155.0	248.3
- SequenceDiag.	9	1	12.2	13.7	16.2	59.4	15.8	75.2	62.8	130.7
- Project	10	2	12.7	11.3	12.6	46.0	29.2	75.2	43.0	119.0
<i>J:</i>										
- Editor	12	1	18.7	21.2	3.7	81.3	28.5	109.8	58.5	191.4
- JavaMode	20	1	9.3	8.1	3.5	29.4	9.6	39.0	24.5	82.0
<i>JHotDraw:</i>										
- ZoomDrawView	4	1	1.1	1.4	3.4	7.6	1.8	9.4	6.4	11.5
- QuadTree	3	10	0.9	0.9	3.7	7.3	12.3	19.6	10.1	9.9

**Table 2. Measurement of data collection performance.**

iment<sup>4</sup>. The first use case concerns the `Editor` class, which appears to be the central class of `J`. We start `J`, load files, create a new file, copy and paste, save a file, close files, and exit. In the second use case, we are looking at the `JavaMode` class, which is responsible for Java specific behavior. We load a Java file, use the Java tree display to jump to different methods, and fold and unfold methods (i.e. hide the body).

`JHotDraw`<sup>5</sup> is a Java GUI framework for graphical applications. It was originally written as an example for application of design patterns but is now used in many applications as well. The tests were performed using the sample `JavaDraw` application that comes with `JHotDraw`. As first use case, we investigated the `ZoomDrawingView` class, the view that contains all the visible objects that are drawn. We create a new drawing, draw eight rectangles, start the animation for 5 seconds, stop it again, then zoom into the image, and exit. The second use case traces for instances of `QuadTree`, which is probably used to partition a drawing. A new drawing is created, eight rectangles are drawn, all eight rectangles are deleted again one after the other, and the application is quit.

**General procedure.** All the use cases have to be performed interactively, therefore it is important to al-

ways execute user commands in the same order with similar timing. In order to limit the influence of such timing differences, we executed each use case at least three times for each of the different versions. The three versions consist of the unmodified code (original), the instrumented code with trace logging (offline), and the instrumented code with raw graph construction (online). Then we took the average of these runs. Also, we measured execution times in terms of CPU time instead of elapsed real time to further minimize those effects. All use cases and time measurements were performed on a 3 GHz Pentium IV machine.

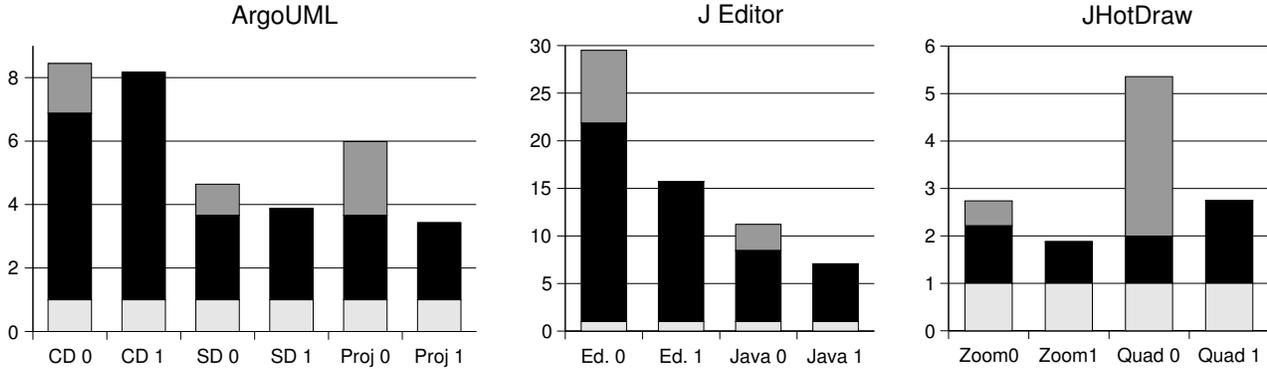
**Static instrumentation overhead.** The right part of Table 1 provides an overview of the cost for doing the instrumentation. Byte code manipulation is the same for both the online and offline approach, so we do not differentiate this. The approaches only differ in what happens in the functions that are called by the instrumented code.

Instrumentation is done very fast. Up to 1.3 MB of byte code are instrumented per second, or up to 2,300 methods per second. The number of inserted calls to tracing routines averages between 100 and 150 per class. This increases byte code size by up to 67%, which is partly due to the location identification strings that have to be stored for each event location.

**Data collection overhead.** The overhead produced by tracing for dynamic object process graphs with the online and offline method is shown in Table 2.

<sup>4</sup>This was done to circumvent problems with the ASM framework that occurred for some classes of the mail part which contained abnormal byte code.

<sup>5</sup><http://www.jhotdraw.org/>



**Figure 8. CPU time [s] consumed by online and offline tracing in comparison to normal program execution. 0=offline, 1=online. The light gray area indicates normal program execution time, the black area denotes the additional tracing overhead. The dark gray offline area indicates time consumed by filtering for object traces.**

Offline construction requires the additional object trace filtering step and the following raw graph construction, so these three steps have to be combined to be comparable to online tracing. These two offline steps will be called “filter”.

The use cases differ in the number of objects that are created of the relevant class and in the number of concurrently running threads. This also has a great impact on the tracing overhead. In the online approach, for each additional object, another `GraphView` has to be created and maintained, so runtime will increase. For the offline approach, additional objects only become important in the filter phase, because then, an own object trace must be extracted for each of the objects. When many threads are involved, they must all be synchronized, so there will be more waiting time than with a few threads. This affects both the online and the offline approach. The online approach additionally has to create and update special data structures for each thread.

The number of events that occur during use case execution differs between the online and offline approach, although the same actions have been performed. This is due to the fact that timing changes, specially in the presence of multithreading. In this way, instrumentation modifies application behavior. For ArgoUML, a longer running time corresponds to more events, which is not true for the other applications.

The trace files produced by the offline approach consumed 248 MB for the largest use case. This trace contains the events of only 19 seconds CPU time. When running even larger use cases, trace file size will increase accordingly. This illustrates the need for alternatives that do not need to store those large amounts of data.

Concerning execution times, in some cases the online variant requires more CPU time, in other cases this is true for the offline variant. When taking into consideration the additional filtering effort required for the offline approach, the online variant is always faster. Figure 8 visualizes CPU time ratios for the different use cases. The y-axis denotes the factor to which the consumed CPU time increases. The use cases as executed with the original program are normalized to 1. The black parts of the bars depict the overhead during execution, while the gray area corresponds to the additionally needed filtering for the offline approach. Execution time overheads differ a lot, from 88% for the `ZoomDrawingView` (online) up to 2,085% for the `Editor` (offline). J even has a higher overhead than ArgoUML, although ArgoUML supposedly is the more complex application. Apparently, J does a lot of background work (such as autosaving) which produces lots of events.

In only one case, the online approach is remarkably slower: In the `JHotDraw` use case with 10 simultaneously followed objects. This can be explained by the increased overhead of constructing multiple `GraphViews`. Therefore, for cases when many instances of a class are created, the offline approach is probably more efficient concerning application runtime. On the other hand, the effort required for offline filtering also increases and here even exceeds the application runtime. In sum, the CPU time required for the offline approach is twice as high as for the online approach in this case.

Another observation is that use cases with many threads (J, `ClassDiagram`) generally have a higher runtime overhead when instrumented. This strengthens our assumption that performance loss will be higher in the presence of many threads.

From the user's view, the applications were usable nearly as normal in all cases. The tracing overhead did not slow down the applications to an unacceptable degree. A maximum CPU usage increase by factor 22 for the J editor may seem very high but was not disturbing in practice. This is probably due to the fact that only GUI driven applications were tested. Performance for batch applications is yet to be analyzed.

## 7 Related Work

A lot of work has been published in the area of dynamic program analysis. One main challenge is how to abstract from the collected information, since the amount of data is usually very high. Reiss et al. [8] describe different encodings that can help reducing a trace's size, and Hamou-Lhadj et al. [3] introduce a framework for lossless trace compression.

One of the first approaches to dynamic visualization of object oriented systems is presented by De Pauw et al. [6]. They introduce a general instrumentation scheme, a protocol for communication between instrumented program and visualization component, and different visualizations. Their visualizations focus on getting an overview of relations and instantiated objects. Lange et al. [5] describe a dynamic analysis that collects method invocation data along with the associated objects. They use merging, pruning and slicing to reduce the search space. The selection of which classes to include is done manually. The idea of merging and pruning also appears in our approach. Jerding et al. [4] create a compacted dynamic call tree to reduce the trace size. They visualize the entire trace graphically, which gives evidence of "interaction patterns". These can be found by various pattern matching algorithms. Systä [10] creates scenario diagrams (i.e. basically sequence diagrams) from traces. Trace sizes are reduced by string-matching based detection of repetitions and subscenarios. State diagrams for single objects are then inferred from these scenarios, also using program location information ("state boxes") to avoid over-generalization. This approach comes closest to ours. Richner and Ducasse [9] present an iterative dynamic analysis which aims at extraction of collaborations from runtime information. They record method invocations and perform pattern matching on that information.

Multithreading is not explicitly considered in most dynamic analysis approaches, although the order of trace events is important for many analyses. The paper by Boroday et al. [1] is one of the few that addresses the topic of dynamic analysis and multithreading. They use the Hyades framework (now part of the Eclipse

Test & Performance Tools Platform Project) which is capable of handling multiple threads.

Breech et al. [2] compare dynamic impact analyses when being performed online and offline.

## 8 Conclusion and Outlook

We have shown that dynamic object process graphs are applicable in practice, even for larger interactive systems. The optimizations for the offline approach and the newly introduced online approach both allow tracing applications with an acceptable overhead. The online approach turned out to be always faster when investigating a single object, while the offline approach may lead to reduced application runtime when following many objects simultaneously.

Online construction of dynamic object process graphs also opens new application potentials. For example, the evolving graph could be shown while the application is running, providing information about which relevant parts of the application have already been visited. Also, multithreading information may be further exploited to detect potential concurrency problems with the help of dynamic object process graphs.

## References

- [1] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. In *Proc. of the 3rd Int'l Workshop on Dynamic Analysis*, pages 1–7, 2005.
- [2] B. Breech, M. Tegtmeier, and L. L. Pollock. A comparison of online and dynamic impact analysis algorithms. In *Proc. of the 9th CSMR*, pages 143–152, 2005.
- [3] A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proc. of IWPC*, pages 159–168, 2002.
- [4] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proc. ICSE*, pages 360–370, 1997.
- [5] D. B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, 1997.
- [6] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. of OOPSLA*, pages 326–337. ACM Press, 1993.
- [7] J. Quante and R. Koschke. Dynamic object process graphs. In *Proc. of the 10th CSMR*, pages 81–90, 2006.
- [8] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of ICSE*, pages 221–230, 2001.
- [9] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. of the 18th ICSM*, pages 34–43, 2002.
- [10] T. Systä. Understanding the behavior of java programs. In *Proc. of WCRE*, pages 214–223, 2000.