SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2411

Compact visualization of Java program execution

S. Jayaraman[1,2,*,†], B. Jayaraman[1,2] and D. Lessa[1,2]

[1]*Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham, Amrita University, Amritapuri, India*
[2]*Department of Computer Science and Engineering, State University of New York at Buffalo, Buffalo, NY, USA*

SUMMARY

The context of this work is a practical, open-source visualization system, called JIVE, that supports two forms of runtime visualizations of Java programs – object diagrams and sequence diagrams. They capture, respectively, the current execution state and execution history of a Java program. These diagrams are similar to those found in the UML for specifying design–time decisions. In our work, we construct these diagrams at execution time, thereby ensuring continuity of notation from design to execution. In so doing, a few extensions to the UML notation are proposed in order to better represent runtime behavior. As sequence diagrams can become long and unwieldy, we present techniques for their compact representation. A key result in this paper is a novel labeling scheme based upon regular expressions to compactly represent long sequences and an $O\left(r^2\right)$ algorithm for computing these labels, where r is the length of the input sequence, based upon the concept of 'tandem repeats' in a sequence. Horizontal compaction greatly helps minimize the extent of white space in sequence diagrams by the elimination of object lifelines and also by grouping lifelines together. We propose a novel extension to the sequence diagram to deal with *out-of-model* calls when the lifelines of certain classes of objects are filtered out of the visualization, but method calls may occur between in-model and out-of-model calls. The paper also presents compaction techniques for multi-threaded Java execution with different forms of synchronization. Finally, we present experimental results from compacting the runtime visualizations of a variety of Java programs and execution trace sizes in order to demonstrate the practicality and efficacy of our techniques. Copyright © 2016 John Wiley & Sons, Ltd.

Received 18 May 2015; Revised 5 April 2016; Accepted 13 April 2016

KEY WORDS: visualization of Java program execution; sequence diagrams; horizontal and vertical compaction; regular-expression labels; tandem repeats; exclusion filters; out-of-model calls; multi-threaded execution; experimental results

1. INTRODUCTION

The goal of our research is to facilitate the comprehension of the runtime behavior of object-oriented programs. The complexity of modern object-oriented software is a consequence of many factors, including the use of indirection, inversion of control and increased method interaction. These, together with event-driven programming, concurrency, and graphical user interfaces have greatly compounded the situation. Despite the availability of modern runtime environments, such as Eclipse, NetBeans, and Visual Studio, software maintenance dominates the total cost of the software development life cycle. This is due to the fact that code changes are often made by programmers who did not develop the original code. Thus, program comprehension is an extremely important albeit time-consuming activity in the software development process, and techniques that provide insight into the working of a program are crucial.

Towards this goal, we have been developing a state-of-the-art visualization, debugging, and dynamic analysis system for Java programs, called JIVE (for Java Interactive Visualization Environment) [1]. JIVE supports a number of useful features, including visualization of object

*Correspondence to: S. Jayaraman, Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham, Amrita University, Amritapuri, India.
†E-mail: swaminathanj@am.amrita.edu

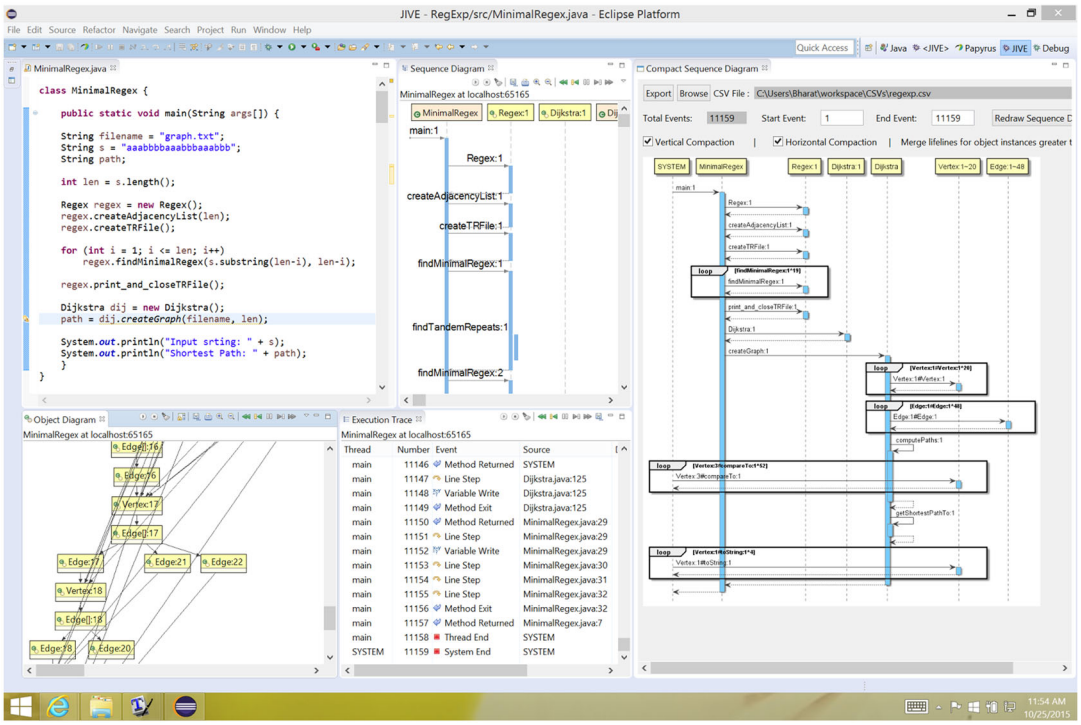


Figure 1. JIVE interface showing source code, execution trace, object diagram, sequence diagram, and also the compact sequence diagram.

structures and execution history, interactive forward and reverse stepping, query-based debugging [2], dynamic slicing of programs, and execution summarization. The system has been used extensively for teaching and learning Java [3] and also serves as a visual Java debugger and is available as a plugin for Eclipse [4].

The focus of this paper is on JIVE's visualizations, for which two types of diagrams are particularly useful: object diagrams and sequence diagrams. An object diagram captures the current state of execution in terms of the active objects and their interconnection, whereas the sequence diagram captures the history of method interactions between objects. Figure 1 is a screenshot of the JIVE interface showing a fragment of the source code for a program that computes the minimal regular expression for a string (discussed later in the paper) and the visualization of its execution in terms of sequence and object diagrams.

These diagrams are similar to those found in the UML for specifying design–time decisions. The important aspect of our work is that we construct these diagrams at execution time, thereby ensuring continuity of notation from design to execution. In so doing, a few extensions to the UML notation are proposed in order to better represent run time behavior. For example, JIVE object diagrams also depict method activations in object contexts (Figure 1), and JIVE sequence diagrams also support the concept of 'out-of-model' calls (Figure 9). An important property of JIVE is that every point on the sequence diagram is associated with the object diagram that would have been in effect at that point in execution. Thus, the sequence diagram serves as an effective temporal navigation tool, allowing a user jump to any point in the execution history and inspect the object diagram at that particular time.

A fundamental problem with visualizations, however, is that they tend to become unwieldy as the number of objects and interactions grows. The complexity of object diagrams depends on not only the number of objects but also the number of fields within objects, nested objects, and references between objects. The complexity of sequence diagrams depends upon the number of objects as well as the interaction between the objects. Whereas an object diagram is concerned about the state at a particular point in the execution history, the sequence diagram is concerned about the entire

DOI: 10.1002/spe

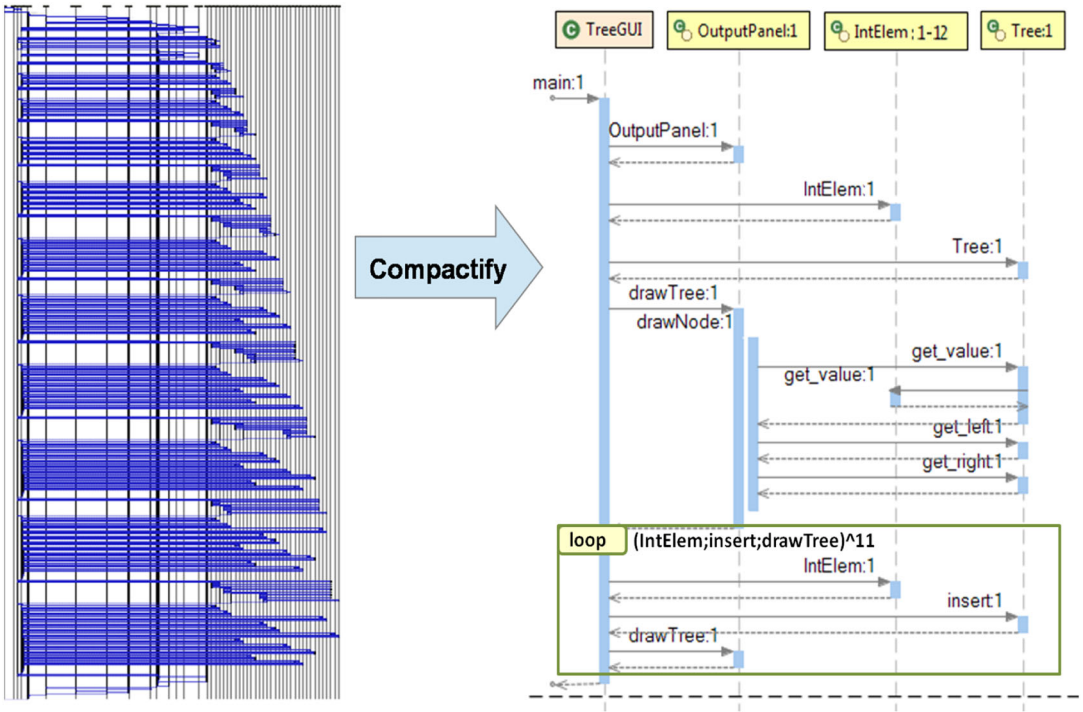


Figure 2. Horizontal and vertical compaction on tree insertion with Graphical User Interface. The sequence diagram on the left, shown in reduced form, consists primarily of a sequence of 11 IntElem node creation, tree insertion, and tree drawing operations. This diagram is shown in compacted form on the right, with label $(\text{IntElem; insert; drawTree})^{11}$.

execution history, and it grows in size and complexity as the execution proceeds. Thus, the compact representation of a sequence diagram is a more challenging problem than compact representation of object diagrams. The length of the sequence diagram depends upon the number of interactions, while the width depends upon the number of objects. The horizontal separation between interacting objects in a sequence diagram can introduce considerable white space into the diagram and greatly inhibit its comprehension. In order for the diagrams to be effective, a visualization tool must present them in a manner that enhances program comprehension.

In our earlier work, we examined techniques for compact representation of object diagrams [1, 4], and thus, the focus of this paper is on compact representation of sequence diagrams. This has been a topic of considerable interest in the literature [5–13]. We provide a more detailed comparison with these approaches in Section 7, but the distinguishing feature of our approach is the use of labeling scheme for the compacted structures based upon regular expressions. For example, a sequence of k calls of a method m would result in a linear growth in the sequence diagram, but the compacted representation would have just one method call with label m^k. Our experimental results in Section 6 show that this approach is effective in labeling the compacted representation for many practical programs. We refer to this as vertical compaction, and we propose a general technique for compacting a sequence of method calls in order to obtain a minimal label in Section 3.

Figure 2 shows how a repetitive pattern of ‘insert’ and ‘draw’ operations on a binary search tree with a graphical user interface may be compacted. As this example illustrates, compaction can take place in both the vertical and horizontal dimensions. In a sequence diagram, the vertical dimension is for time, the start of program execution being at the top of the diagram. Along the horizontal dimension are placed the lifelines of objects, typically the left-to-right order reflecting the temporal order in which objects were created.

Thus, the main contributions of this paper are as follows:

1. Vertical compaction of a sequence of method invocations using regular-expression labels and an $O(r^2)$ algorithm for computing these labels, where r is the length of the input sequence,

DOI: 10.1002/spe

based upon the concept of *tandem repeats* in a sequence [14]. The labeling and compaction can be applied in a top-down or bottom-up fashion to achieve automatic compaction of sequence diagram and support progressive expansion.
2. Horizontal compaction of object lifelines, supporting both the elimination of object lifelines as well as grouping lifelines together. Horizontal compaction can also occur because of vertical compaction of nested calls and when methods belonging to filtered classes are invoked.
3. The introduction of exclusion filters to allow the user to remove interactions that are not of interest, referred to as *out-of-model* calls, and a novel extension to sequence diagrams to allow regular calls as well as out-of-model calls in the diagram.
4. Extension of the aforementioned techniques for multi-threaded programs and the introduction of new notations for compact representation of thread interactions that are overlapping and possibly interfering with one another.
5. Experimental results based upon the execution of a variety of Java programs, including programs with long executions, using the JIVE system in order to demonstrate the scalability and practicality of the compaction strategy.

Apart from horizontal and vertical compaction, geometric zooming out/in is supported in JIVE for intra-object and inter-object visualizations as well as for sequence diagrams. Both compaction and zooming can be initiated explicitly by the user through JIVE's user interface.

The rest of the paper is structured as follows. Section 2 provides a brief overview of the JIVE architecture and its event-based execution model. Section 3 defines the call tree representation for sequence diagrams and presents the algorithms for labeling and vertical compaction. Section 4 presents our approach to horizontal compaction using the concept of 'out-of-model' calls. Section 5 explains how we can adapt our strategies for multi-threaded Java programs. Section 6 presents the compaction results from testing a variety of Java programs. Section 7 presents the related work in this area, and Section 8 presents a summary, conclusions, and directions of future work.

2. JIVE ARCHITECTURE

We provide a brief overview of underlying architecture on which the JIVE's functionalities are based before we proceed with our presentation of compaction of sequence diagrams. JIVE is a versatile dynamic analysis tool suitable for a number of applications: software debugging and comprehension, dynamic visualizations, teaching programming languages, and software engineering. In some aspects, JIVE resembles a traditional debugger, allowing one to define breakpoints, inspect variables, step into and over instructions, and so on. However, JIVE provides features that transcend the abilities of traditional debuggers: dynamic visualizations of the runtime state and execution history, query-based debugging, and interactive forward and reverse stepping.

In order to support these features, JIVE runs Java programs in debug mode while collecting *execution events* corresponding to particular instructions of the program such as variable assignments, method calls, and object creation. These events are collected into an in-memory model. Derived models are constructed in order to support the dynamic visualizations. JIVE's implementation is based on a model–view–controller architecture, and the main components of which are illustrated in Figure 3.

The debugger part of JIVE is implemented on top of the Java platform debugger architecture (JPDA), an event-based debugging architecture where debugger and debuggee tiers run in separate Java virtual machines (JVMs). The types of event supported by JPDA are

- Virtual machine start, death, and disconnect
- Class prepare and unload
- Thread start and death
- Method call and return
- Field access and modification
- Exception thrown and caught
- Line step

DOI: 10.1002/spe

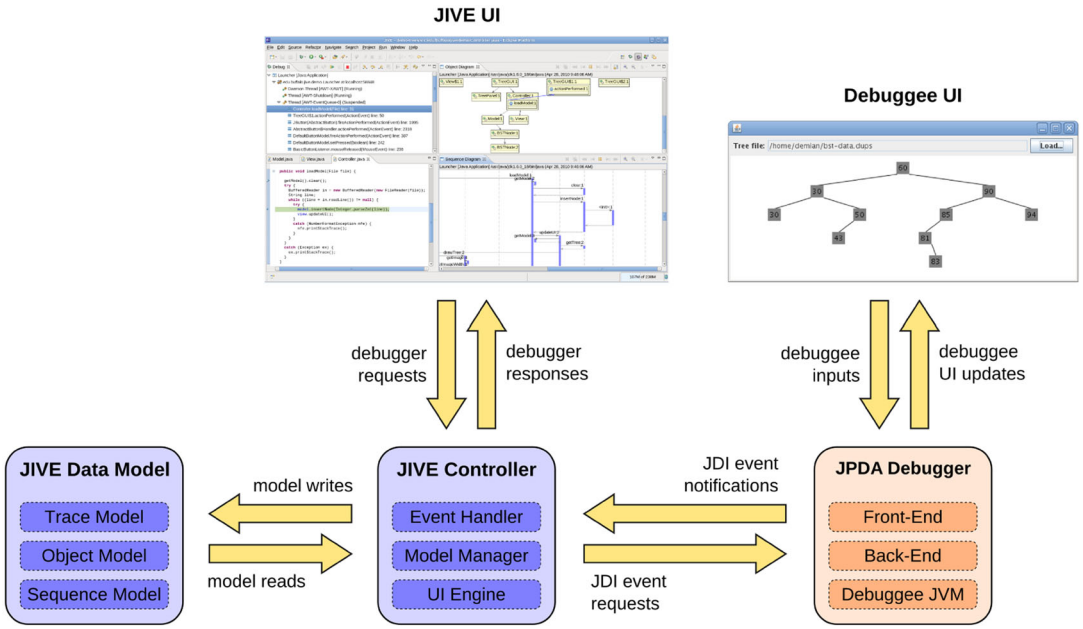


Figure 3. JIVE architecture.

The sequence diagram captures the history of object-to-object interactions, and hence, method call and exit events are of particular interest to us. The method call and exit events are of the form

- Method call: ⟨*thread, event-id, caller, target*⟩
- Method exit: ⟨*thread, event-id, returner, value*⟩

where the *caller*, *target*, and *returner* are each identified as an invocation instance of some method in the context of some object. This invocation instance is represented as *m:i # C:j*, where *m:i* is the i^{th} invocation instance of method *m* in the context of object *C:j*, that is, the j^{th} instance of class *C*. The sequence diagram can be drawn with these events.

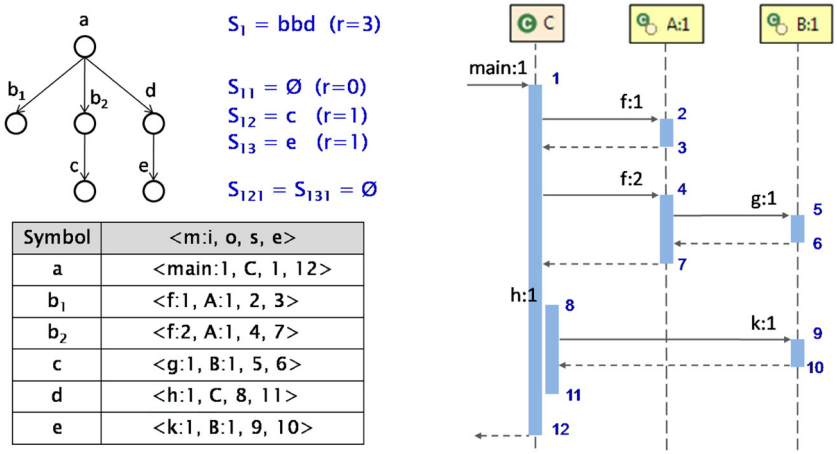
This event-driven architecture of JIVE is crucial to our approach to sequence diagram compaction, especially for multi-threaded execution, which we shall examine in detail in Section 5. The JIVE controller requests events from JPDA and processes event notifications received from JPDA. The model manager receives events from the event handler and triggers appropriate model changes. Finally, the UI engine uses data contained in the models to update the object and sequence diagrams.

3. CALL TREES AND VERTICAL COMPACTION

Call trees are fundamental structures that capture the hierarchic caller–callee relation of procedural program execution. At execution time, a call tree grows in a depth-first manner; when execution has reached a particular node, all nodes on the path from the root of the tree to this node represent the stack of outstanding method calls. The sequence diagram can be viewed as a call tree that is displayed in a horizontal left-to-right manner, except that object lifelines are not captured in a traditional call tree. (A more precise correspondence is given further in the succeeding text.) As objects are an important component of a sequence diagram, in our representation, each node of the call tree also has the identity of the object in whose context the call was made. Our interest in call trees stems from the fact that it facilitates the formulation of vertical compaction of sequence diagrams, because we can replace a subtree by a single node or a collection of sibling nodes by a single node.

Definition 3.1
A *node* in a call tree represents a method call of the form $\langle m : i, o, s, e \rangle$, meaning that the i^{th} call of method *m* in context *o* started at event number *s* and ended at event number *e*. The context for

DOI: 10.1002/spe



Symbol	<m:i, o, s, e>
a	<main:1, C, 1, 12>
b_1	<f:1, A:1, 2, 3>
b_2	<f:2, A:1, 4, 7>
c	<g:1, B:1, 5, 6>
d	<h:1, C, 8, 11>
e	<k:1, B:1, 9, 10>

Figure 4. One-to-one correspondence between call tree and sequence diagram.

an *instance method* is an object ID, and the context for a *static method* would be a class name. Each event has a unique number, and the set of all event numbers is a totally ordered set.

Definition 3.2
A *directed edge* in a call tree is from a node $n_1 = \langle m_1 : i_1, o_1, s_1, e_1 \rangle$ to a node $n_2 = \langle m_2 : i_2, o_2, s_2, e_2 \rangle$, meaning that there is a direct call from node n_1 to n_2. Furthermore, $s_1 < s_2 < e_2 < e_1$.

The root of the call tree is a special node $n = \langle \texttt{main}, C, s, e \rangle$, where C is the class in which method `main` is defined, and s and e are its start and end events.

Given a call tree with N nodes, let the set of objects contained in these nodes be $\{o_1, \ldots, o_m\}$, for some $m \leqslant N$. Then the sequence diagram with m object lifelines is related to the call tree as follows.

- The leftmost lifeline corresponds to the root of the tree, $\langle \texttt{main}, C, s, e \rangle$, with an activation box called `main`.
- Every non-root node $n = \langle m : i, o, s, e \rangle$ in the call tree corresponds to an activation box with label $m : i$ on the lifeline of object o in the sequence diagram.
- In a depth-first traversal of the call tree from the root, if object o_i is found in some node before o_j, then the lifeline for an object o_i precedes that of o_j in the sequence diagram.
- Let the child nodes of n in the call tree in left-to-right order be $n_1, \ldots, n_k$. If $n_1 = \langle m_1 : i_1, o_1, s_1, e_1 \rangle \ \ldots \ n_k = \langle m_k : i_k, o_k, s_k, e_k \rangle$, then there are k method calls from the activation box of n to activation boxes on the object lines of $o_1 \ldots o_k$, respectively, with labels $m_1 : i_1 \ldots m_k : i_k$, respectively. Furthermore, $s_1 < e_1 < \ldots < s_k < e_k$.
- An activation box for node $n_1 = \langle m_1 : i_1, o, s_1, e_1 \rangle$ precedes the activation box for node $n_2 = \langle m_1 : i_1, o, s_2, e_2 \rangle$ on the lifeline of object o if $e_1 < s_2$.

Figure 4 shows an example of a call tree and its sequence diagram. The symbol next to each node is elaborated in the adjoining table in the figure. The call sequence made by each node can be written as a sequence of symbols. For example, sequence $S_1 =$ bbd denotes the sequence of three calls made by 'a'. The call sequence of each node is given to the right of the call tree in Figure 4. Note that the call sequence of leaf nodes is null. This form of representing the calls made by each node as a sequence of symbols facilitates folding of the call tree, which will be detailed in the subsequent subsections.

As noted earlier, call trees facilitate compaction operations because they can be viewed as replacing entire subtrees by a single node. An important consideration in diagram compaction is the ability to reconstruct the original diagram. In the context of call trees, we need to insert subtrees back to

DOI: 10.1002/spe

their respective positions when needed. In the next subsections, we define the two main operations needed for obtaining maximally compact sequence diagrams: labeling and compaction. We also briefly present the expansion of a compacted sequence diagram.

3.1. Labeling

Vertical compaction of a sequence diagram involves replacing a sequence of consecutive activation boxes by a single box with a regular expression label that is computed from the method names associated with the activation boxes. In terms of the call tree, consecutive activation boxes correspond to a sequence of child nodes of some node. Thus, given a node n with r child nodes $n_1 = \langle m_1 : i_1, o_1, s_1, e_1 \rangle \ldots n_r = \langle m_r : i_r, o_r, s_r, e_r \rangle$, the regular expression label is constructed from the sequence of method names $m_1 \ldots m_r$. Note that these method calls can be on different contexts (objects or classes) $o_1 \ldots o_r$, although often the context will be the same. It is acceptable for the contexts to be unrelated to one another. Also, we do not distinguish overloaded methods in our labeling, because overloaded methods typically implement similar functionality.

ALGORITHM 1: MinRE(S), Minimal Regular Expression Computation

Input: S representing a sequence of r calls
Output: MinRE(S)
1. Find all primitive *tandem repeats* of S of the form $[i, \alpha, k]$;
2. Create a graph with $r + 1$ vertices: $V_1 \ldots V_{r+1}$;
3. Add a directed edge from V_i to V_{i+1} for $i = 1$ to r of weight 1;
4. Label each edge (V_i, V_{i+1}) with the symbol $S[i]$ for $i = 1$ to r;
5. **for** *every tandem repeat of the form* $[i, \alpha, k]$ **do**
   Add a directed edge between V_i and $V_{i+|\alpha|.k}$ of weight 1;
   Label the edge with α^k;
6. Run Dijkstra's shortest path algorithm to determine the sequence of edges from V_1 to V_{r+1};
7. Concatenate the labels of these edges to get the minimal regular-expression $MinRE$;

Algorithm 1 shows how minimal regular-expression labels are computed for a string sequence S of length r, using the concept of tandem repeats of a sequence. First, all primitive *tandem repeats* of S of the form [i,α,k] are computed using the $O(r \times log(r))$ algorithm from [14]. The term 'tandem repeat' means 'consecutive occurrences'. Here, i represents the start index position of a tandem repeat, α represents the substring that is repeated, k is the number of consecutive repetitions and is > 1. Figure 5 (ii) illustrates the tandem repeats for a sample sequence $S = abaababa$. The details of computing tandem repeats of a sequence and the complexity analysis can be found in [14].

Once the tandem repeats are determined, they can be combined with the remaining characters in multiple ways to represent the original sequence. Possible short representations for $abaababa$ include $(aba)^2ba$, $aba^2(ba)^2$, and $aba(ab)^2a$ of lengths 3, 4, and 5, respectively. Our goal is to find the shortest combination.

To this end, we construct a directed acyclic graph of $r + 1$ vertices $V_1 \ldots V_{r+1}$ with r edges, one each from V_i to V_{i+1} of weight 1. The edges are labeled with $S[i]$. Now, for every tandem repeat that we computed previously, we add an edge of weight 1 from V_i to $V_{i+|\alpha|.k}$. We label each of these edges with the corresponding α^k. The time taken by this step is equal to r plus the number of tandem repeats (computed in the first step). In terms of the directed acyclic graph, this is equivalent to the number of edges added to the graph. The worst case upper bound on the number of edges in a directed acyclic graph of $r + 1$ vertices is given by $r + (r - 1) + \ldots + 1 = r * (r + 1)/2$. Hence, the time complexity of steps 2 through 5 is $O\left(r^2\right)$.

The next step is to find the shortest path from V_1 to V_{r+1} using Dijkstra's shortest path algorithm. If there are multiple paths with the same shortest distance, we let Dijkstra's algorithm choose one arbitrarily. The time complexity of Dijkstra's algorithm is established as $O(V + E)$ for V vertices and E edges. In this case, we have $r + 1$ vertices and at most r^2 edges, and hence, the time complexity of step 6 works out to $O\left(r^2\right)$.

DOI: 10.1002/spe

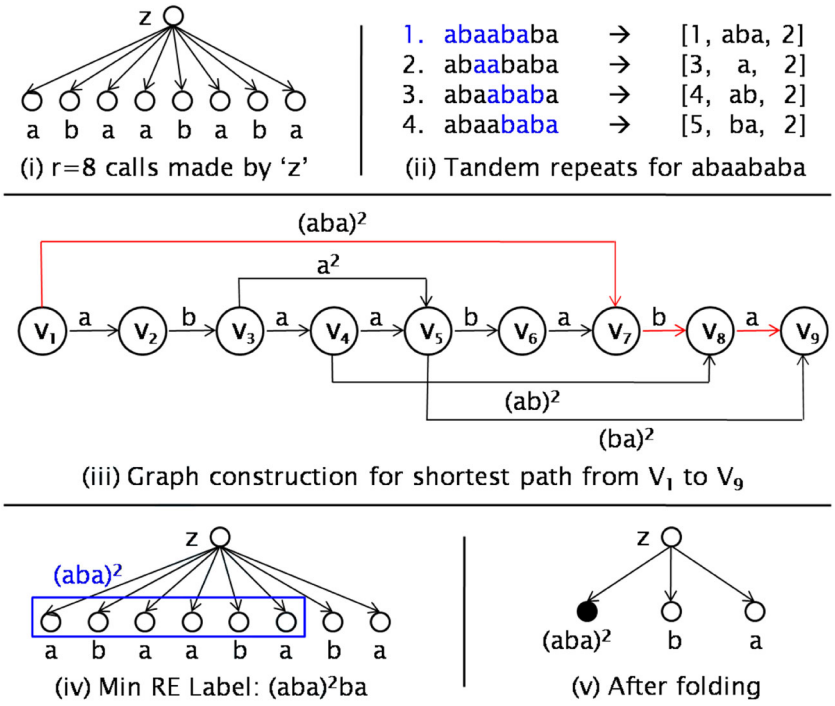


Figure 5. Minimal regular expression computation.

Concatenating the labels on the edges of the shortest path gives the minimal regular expression, $MinRE(S)$. The last step is to return the labels and edges along the shortest path, that is, triples of the form $[expr,\ i,\ j]$ where $i \neq j$ and i and j denote start and end positions of the sibling nodes that can be grouped. This step takes linear time. Thus, the time complexity of the algorithm to compute the minimal regular expression of a sequence of length r is $O\left(r^2\right)$.

Figure 5 illustrates the construction of a minimal regular expression for the sequence $S = abaababa$. Here, $r = 8$. For this sequence, three regular expressions are possible: $(aba)^2ba$, $aba(ab)^2a$, and $ab(a)^2(ba)^2$. The minimal regular expression is $(aba)^2ba$. To determine this answer, first the algorithm computes the primitive tandem repeats: $[1, aba, 2]$, $[3, a, 2]$, $[4, ab, 2]$, and $[5, ba, 2]$. A graph with nine vertices $\{V_1 \ldots V_9\}$ is then constructed by introducing an edge between every V_i and V_{i+1}, for $i = 1 \ldots 8$ and also an edge for each of the four primitive tandem repeats, (V_1, V_7), (V_3, V_5), (V_4, V_8), and (V_5, V_9), with edge labels $(aba)^2$, $(a)^2$, $(ab)^2$, and $(ba)^2$, respectively. Next, the shortest path from V_1 to V_9 is determined: $V_1 \rightarrow V_7 \rightarrow V_8 \rightarrow V_9$. The labels along this path are then concatenated to determine the minimal regular expression, $MinRE(S) = (aba)^2ba$. The $MinRE(S)$ consists of three terms, $(aba)^2$, b, and a, and hence, its length $= 3$.

In the best case, $MinRE$ returns a single term. For example, if $S_1 = abaabaaba$, then $MinRE(S_1) = (aba)^3$. In the worst case $MinRE(S) = S$ itself, for example, when $S_2 = abcde$. In terms of compaction (to be described in the next subsection), in the former case, the nine nodes for S_1 will be replaced by a single node in the compacted tree, with label $(aba)^3$; and, in the latter case, no compaction is possible. The sequence $S_3 = abababa$ has two minimal regular expressions, $(ab)^2a$ and $a(ba)^2$. The algorithm chooses one among the two shortest path arbitrarily and returns the corresponding regular expression. Because this expression has length 2, the seven nodes of S_3 will be replaced by two nodes in the compacted tree. If $MinRE(S_3)$ chooses $(ab)^2a$, the labels of the two compacted nodes will be $(ab)^2$ and a, respectively.

3.2. *Compaction*

The purpose of *compaction* is to fold the call tree so as to achieve maximal compaction. Compaction makes crucial use of the labeling procedure described in the previous subsection. We can apply this labeling process at all levels of the call tree either in a top-down or a bottom-up fashion. However,

DOI: 10.1002/spe

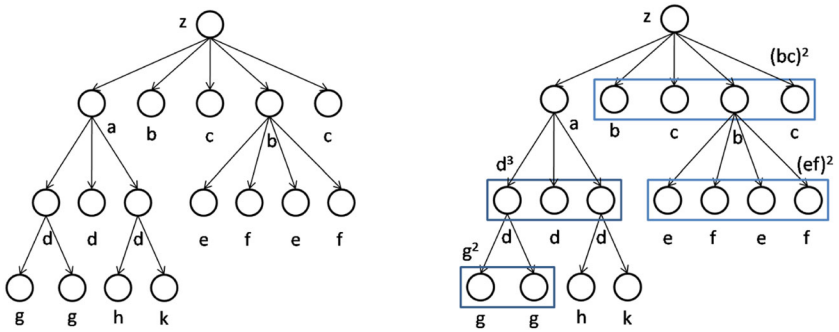


Figure 6. A call tree and its regular expression labels.

the bottom-up approach can be applied to both partial and complete call trees and hence is more suitable for use in an interactive execution environment. In the bottom-up approach, we start with the lowest level of the call tree and proceed in a depth-first left-to-right order identifying labels at each level, as illustrated in Figure 6. These labels correspond to tandem repeat sequences and serve as the basis for compaction.

When a node has a subtree below it, we need to compact the subtree recursively and also construct a compact encoding of the subtree. The encoding will enable us to later expand the subtree level-by-level, as needed. As noted earlier, the regular expression label is made of symbols (method names), sequencing, and definite iteration. The encoding is a Lisp-like expression enclosed within [and], as it provides a simple unambiguous encoding of a tree structure. We explain this encoding with an example. In Figure 6, the subtree with two g nodes rooted at d can be encoded as $[(d \;\; g^2)]$; the subtree rooted at b and with e, f, e, f below it can be encoded as $[(b \;\; (ef)^2)]$. The primitive compaction and encoding operations on a call tree are as follows:

1. A leaf node with method name m that is not part of a tandem repeat sequence is encoded as $[m]$.
2. A tandem repeat sequence of leaf nodes S is replaced by a new node with regular expression label $MinRE(S)$ and encoding $[MinRE(S)]$.
3. For a non-leaf node with method name m and subtree encodings $[e_1], \ldots, [e_n]$, the composite encoding for the node is $[(m\ e_1\ \ldots\ e_n)]$.
4. A tandem repeat sequence S of non-leaf nodes of length k is replaced by a single new node with label $MinRE(S)$. If the encodings for the nodes in S are $[e_1] \ldots [e_k]$, the composite encoding for S is $[e_1 \ldots e_k]$.

When the compaction and encoding operation are applied on a call tree, the resulting tree will have two types of nodes: ordinary nodes (from the original call tree) as well as a new type of node, the *compact node*, which is a leaf node with a regular expression label for the sequence of nodes that were compacted as well as an encoding of the subtrees that were present below this sequence of nodes. The compact nodes are shown in black in Figure 7.

A *maximally compacted tree* is the one that is obtained by repeated application of the compaction and encoding operations on a call tree in a bottom-up left-to-right order until no tandem repeat sequences are left in the tree. Figure 7 illustrates this process of obtaining a maximally compacted tree for the call tree of Figure 6. The four cases of compaction can also be seen from this figure. The encodings $[h]$ and $[k]$ in step 1 illustrate case (1); encodings $[g^2]$ and $[(ef)^2]$ at steps 1 and 3 illustrate case (2); the sub-encodings $[(d\ g^2)]$ and $[(d\ h\ k)]$ in step 2 illustrate case (3); and the encoding $[(d\ g^2)\ \ d\ \ (d\ h\ k)]$ in step 2 illustrates case (4). As Figure 7 clarifies, we do not compact nodes that are not part of a tandem repeat sequence. This is a design choice, but our approach also extends to compacting and encoding such nodes.

Algorithm complexity. We briefly discuss the complexity of obtaining the maximally compacted tree. Because compaction is applied at every internal (or non-leaf) node of the call tree, the cost of compaction varies with respect to the call tree structure. At each internal node, labeling is first applied and based on the minimal regular expression obtained, one or more sequence of nodes are

DOI: 10.1002/spe

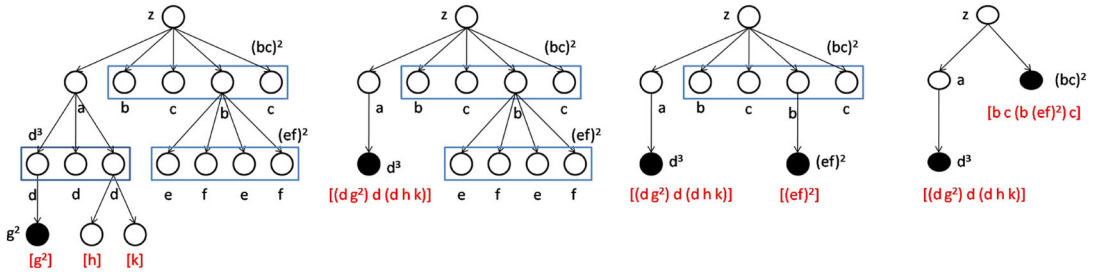


Figure 7. Step-by-step compaction of call tree of Figure 6. Black nodes are compacted nodes, and their encoding is shown in red below. The maximally compacted tree is shown at the end.

replaced by compacted node(s). (i) As discussed in Section 3.1, the labeling of r child nodes takes $O\left(r^2\right)$ time. (ii) Replacing a sequence of nodes by a single node takes linear time. At worst, there can be $r/2$ replacements when each adjacent pair of nodes is replaced by a single compacted node. This leads to removal of r nodes and insertion of $r/2$ nodes leading to $1.5r$ operations. Hence, compaction of a single node with r child nodes is bounded by $O\left(r^2\right)$.

Let N be the total number of nodes in the call tree. Let k be the number of internal nodes. Let r_i $(1 \leqslant i \leqslant k)$ denote the number of child nodes for each of the k internal nodes, respectively. Hence, $r_1 + \cdots + r_k \leqslant N$. That is, the total number of child nodes of the call tree cannot exceed N. Hence, the complexity of compacting the entire call tree is $O\left(r_1^2 + \cdots + r_k^2\right)$. Because $r_1^2 + \cdots + r_k^2 \leqslant (r_1 + \cdots + r_k)^2 = N^2$, the complexity of obtaining a maximally compacted tree is bounded by $O(N^2)$.

A maximally compacted tree can be expanded to an ordinary call tree by repeated application of the expansion and decoding operations in a top-down right-to-left order until no compact nodes remain. The expansion operation is simply the reverse of the process of constructing a maximally compacted subtree. Suppose an ordinary node n points to a compact node c with encoding E. Node c is removed along with the edge from n to c. There are two cases:

1. Case $E = [e_1 \ldots e_k]$. Create k ordinary nodes corresponding to $e_1 \ldots e_k$ and create directed edges from n to each such node. If any e_i is not a method name, it must be of the form $(m_i\ e_{i1} \ldots e_{ij})$; create a compact node with encoding $[e_{i1}, \ldots, e_{ij}]$ with m_i pointing to it.
2. Case $E = [RE]$. Expand RE into its tandem repeat sequence $S = m_1 \ldots m_k$. Create k ordinary nodes with labels $m_1 \ldots m_k$, respectively, and edges from node n to each of these nodes.

4. HORIZONTAL COMPACTION

There are three ways by which horizontal compaction occurs: (i) during vertical compaction; (ii) by grouping lifelines; and (iii) when classes are filtered out. The simplest case of horizontal compaction during vertical compaction arises when a method makes nested calls to methods in other objects. In rooted structures such as trees, vertical compaction generally results in horizontal compaction as well because all operations start at the root of the structure. For example, when n values are inserted into a binary search tree, each insertion starts with a call such as *root.insert(v)*. Each of these inserts will, in general, result in call to *left.insert(v)* or *right.insert(v)*. After compaction, the set of n insertions will be replaced by a single call $insert^n$ hiding the underlying traversals for each of n insert operations.

4.1. Grouping lifelines

Grouping can help in significant reduction of sequence diagram size when multiple objects of a class are present. This is best applicable when the objects of a class interact only with objects of other classes. Hence, choosing a representative object of each class is sufficient for conveying the intended effect, the benefit being that the horizontal spread of the sequence diagram is minimized.

DOI: 10.1002/spe

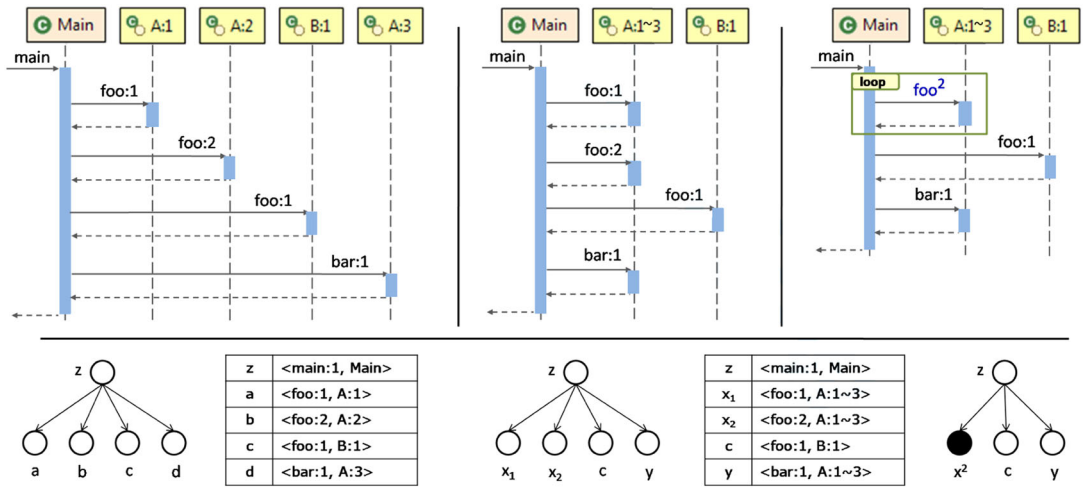


Figure 8. Horizontal compaction by grouping lifelines: the three lifelines A:1, A:2, and A:3 are replaced by a single lifeline A:1–3. This is followed by vertical compaction where two 'foo' calls are replaced by foo^2. The corresponding call trees are shown below.

The procedure for applying this kind of horizontal compaction is straightforward. Once the call tree is constructed and the number of object instances of each class has been noted, the call tree can be traversed, and individual object names of each node are replaced by a single object group name.

Figure 8 illustrates how this is performed. Here, the lifelines pertaining to three instances of class A, namely, A:1, A:2, and A:3, are replaced by a single representative lifeline A:1∼3, thus achieving horizontal compaction. This is followed by vertical compaction – note that the grouping of lifelines sets the stage for vertical compaction. If the order of compaction is reversed (i.e., first vertical, then horizontal), no vertical compaction can be achieved.

As an illustration of grouping, when initializing a graph consisting of nodes and edges, the constructor calls are executed in series, and each of the node and edge calls are invoked in different instances node and edge objects. By grouping all node objects into a single object and all edge objects into a single object, the constructor that calls in the vertical dimension can then be compacted. Another illustration of group is shown in Figure 2. Here, a sequence of 11 objects are inserted into a binary search tree, and the tree is displayed on the output panel after each insertion. In order to support different types of values, the inserted values were objects of classes such as `IntElem` and `StringElem`. The recurring pattern `(IntElem; insert; drawTree)`[11] that is detected actually involves method calls on three different types of objects, namely, `IntElem`, `Tree`, and `OutputPanel`.

Sometimes it is desirable not to perform horizontal compaction because we want to observe the difference in behavior of the individual objects of a class. In these cases, grouping of object lifelines is not performed. As Figure 1 illustrates, the choice of which compaction is to be applied is specified by the user.

4.2. *Exclusion filters*

Java Interactive Visualization Environment's *exclusion filters* are motivated by the observation that often users know a priori that certain parts of the code are trusted or irrelevant to a particular task. For example, in a debugging scenario, all library code may be considered trusted, whereas in a program comprehension scenario, the user may be interested only in the public interactions among objects. Regardless of users' motivation, this opens the possibility for a significant reduction in the diagrams that are to be displayed. JIVE supports a number of filtering options:

(i) *Type filters*. Users may exclude events originating in a type whose name matches a given regular expression, for example, java.* would filter out events from types defined in any package

DOI: 10.1002/spe

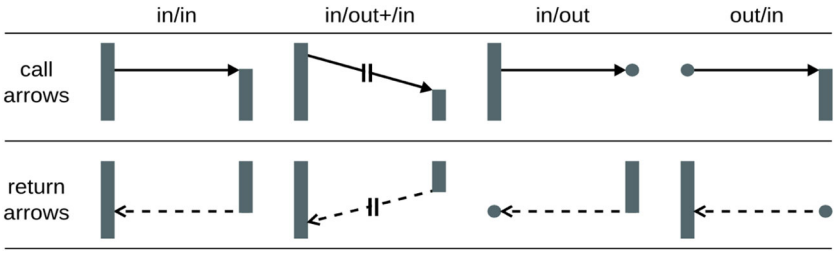


Figure 9. Notation for out-of-model calls.

starting with java. JIVE provides sensible default package filters for applications, applets, and unit tests.

(ii) *Method filters.* Similar to type filters, users may exclude behavior considered uninteresting such as getter and setter methods of a given class, by using filters such asMyClass.get* and MyClass.set*. While these filters effectively eliminate getter and setter calls made to instances of MyClass, JIVE preserves any side effects resulting from the execution of such methods.

(iii) *Visibility scope filters*. Object-oriented programming practice dictates that implementation details should be encapsulated in private (or protected) methods, while public APIs should be publicly visible. JIVE allows users to focus on interactions happening at any visibility scope. For instance, users trying to gain a high-level understanding of a software may choose to view only public interactions involving public classes, whereas users trying to debug the implementation of a particular class may choose to view all methods.

4.3. Out-of-model calls

An important consequence of filtering is that an execution trace can no longer be used to derive the full model necessary to render the object and sequence diagrams. Therefore, the diagrams must be adapted in order to handle missing information gracefully, without compromising the users' understanding of the program behavior as a whole or the interactions happening among specific objects. To address this problem, we introduce the notion of *out-of-model* entities as well as extensions to the sequence diagram to represent interactions between in-model and out-of-model entities (Figure 9).

Definition 4.1
A type (class) is out-of-model if it is filtered by an execution trace. Instances of an out-of-model type (class) are out-of-model. Any call to or return from a method of an out-of-model class or instance is out-of-model.

Any entity that is not out-of-model is necessarily in-model. Detecting out-of-model calls and returns is particularly important to reduce the amount of discontinuities in the sequence diagram. To this end, every time a method call or return notification is received from the JVM, JIVE determines whether the event originated/terminated in-model or out-of-model by inspecting the debuggees call stack and comparing it with a locally maintained copy. Once JIVE determines that a method call (return) originates in-model or out-of-model, it can render the correct arrow in the sequence diagram. If a method is called from an out-of-model caller, JIVE uses a found message arrow. If a method returns to an out-of-model caller, it uses a lost message arrow. Lost and found messages are defined as part of UMLs sequence diagrams. JIVE introduces variants of lost and found messages through its out-of-model call and return arrows, which are defined as follows.

1. If an in-model method call m_1 makes an out-of-model call and before this call returns to m_1, it results in zero or more out-of-model calls culminating in an in-model method call m_2, then an *out-of-model call arrow* is said to connect the activation box of m_1 to m_2. This call is referred to as an in/out$^+$/in call.
2. If an in-model call m_2 returns to an out-of-model caller that in turn makes zero or more additional out-of-model returns/calls culminating in a return to an in-model call m_1, then an

DOI: 10.1002/spe

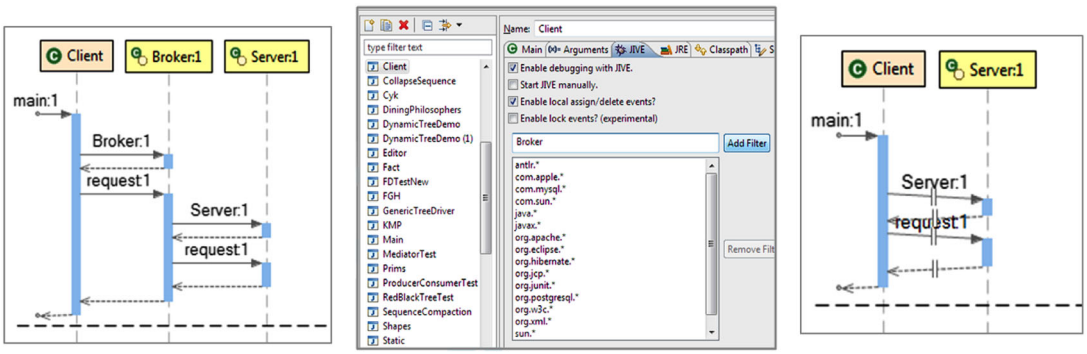


Figure 10. Out-of-model calls arising due to exclusion filters.

out-of-model return arrow is said to connect the activation box of m_2 to m_1. This return is referred to as an in/out$^+$/in return.

3. If an in-model method call makes an out-of-model call m and this call makes zero or more out-of-model calls but does not make any in-model call before returning back to m, then no out-of-model call arrow or return arrow is created for the call to and from m.

Figure 10 provides a simple illustration of the out-of-model call and return arrows. Figure 10(a) sketches a `Client` accessing a `Server` through an intermediary `Broker`. If we wish to filter out the `Broker` object, this can be specified through a JIVE interface as shown in Figure 10(b). Then the execution of the same program will involve an out-of-model call ('request') from the `Client` object to the `Broker` object followed by an in-model call to the `Server` constructor. This is depicted by an out-of-model call arrow from the `Client` object to the `Server` object. The return from the `Server` constructor first goes to the out-of-model `Broker` object and from there to the in-model `Client` object, and hence, an out-of-model return arrow depicts this transition. In a similar manner, out-of-model call and return arrows are used to depict the actual request from the `Client` object to the `Server` object.

5. MULTI-THREADED EXECUTION COMPACTION

5.1. *Multi-threaded execution*

In a multi-threaded environment, control may shift between threads in an unpredictable manner during the course of execution. Threads may access one or more objects in common and may do so in an exclusive or a non-exclusive manner. While the calls made by the threads may remain the same, the interleaving pattern may vary between two executions depending on the synchronization primitives used. Figures 11 and 12 portray sequence diagrams from four different executions of the classical producer–consumer problem. While Figure 12 depicts the two interleaving patterns that one might observe in practical settings for the aforementioned problem, Figure 11 presents scenarios that cover the remaining interleaving patterns possible in multi-threaded programs in general.

The figures depict three threads: `Main`, `Producer`, and `Consumer` in the program. The `Main` thread initializes the `Buffer` object(s) and starts the `Producer` and `Consumer` threads. Thereafter, `Producer` and `Consumer` invokes `write()` and `read()` methods of `Buffer` respectively twice and exit. Note that each thread is represented by separate color. Some observations are in order.

1. In all the scenarios, the call sequences by the three threads, when considered individually, remain the same, that is, `Main` instantiates `Buffer`, `Producer` executes two `write()` operations, and `Consumer` executes two `read()` operations.
2. The `write()` and `read()` calls made by the `Producer` and `Consumer` threads respectively are interleaved. Consequently, compaction of `write()` and `read()` calls cannot

DOI: 10.1002/spe

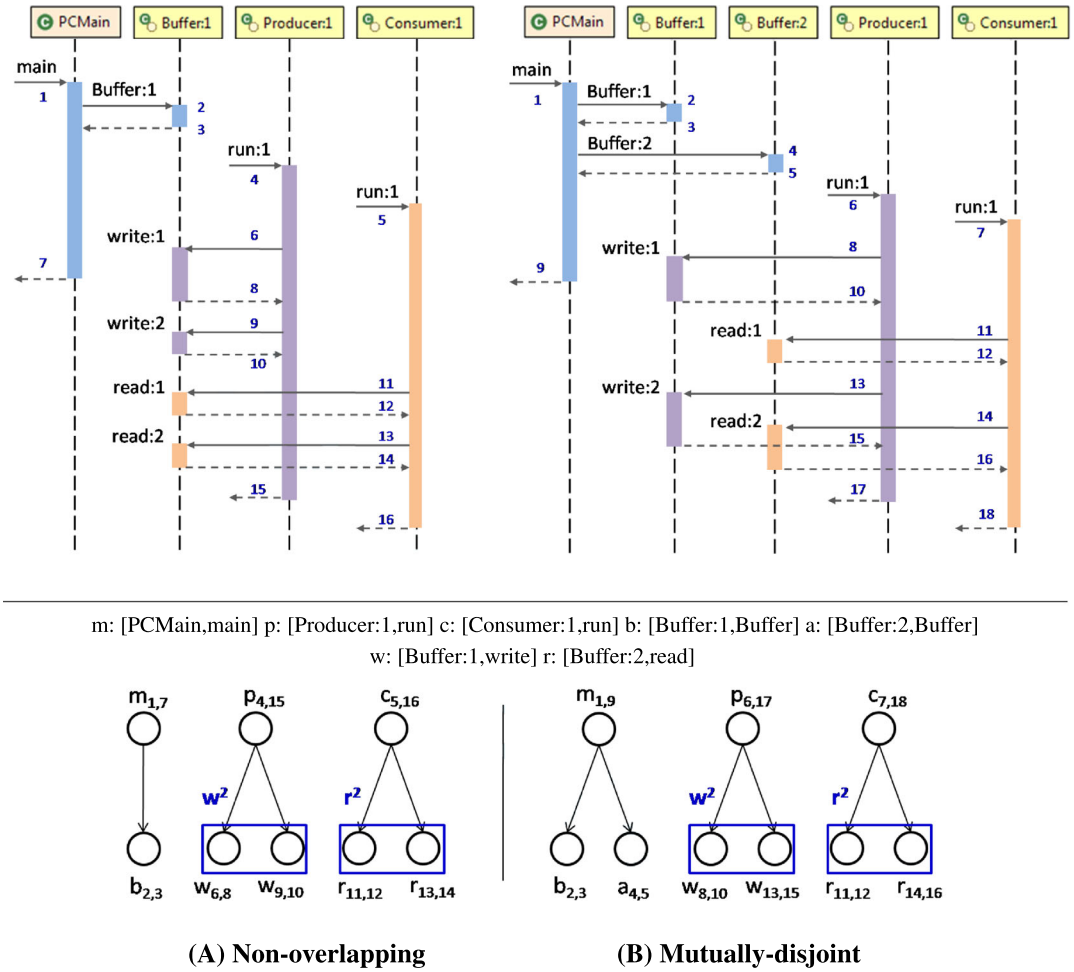


Figure 11. Sequence diagrams constructed from executions of two of the four possible implementations of the classical Producer–Consumer program and their respective call trees below: In scenario A (left) Producer and Consumer threads are non-overlapping. The Buffer is accessed one after the other. In scenario B (right) Producer and Consumer are mutually disjoint. Producer accesses Buffer:1 while Consumer accesses Buffer:2.

be performed separately. If we were to compact them independently, one after the other, say `write`^{2} followed by `read`^{2}, the compacted sequence diagram would convey that both `write()` calls happened before the `read()` calls which is true only for scenario A in Figure 11. Hence, we need a strategy to determine the compaction boundary and apply composite labeling.

3. The manner in which the `write()` and `read()` calls are interleaved are different. In the non-overlapping scenario (Figure 11(A)), calls from `Producer` and `Consumer` threads on the shared object `Buffer` do not overlap. In the mutually-disjoint scenario (Figure 11-B), the threads access their own independent buffers - `Buffer:1` and `Buffer:2`. In the mutually exclusive scenario (12-C), the threads access `Buffer` in an alternate fashion, and in the last scenario (12-D), they access `Buffer` simultaneously. This calls for extensions to labeling mechanism to convey differing semantics based on the interleaving pattern.
4. The non-overlapping scenario depicted in Figure 11(A) denotes the fact that there is no interleaving really between the `Producer` and `Consumer` threads. Compaction, in such a case, can be performed independently, and no special handling is required for this scenario.

Later, we discuss our strategy to systematically compact sequence diagrams obtained from execution of multi-threaded programs that address aforementioned interleaving behaviors.

DOI: 10.1002/spe

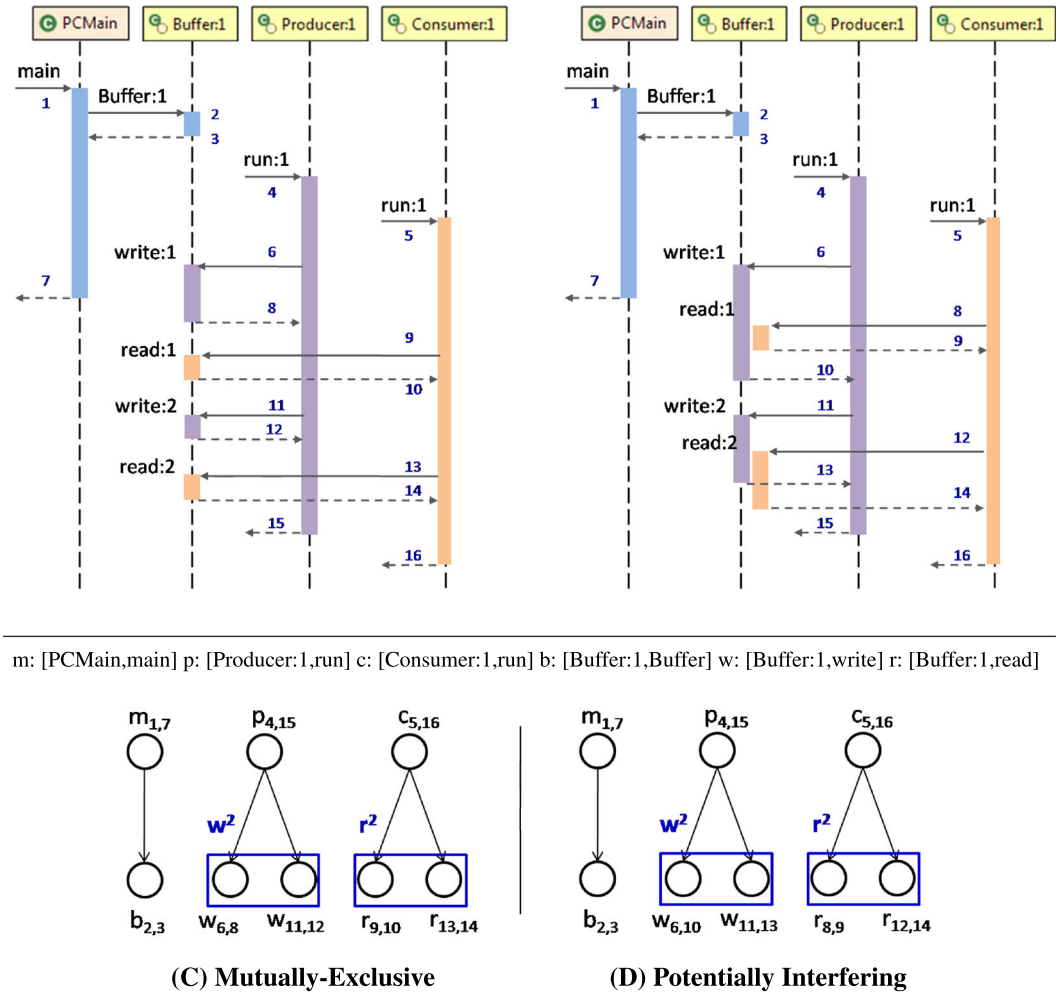


Figure 12. Sequence diagrams constructed from executions of two remaining implementations of classical Producer–Consumer program and their respective call trees below: In scenario C (left) Producer and Consumer threads are mutually exclusive. The Buffer is accessed in a mutually exclusive manner. In scenario D (right), Producer and Consumer are potentially interfering. The Buffer is accessed in a simultaneous manner by both the threads.

5.2. *Compaction boundaries*

In a multi-threaded program, a call tree is constructed for each thread because each method call/return event includes the thread on which this event occurred. Thus, we augment each node of the call tree with the thread ID on which the call occurred. Figures 11 and 12 show sequence diagrams for two multi-threaded executions along with the respective call trees for these executions.

Definition 5.1
Every node in a multi-threaded call tree represents a method call $\langle m : i, o, t, s, e \rangle$, where method call *m:i* on object *o* in thread *t* has *s* and *e* as the call (start) and return (end) event numbers, respectively.

Once the call trees are constructed, the labeling of each tree is performed separately as per the Algorithm 1. The labels for the two `read()` and `write()` operations in Figures 11 and 12 are r^2 and w^2, respectively. Next, we consider each label from every call tree to determine if the compaction can be performed independently. If a set of labels are overlapping, we also need to determine the type of interleaving between the labels. Through a composite-labeling scheme, the compacted sequence diagrams can then portray differing thread-interleaving semantics. Finally, the encoding

DOI: 10.1002/spe

scheme should allow us to reconstruct the original sequence diagram back. The event numbers will allow us to accomplish these three goals.

Definition 5.2
The *span* of a method call $\langle m : i, o, t, s, e\rangle$ is $\langle s, e\rangle$. For a sequence of method calls $\langle m_1 : i_1, o_1, t, s_1, e_1\rangle \ldots \langle m_k : i_k, o_k, t, s_k, e_k\rangle$ on the same thread t, the label $L = m_1 \ldots m_k$ and the span of L is $\langle s_1, e_k\rangle$.

For example, in scenarios depicted by Figures 11(A), 11(B), 12(C) and 12(D), the spans of label w^2 are $\langle 6, 10\rangle$, $\langle 8, 15\rangle$, $\langle 6, 12\rangle$, and $\langle 6, 13\rangle$, respectively. In a similar manner, the *span* of r^2 in the same set of scenarios are $\langle 11, 14\rangle$, $\langle 11, 16\rangle$, $\langle 9, 14\rangle$, and $\langle 8, 14\rangle$, respectively.

Definition 5.3
Given two labels L_1 and L_2 with spans $\langle s_1, e_1\rangle$ and $\langle s_2, e_2\rangle$, respectively, we say that L_1 and L_2 are *overlapping* if $s_1 < s_2 < e_1$ or $s_2 < s_1 < e_2$.

The aforementioned definition can be generalized in a straightforward way to labels $L_1, \ldots, L_m$ from *different call trees* having spans $\langle s_1, e_1\rangle \ldots \langle s_m, e_m\rangle$, respectively, with $s_1 < s_2 < \ldots < s_m$.

Of course, overlapping labels denote concurrent execution. In scenario A, depicted in Figure 11, we observe that the labels w^2 and r^2 are not overlapping because the start event of w^2's span is not within r^2's span and the start event of r^2's span is not within w^2's span. In the scenarios depicted by Figures 11(B), 12(C), and 12(D), we note that w^2 and r^2 are overlapping because the start event of r^2's span is within the span of w^2.

Definition 5.4
Let $L_1, \ldots, L_m$ be a set of overlapping labels with spans $\langle s_1, e_1\rangle \ldots \langle s_m, e_m\rangle$, respectively. The *compaction boundary* for the labels is $\langle b_1, b_2\rangle$ where $b_1 = min(s_1, s_2, \ldots, s_m)$ and $b_2 = max(e_1, e_2, \ldots, e_m)$.

That is, when there are more than two overlapping labels, the compaction boundary is determined as the span between the labeled node with the lowest start event number and the labeled node with highest return event number. When a label does not overlap with another label, its compaction boundary is the same as its span. In scenario A, there are two non-overlapping compaction regions with labels w^2 and r^2. The compaction boundaries for scenarios in Figures 11(B), 12(C), and 12(D) are $\langle 8, 16\rangle$, $\langle 6, 14\rangle$, and $\langle 6, 14\rangle$, respectively.

The encoding strategy remains almost the same except that the start and event numbers are now part of the encoding scheme. The event numbers are essential in order to correctly reconstruct the sequence diagram back during expansion. The compacted call trees, the spans of the labels, and the compaction boundaries for all the four scenarios are given in Figure 13. The encoding of the compacted nodes are shown in red.

5.3. Compaction labels

The concurrent behavior is depicted by enclosing the calls denoted by the overlapping labels within |...|. The type of overlapping, whether mutually disjoint, mutually exclusive, or potentially interfering, is depicted using the notations $\otimes$, $\oplus$, and $\bowtie$, respectively. Because the sequence diagram captures history of execution in terms of method invocations, we distinguish their overlapping behavior at the method interaction level.

1. Two overlapping labels L_1 and L_2 are said to be *mutually disjoint* if, for every method call $\langle m_1 : i_1, o_1, t_1, s_1, e_1\rangle$ in L_1 and every method call $\langle m_2 : i_2, o_2, t_2, s_2, e_2\rangle$ in L_2, we have $o_1 \neq o_2$. Scenario B in Figure 11 illustrates the mutually disjoint case. Here, we note that *write()* method is invoked on *Buffer:1* object by *Producer* thread, while the *read()* method is invoked on *Buffer:2* object by the *Consumer* thread.
2. Two overlapping labels L_1 and L_2 are said to be *mutually exclusive* if, for every method call $\langle m_1 : i_1, o_1, t_1, s_1, e_1\rangle$ in L_1 and every method call $\langle m_2 : i_2, o_2, t_2, s_2, e_2\rangle$ in L_2, we have $o_1 \neq o_2$ or $s_1 > e_2$ or $s_2 > e_1$. Scenario C in Figure 12 illustrates mutually exclusive

DOI: 10.1002/spe

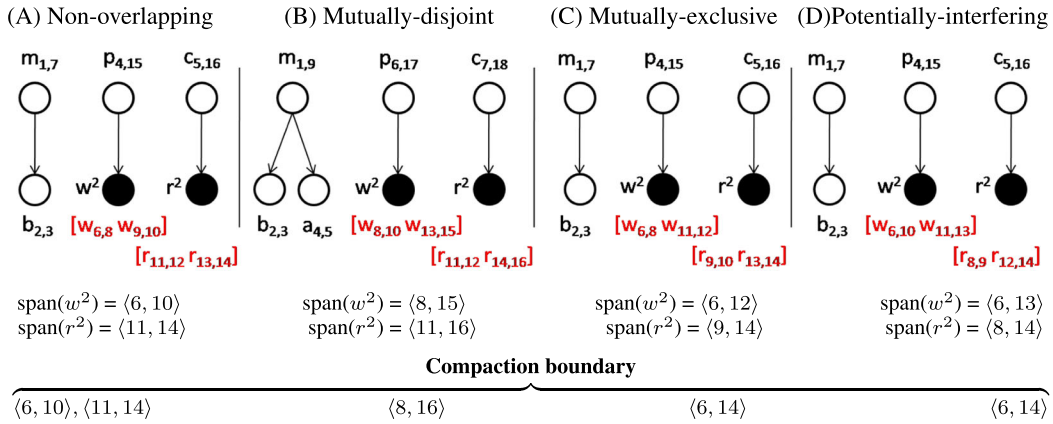


Figure 13. Compacted tree versions for the four scenarios with encoded sequences in red. Span of labels w^2 and r^2 and the compaction boundaries are shown below.

execution of *read()* and *write()* operations. Because the sequence diagram is concerned with method calls, the criterion for mutual exclusion is that at most one method invocation is active on an object. We assume that the fields of the shared object are not made public, as this is a reasonable assumption for safety in concurrent applications.

3. Two overlapping labels L_1 and L_2 are said to be *potentially interfering* if there is some method call $\langle m_1 : i_1, o_1, t_1, s_1, e_1\rangle$ in L_1 and some method call $\langle m_2 : i_2, o_2, t_2, s_2, e_2\rangle$ in L_2, such that $o_1 = o_2$ and ($s_1 < s_2 < e_1$ or $s_2 < s_1 < e_2$). Scenario D in Figure 12 shows that both *write* and *read* calls are simultaneously active on the *Buffer* object. Note that we only term it as *potentially interfering* because, at the method level, we do not know the underlying operation performed by both of these methods. It is possible that they manipulate the fields simultaneously. Also, from the application point of view, simultaneous access of the fields may be deemed as acceptable.

In the presence of more than two overlapping labels, multiple types of interleaving are possible between every pair of threads. The representation will be addressed as per the following definition.

Definition 5.5
Let $L_1, \ldots, L_m$ denote a set of overlapping labels. Then the composite label for the compaction region is deduced in the following order.

1. If there exists L_i and L_j ($i \neq j$) that are *potentially interfering*, then $L_1 \bowtie L_2 \bowtie \ldots \bowtie L_m$.
2. Else if all L_i and L_j ($i \neq j$) are *mutually exclusive*, then $L_1 \oplus L_2 \oplus \ldots \oplus L_m$.
3. Else $L_1 \otimes L_2 \otimes \ldots \otimes L_m$.

The portions of compacted sequence diagrams with the labeling is shown in Figure 14. In summary, scenarios A, B, and C indicate different cases of non-interfering concurrent behavior, while the scenario D indicates potentially interfering concurrent behavior. Algorithm 2 provides the steps involved in producing compact sequence diagram for the multi-threaded case.

Algorithm complexity. To compute the time complexity of this algorithm, let N denote the total number of nodes in all the call trees. Construction of the call trees (Step 1) takes $O(N)$ time. Labeling of the call trees (Step 2) takes $O(N^2)$ time as seen in Section 3.2. The fact that we have multiple call trees, one for each thread, does not affect the complexity of labeling because the number of nodes in all the call trees put together cannot exceed N.

Time complexity of Step 3 is determined by the number of labels. The maximum number of labels for the call trees consisting of N nodes cannot exceed $N/2$. This happens when every pair of nodes is labeled throughout. Hence, the loop is executed at most $N/2$ times. Determining the span of a label L (Step 3a) takes $O(1)$ time considering that we store this information during the labeling step. Steps 3b, 3c, 3d, and 3e can be performed together. In the worst case, we will end up examining all

DOI: 10.1002/spe

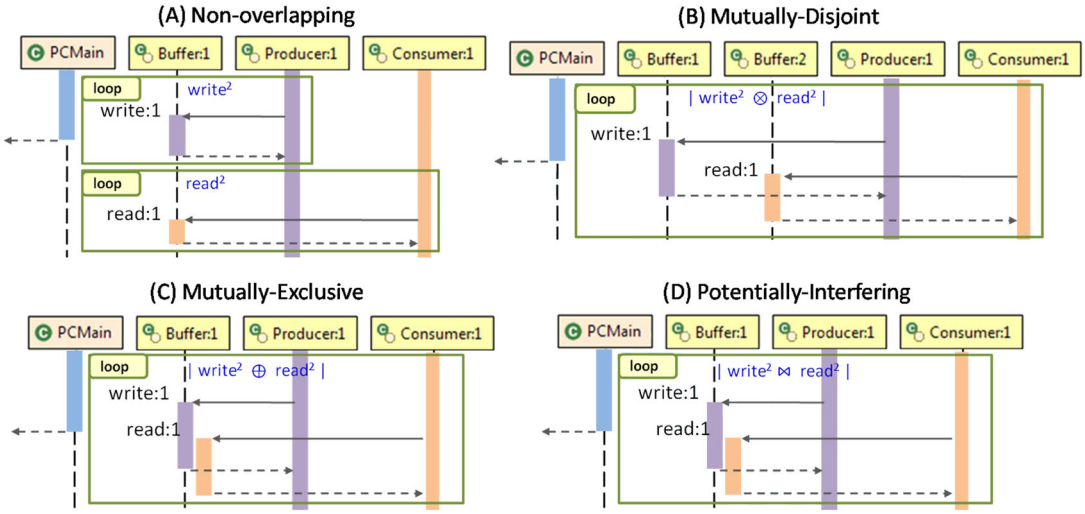


Figure 14. Portions of compacted sequence diagrams for the four scenarios.

ALGORITHM 2: Multi-threaded labels

Input: Execution call sequence of multi-threaded program with 1…T threads
Output: Compacted Call Trees
1. Construct call trees $C_1, C_2, \ldots, C_T$ assigning call and return event numbers to each node;
2. Apply labeling to each call tree using Algorithm 1;
3. **for** *(each label L not yet marked as 'compacted')* **do**
   a. Determine the span of L;
   b. Determine all labels $L_1, L_2, \ldots, L_m$ that overlap with L;
   c. Compute their respective compaction boundaries;
   d. Determine overlapping type of $L, L_1, L_2, \ldots, L_m$;
   e. Compact $L, L_1, L_2, \ldots, L_m$;
   f. Mark $L, L_1, L_2, \ldots, L_m$ as 'compacted';

other labels to compute the compaction boundaries, determine the overlapping type, and then doing the compaction. Hence, it takes $O(N/2)$ time for $N/2$ labels. Marking the labels (Step 3f) takes linear time, that is, $O(N/2)$. Hence, Step 3 takes $O(N^2)$ time. Hence, the overall time complexity works out to $O(N^2)$ for Algorithm 2.

The compacted sequence diagram can be correctly derived from the call trees by using the event numbers. We decipher the method calls and returns in the increasing order of the event numbers. Each call tree is traversed in a depth-first manner. Whenever there is a break in the event number, we jump to the call tree with the next event number and continue the traversal. The concurrent loops are merged into a single loop with regular expression labels assigned as per their overlapping behavior given in Definition 5.5. Once all the nodes in all the call trees are covered, we have the complete sequence diagram.

6. EXPERIMENTAL RESULTS

We present experimental results to show the benefit of sequence diagram compaction for a variety of programs. These experiments have been run under JIVE extended with the sequence diagram compaction plug-in, which is downloadable from http://www.cse.buffalo.edu/jive/latest.html. The compaction plug-in requires as input an execution trace, that is, a sequence of execution events. Thus, a Java program is first executed under JIVE, its execution trace is saved (as a .csv file), and this file is given as input to the compaction plug-in. JIVE does not require access to Java source code in order to construct runtime visualizations; it suffices to provide just the class files. The compaction

DOI: 10.1002/spe

plug-in allows the user to choose the type of compaction to be applied: vertical, horizontal, or both. It also allows the user to specify a region of interest through start and end event numbers, and focus on a specific part of the execution.

Before presenting our experimental results, we note some general properties of all execution traces. Three types compaction scenarios recurred in our experiments:

1. *Pure vertical compaction.* This scenario is the result of the execution of iterative loops in the program. The size of the resulting regular expression depends on the length of the sequence, the number of distinct symbols (method calls) in it, and the actual sequence of the symbols. Typically, sequences with fewer symbols are amenable to more compact labels.
2. *Pure horizontal compaction.* Horizontal compaction is applicable when there are multiple objects of the same class; grouping their lifelines can provide significant reduction of sequence diagram size. When object-specific interactions need to be visualized, as is usually the case in multi-threaded execution, it is desirable to omit horizontal compaction. Hence, the compaction plug-in allows the user to determine when a particular compaction should be applied.
3. *Hybrid vertical-and-horizontal compaction*: Vertical compaction often results in horizontal compaction when a call to an object results in nested calls to the other objects of the same or different class. We have noted that this form of compaction is very common in tree operations, such as insertion into a binary search tree. And, as illustrated in Section 4, horizontal compaction by grouping lifelines can enable vertical compaction in the next stage.

6.1. *Efficacy of compaction*

Tables I and II summarize the results from our experiments in compacting sequence diagrams for a variety of programs and execution trace sizes based upon the techniques described in this paper. These programs cover typical algorithms and data structures that are studied in computer science curricula, such as binary search trees, Adelson-Velsky and Evgenii Landi's (AVL) trees, and Dijkstra's shortest path. They include programs that exhibit with multiple threads, such as producer–consumer interactions and dining philosophers. And, finally, a couple of larger programs, including

Table I. Experimental results: benefit of sequence diagram compaction.

S.No.	Program	# Events	Interactions before	Interactions after	Percentage of compaction
1	Recursion	196	(17, 1)	(17, 1)	(0%, 0%)
2	ProducerConsumer	510	(26, 4)	(8, 4)	(69%, 0%)
3	TwoProcessMutex	667	(52, 8)	(14, 8)	(73%, 0%)
4	Delegation	762	(83, 13)	(83, 11)	(0%, 15%)
5	ExternalIterator	1,244	(93, 17)	(12, 5)	(87%, 70%)
6	PrimsMST	1,367	(20, 2)	(8, 2)	(60%, 0%)
7	TreeGUI	2,698	(280, 21)	(8, 8)	(97%, 62%)
8	AVLTree	3,632	(292, 14)	(5, 2)	(98%, 85%)
9	Trie	4,656	(583, 76)	(8, 3)	(60%, 0%)
10	DiningPhilosopher	5,310	(302, 20)	(56, 17)	(98%, 96%)
11	MinRegExp	11,162	(499, 92)	(14, 6)	(97%, 93%)
12	Parser(Recursive)	20,254	(1,075, 563)	(753, 16)	(29%, 97%)
13	BST	34,398	(2,547, 98)	(3, 2)	(99%, 97%)
14	Parser(Iterative)	59,461	(3,111, 1591)	(10, 8)	(99%, 99%)
15	Plugin(PrdcrCnsmr)	114,469	(7,216, 1481)	(34, 10)	(99%, 99%)
16	Plugin(Dijkstra)	197,268	(12,647, 143)	(47, 11)	(99%, 99%)
17	SpecJVMDerby	536,822	(5,052, 89)	(421, 29)	(91%, 67%)
18	SpecJVMCompiler	539,106	(43,807, 2279)	(642, 35)	(98%, 98%)
19	SpecJVMCompress	654,802	(5,882, 177)	(626, 33)	(89%, 83%)
20	Plugin(TreeGeneric)	1,476,604	(102,386, 613)	(38, 10)	(99%, 99%)

The size of a sequence diagram is given by a pair (m, n), where m is the number of method calls and n is the number of objects.

DOI: 10.1002/spe

Table II. Vertical compaction: regular expression labels.

S.No.	Program	MinRE labels	Label expansion
1	ProdConsSync	$(a^i \oplus b^i)$	a=put, b=get
2	ProdConsUnsync	$(a^i \bowtie b^i)$	a=put, b=get
3	TwoProcessMutex	$(ab)^i \otimes (ab)^i$	a=getLock, b=unLock
4	ExternalIterator	$a^i, (bbcc)^j$	a=insert, b=done, c=next
5	TreeGUI	$(abc)^i$	a=IntElem, b=insert, c=drawTree
6	PrimMST	$(abc)^i$	a=getMinKeyVertex, b=evaluate, c=getCount
7	Kruskal	$(ab)^i, (cd)^j, e^k$	a=read, b=addEdge, c=union, d=find, e=print
8	AVLTree	$a^i, (bc)^j, d^k$	a=insert, b=search, c=inorder, d=delete
9	Trie	$(abc)^i$	a=isEmpty, b=readString, c=put
10	DiningPhilospher	$(abc)^i \otimes (abc)^j \otimes (abc)^k \otimes (abc)^m \otimes (abc)^n$	a=hungry, b=pick, c=drop
11	MinRegExp	a^i, b^j, c^k, d^m	a=findMinRE, b=Vertex, c=Edge, d=compareTo
12	Parser (Recursive)	$a^i, b^j, (cb)^k, (bd)^m$	a=getChar, b=lex, c=Expr, d=idList
13	BST	a^i	a=insert
14	Parser (Iterative)	a^i, b^j	a=Stmt, b=add
15	Comp(PrdcrCnsmr)	$(ab)^i, c^j, d^k, e^m, f^n$	a=readEvent, b=parseEvent, c=compact,
16	Comp(Dijkstra)	$(ab)^i, c^j, d^k, e^m, f^n, g^p$	d=groupLifelines, e=dfsC,
17	Comp(TreeGeneric)	$(ab)^i, c^j, d^k, e^m, f^n, g^p$	f=printInteractions, g=MinimalRegex

the SpecJVM family of programs and the Sequence Diagram Compaction Plug-in itself. A more detailed explanation of each program is provided in the Appendix, and the results of running these programs is available at the website: http://www.cse.buffalo.edu/jive/compaction.

In Table I, column 2 indicates the number of events in the execution trace, where an execution event is as defined in Section 2 and could be a variable write, method call/return, object creation, thread start/end, and so on. Columns 3 and 4 show the number of interactions before and after compaction. The number of interactions is measured by a pair (n_m, n_o), where n_m is the number of method calls and n_o is the number of object lifelines present in the sequence diagram. Finally, column 5 indicates the extent of compaction both with respect to the number of calls and the number of objects. We compute the pair of ratios, $((n1_m - n2_m)/n1_m, (n1_o - n2_o)/n1_o)$, where $n1$ and $n2$ are the numbers before and after compaction, respectively. These ratios are expressed as percentages in Column 5.

In Table II, column 2 shows the form of the minimal regular expression labels computed for the execution traces shown in Table I. For example, in row 5, for the TreeGUI program, the regular expression $(abc)^i$, with a=IntElem, b=insert, c=drawTree, indicates that a sequence of three calls, on the IntElem constructor, the insert method, and the drawTree method was repeated a certain number of times. The form of the regular expression remains the same for all input data, while the repetition count (the superscript i) varies with input data. When more than one label is shown in column 2, for example, $(ab)^i$, $(cd)^j$, and e^k for row 7, Kruskal, it means that these labels occurred at one or more places in the compacted diagram.

Our experiments clearly demonstrate the benefits of compaction, especially that vertical compaction with regular expression labeling is a scalable technique. We discuss our experimental results for three types of execution traces: *small execution traces*, where the number of events is less than 10000; *medium execution traces*, where the number of events is between 10,000 and 100,000; and *large execution traces*, where there are over 100,000 events. It should be noted that the sequence diagram is determined by the number and structure of method calls, rather than the events *per se*. Typically, the number of events in our test programs is 10 to 20 times the number of method calls.

The small execution traces in our experiments result from running programs that implement the classic algorithms and data structures found in a Computer Science (CS) curriculum. A characteristic of these programs is that they are driven by few inputs. Varying the input size generally does not change the compacted diagram in a significant way; mainly the superscripts

DOI: 10.1002/spe

of the regular expression labels change. While the sequence diagrams of medium and large execution traces are increasingly varied in their respective structures, they are amenable to vertical compaction with regular expression labels in many sections of the diagrams. In applying our compaction methodology to the execution of the compaction plug-in on an event trace from Dijkstra's algorithm for shortest path (with a total of 197,268 events and 12,647 method calls), we initially observed that the event-processing loop could not be compacted with any simple pattern. However, after encapsulating the branching within the loop in a method, we were able to achieve dramatic compaction of the form: `(readEvent; parseEvent)`^{2165} where 2165 denoted the number of events in the event trace of Dijkstra's algorithm.

In general, the degree of compaction for a loop depends upon the extent of internal branching. For example, consider the following program structure:

while (expr1) { m1(); if (expr2) m2() else m3(); m4() }

Because there is no simple pattern in the values of the Boolean expression *expr2*, we can obtain a better compaction if we encapsulate the if-then-else in a method, say, *m23*, whose body is {*if (expr2) m2() else m3();* }. This would give a regular expression of the form *(m1;m23;m4)*^{n}, for n iterations of the while-loop. Unfortunately, this transformation cannot be performed automatically by our compaction technique because it works on an execution trace and does not have access to the source code. Thus, the transformation needs to be performed by the programmer on the source code to have the benefit of producing more compact visualizations at execution time.

Horizontal compaction is also very effective in minimizing white space in sequence diagrams arising due to 'block diagonal' pattern of method interactions. This was particular noticeable in the medium to large execution traces, such as the Parsing and SpecJVM applications. In a specific run of the parsing application (with a total of 59,461 events), the input program to be parsed had a sequence of 72 statements of a few different kinds (assignments, loops, conditionals, etc.) and resulted in 54 blocks of method interactions, each having nothing to do with the other. Horizontal compaction was very effective reducing the horizontal spread of the sequence diagram. When we further replaced the top-level recursive structure (for statement sequence) by an iterative loop, we were able to obtain dramatic vertical compaction as well.

6.2. Large execution traces and the SpecJVM benchmarks

Large execution traces typically result from a high degree of iteration and therefore lend themselves well to a compact representation. Large execution traces that arise from small programs lend themselves to an even greater degree of compaction than large execution traces from larger programs. When the sequence-diagram plug-in's execution was compacted, 267 object lifelines of CNode and 244 object lifelines of MinimalRegex could be merged, leading to over 95% horizontal compaction. A similar phenomenon was noticed during the execution of the SpecJVM 2008 benchmarks tests, which we discuss in more detail in the succeeding text.

The SpecJVM benchmarks are designed to capture the computations of typical general-purpose applications. They are a suite of real-life applications that help measure the performance of a Java Runtime Environment, focusing on the performance of the hardware processor and memory subsystem. The benchmarks have minimal dependence on file I/O and do not exercise any network communication between machines.

Our experiments considered three representative SpecJVM applications: compiler, compress, and Derby. These applications involve a large number of execution events, in the range 536,000–655,000 events. Table I, lines 17–19, summarizes the number of events, method interactions, and extent of compaction achieved.

The execution of a SpecJVM benchmark involves two phases: the test harness phase and the application execution phase. The test harness phase is common to all applications. In the three applications that we ran, the behavior of the test harness phase was fairly uniform, and some of the regular-expression labels found when compacting this phase were of the form

$a^i, (ab)^j, c^k, (de)^m, e^n, (fgh)^p, s^q$, and $(wxyz)^r$

DOI: 10.1002/spe

Table III. SpecJVM benchmarks: number of different regular expressions (REs).

Program	Total #REs	#REs in test harness	#REs in application
SpecJVM-Derby	106	35	71
SpecJVM-Compiler	180	42	138
SpecJVM-Compress	239	35	204

where a=setupProp, b=getFixedOperationsProp, c=getPropHelper, d=getSpecBasePath, e=getOut, f=getProp, g=getDefaultProp, h=validateProp, s=BenchmarkResult, w=setNumberBMThreads, x=getTimeProp, y=setWarmUpTime, and z=getBoolProp. As Table III shows, many more regular expression labels are discovered in the test harness phase.

The SpecJVM benchmarks exhibit several opportunities for horizontal and vertical compaction. As with other programs, large execution traces are the result of loops, which are quite varied in nature for these benchmarks. They range from simple REs of the form a^i where a is a single operation, such write, getOut, getBenchmarkForcegc, etc., to more complex expressions of the form $(a;b;c;d;e)^n$, where a=getOut, b=constraintAt, c=d=write, and e=size. And, as noted earlier, the sequence diagram for the SpecJVM benchmarks were amenable to considerable degree of horizontal compaction. In the SpecJVM-compiler application benchmark, 1977 object lifelines of SpecFileManager$CachedFileObject could be compacted to one object lifeline, leading to 98% compaction.

6.3. Time–space cost of compaction

The time for compaction can be broken down into four parts: the time for event processing, the time for call tree construction, the time for labeling, and the time for constructing the compacted call tree. We consider each of these as follows:

1. Event processing involves a simple case analysis on each event to determine whether it is a method call, method return, a thread start, or thread end event. These are the four types of events that need to be subsequently considered. Thus, the time for event processing grows linearly with the number of events.
2. Call tree construction as well as the time for constructing a compacted call tree grow linearly with the number method calls, because each node of the call tree corresponds to a method call.
3. The time for computing one regular expression label in a call tree grows quadratically with the size of the sequence of symbols (method calls) for which a label needs to be determined. The time complexity for this analysis was discussed in detail in Section 3.1. In general, more than one label may need to be computed for a call tree, because labeling and compaction occur at multiple levels of the call tree. However, any given label in general may span a subset of nodes of the call tree.

While the running time generally grows with the size of the execution trace, it more closely depends upon the number of method calls. Therefore, Figure 15 shows the time for computing the compacted representation of a sequence diagram as a function of the number of method calls. The running times shown in milliseconds are for a very modestly configured machine – an Intel 2.66 GHz processor with 2-GB RAM. The stack and heap sizes were 16 and 512 MB, respectively. As the analysis in Section 3.2 shows, the compaction time is dependent not just on the number of method calls, it is more intimately connected with the structure of the call tree and the pattern of method calls. As the structural properties of trees and patterns of method calls are hard to quantify, the running time in Figure 15 is shown as a function of the number of method calls.

Figure 15 shows the growth in compaction time for the 20 benchmark programs in Table II. This running time is the summation of a quadratic cost in labeling together with a linear cost for event processing and tree construction. In our experimental benchmarks, we noted that each sequence

DOI: 10.1002/spe

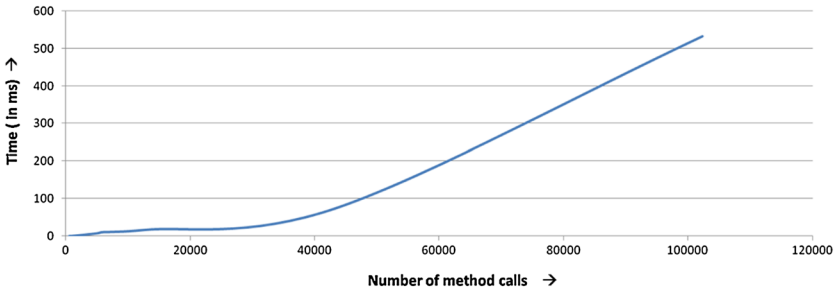


Figure 15. Method calls versus compaction time.

of symbols that is compacted typically has a length that is much smaller than the total number of method calls. For example, in the SpecJVM-Compiler benchmark, the number of method calls was 43,807, but the maximum superscript for a label (the number of iterations) was 766, with most of the superscripts in the range 25–250. The worst case labeling time, of $O(n^2)$, will be for a sequence of method calls of length n on the same function, whereas the best case labeling time, of $O(n)$, will be for a sequence of length n of calls to distinct methods. Practical programs fall somewhere in between these two extremes, and our experiments with the 20 programs in Table II showed that

1. the compaction time for small execution traces (50 to 500 method calls) was in the range 100 to 500 ms;
2. the compaction time for medium execution traces (with 500 to 5000 method calls) grew linearly over the range 0.5 to 8 s; and
3. the compaction time for large execution traces up to 500,000 events (with 5000 to 50,000 method calls) grew linearly up to 100 s.
4. the compaction time for the largest execution traces took between 100 to 500 s. These traces were generated by the Spec JVM Compiler and the Sequence Diagram Plugin with events in the range from 500,000 to 1.5 million (and with 50,000 to 100,000 method calls).

The space requirements for compaction consists of two parts: (i) The space requirements for event processing and tree construction are linear in the number of events and nodes (method calls) of the tree respectively. (ii) The space requirements for labeling is quadratic in the number of symbols to be compacted - the best case being linear and the worst case being quadratic. For our benchmarks, the space requirements were not excessive, except when labeling worst-case sequence of symbols with over 100,000 method calls.

7. COMPARISON WITH RELATED WORK

As noted earlier, there has been considerable interest in the study of sequence diagrams. Two recent surveys explore, respectively, tools for understanding sequence diagrams [15] and program comprehension through dynamic analysis [16]. Sneiderman [17] provides a visual information-seeking mantra for designing advanced graphical user interface.

- *Zoom.* One of the simplest techniques to view sequence diagrams in a smaller display area is to use the zooming out feature [16]. However, shrinking large sequence diagrams to fit smaller area affects the visibility and understandability of the diagram. It should be noted that JIVE already has this zooming feature from its earliest version.
- *Collapse.* Myers *et al.* and Taniguchi [6, 7, 18, 19] propose techniques to collapse recursive calls and iterative loops by using the source code or debug information. In the absence of such information, compaction cannot be performed. Our algorithm can compact the sequence diagram by recognizing the repeated patterns from runtime events itself and does not need the source code. Also, our technique can be used to perform compaction while program is running and not just restricted to performing offline compaction.
- *Remove.* Reduction in sequence diagram size can be achieved by removing less-relevant interactions such as common utility calls [6, 8, 10, 11], constructors, getters/setters [11, 15], local

DOI: 10.1002/spe

interactions within objects [13] and removing specific interactions [20, 21]. However, these approaches require prior information and/or user guidance in order to accomplish reduction. These approaches may result in (vertical) size reduction only when there are significant number of such calls. Also, they do not provide any indication that certain objects/interactions are omitted. Our approach is better in that JIVE does not discard any calls and provides visual cues to indicate the existence of additional structure and/or information in the diagram. Moreover, our compaction technique does not require user guidance or manual intervention to accomplish compaction.

- *Filter.* A brute-force approach to limit the sequence diagram size is to apply depth filter [7, 8, 15, 20] that hides calls/interactions beyond a (user)-specified call tree depth. The disadvantage with this approach is that 'what is an optimal call tree depth' varies from program-to-program and can be arrived at only after postmortem analysis of the actual diagram. Moreover, if the call tree happens to be highly imbalanced, it may not be possible to arrive at optimal depth. Our approach has the advantage that it is possible to expand and explore one part of the call tree without expanding other parts of the call tree.

  Other forms of filtering include temporal filters [1, 20] that capture part of the execution and package/class filters [1] that discard calls/interactions to specific packages and/or classes. Manual intervention is necessary for both these forms of filtering. JIVE supports both these filters. These filters play a complementary role to our compaction technique as Section 4.2 elaborates.
- *Abstract.* Abstraction involves techniques to infer partial behavior and representing them in the form of sequence diagrams. Typically, these approaches employ offline analysis of several program traces in order to determine some specific behavior. Grati *et al.* [22] presents a semi-automated approach for constructing sequence diagrams from a set of execution traces that are aligned in order to determine the common behavior. Briand [23] provides a methodology to reverse-engineer scenario diagrams – partial sequence diagrams for specific use case scenarios – by analyzing the execution traces. Abstraction approaches, in general, tend to ignore information that are not deemed relevant to the specific objective. While abstraction methods are useful, they serve a different purpose, that is, mainly to understand the correlation between the design time scenario diagrams and the merged runtime versions.
- *Group.* Noda [21] presents an abstraction-based method by identifying and grouping strongly correlated objects to visualize the system's behavior in terms of intergroup interactions. The sequence diagram thus helps in grasping the big picture of the overall behavior of the system and serves as a valuable aid for program comprehension.
- *Slice.* Reticella [24] uses slicing based on behavior model that can treat dependencies identified based on static and dynamic analysis to cut down the sequence diagram size. JIVE, in contrast, provides a systematic approach not only to compact but also progressively expand the compacted diagram allowing the user to interactively move forth between the big picture and the detailed view.

There has been work on using alternate representations to sequence diagrams in order to visualize the execution behavior. While these play a complementary role in comprehending the execution behavior, JIVE stays within the framework of sequence diagram that preserves the temporal aspect of the execution.

- *Circular bundle view.* Extravis [25] enables visualization of execution traces by providing two synchronized views: a circular view that shows the systems structural decomposition and the nature of its interactions during a (part of the) trace and a massive sequence view that provides a concise and navigable overview of the consecutive calls between the system's elements in a chronological order. While the circular view is a good means to display a series of call relations without the need for scrolling, it can be fairly difficult to grasp the temporal aspect within a time fragment. In contrast, JIVE carries out compaction within the framework of a sequence diagram rather than a new representation. Also, Extravis is not equipped with a means to effectively visualize the interactions between the threads. JIVE's visualization, on the other hand, supports multithreading with a unique color for every thread.

DOI: 10.1002/spe

- *State diagram view.* Shimba [12] represents traces as scenario diagrams, extracts state machines from scenario diagrams, detects repeated sequences of events (i.e., behavioral patterns), and compresses contiguous (e.g., loops) and non-contiguous (e.g., sub-scenarios) sequences of events. Again, in this view, the temporal aspect of the execution is lost.
- *Message flow diagram.* ISVis [9] uses both static and dynamic analyses to construct message flow diagrams similar to sequence diagrams. These diagrams represent interaction patterns in the trace. A global view of the execution is displayed in its execution mural. The purpose here is to highlight the interaction patterns.
- *Visual thread overview and execution sequence view.* Trumper *et al.* [26] presents a visualization technique to understand and explore multi-threaded system behavior on various levels of detail. A selected thread's trace data is depicted in a visual thread overview and a sequence view. These views are synchronized, thus allowing developers to efficiently analyze the threads' concurrent behavior. While visual thread overview facilitates quick orientation within the data, execution sequence view depicts call stack and invocation relations between methods for every point in time. Zooming and panning are supported to explore the visualization. JIVE, in contrast, employs a range of compaction techniques to visualize the execution behavior using standard UML notations.
- *Execution pattern view.* Ovation [27, 28] visualizes execution traces using an execution pattern view, a form of interaction diagram that depicts program behavior. Diagrams support a number of operations such as collapsing, expanding, filtering, and execution pattern detection (e.g., repetition). Ovation also supports searches for execution patterns on different criteria. JIVE and Ovation both rely on trace data to construct their diagrams, provide search capabilities to explore the program execution via their respective diagrams, and support techniques to help users focus on regions of interest in the diagram. In contrast with Ovation, JIVE is an online debugger, uses a declarative temporal query language for querying the underlying data model and uses an enhanced sequence diagram to represent program execution.

In short, the advantages of JIVE in comparison with the related work are summarized as follows.

1. JIVE takes a higher-level view of vertical compaction and aims to construct the minimal regular expression label that can concisely represent a sequence of method calls. As illustrated in Figure 2, JIVE's regular expressions use numeric superscripts (rather than + and *) with definite iteration that captures the number of repetitions precisely.
2. JIVE's compaction is semantically richer in that it reduces the diagram and provides an intuitive and concise description of the hidden sub-structure.
3. JIVE also caters to horizontal compaction and the ability to draw a sequence diagram even when there are missing calls, which have been abstracted away because of various filtering techniques (Figure 10).
4. The encoding scheme adopted by JIVE allows one to expand the compacted diagram in a step-by-step fashion.

8. CONCLUSIONS AND FURTHER WORK

Object-oriented program execution can be visualized in a clear and intuitive way using object and sequence diagrams. While object diagrams capture the current state of execution, sequence diagrams summarize the history of interaction between objects for a particular input data. These diagrams were originally intended for use at design time (in UML), but we have adapted these notations for use at runtime and have incorporated them as part of a practical visualization system for Java, called JIVE [1, 3, 4].

This paper addresses the problem of managing the size and complexity of sequence diagrams for long execution runs. Two forms of compaction are discussed, vertical and horizontal compaction, as well as hybrid compaction in which both vertical and horizontal compaction are present. Because long execution runs are the result of iterative computations, vertical compaction is performed by identifying repetitive patterns in a sequence of method invocations. We presented a general labeling scheme based upon the regular expression notation to compactly represent long sequences. A key

DOI: 10.1002/spe

result of the paper is the development of a quadratic-time algorithm for computing these labels based upon the concept of tandem repeats in a sequence. The labeling and compaction can be applied in a top-down or bottom-up fashion to achieve automatic compaction of sequence diagram and support progressive expansion. Our experiments on a wide range of test cases show that the quadratic-time complexity is not a limiting factor in practice, as the labeling is typically applicable to small subsequences of the overall execution sequence.

Horizontal compaction helps reduce the horizontal spread of a sequence diagram by replacing all objects of the same class by a single object. This technique is very effective in minimizing white space in a sequence diagram and is best applicable when objects of a class interact only with objects of other classes. As part of future work, we also plan to extend this technique to polymorphic calls to subclasses of the same class. Horizontal compaction also comes about when the user selectively filters classes and packages from being visualized during execution. This form of filtering also results in horizontal compaction, but it results in out-of-model calls when methods belonging to filtered classes are invoked. Another important contribution of this paper is a notational extension to sequence diagrams to account for missing objects when packages and classes are filtered out from the view.

The paper also presented compaction techniques for multi-threaded Java execution with different forms of interaction: mutually disjoint, mutually exclusive, or potentially interfering. A technique for detecting these interactions and labeling notations for representing then compactly are described. Essentially, multi-threaded programs require a generalization of the call tree representation to a forest of call trees. We identify compaction boundaries and present an quadratic-time algorithm for labeling and compaction.

Finally, in order to validate our techniques, we analyzed the execution runs of a variety of Java programs using the JIVE system in order to demonstrate that the compaction techniques are scalable. Our experimental results show that practical programs lend themselves well to compaction with regular expression labels. Large executions arise because of the iterative nature of programs and the creation of a large number of objects of a few classes. These are the fundamental reasons why Java programs are amenable to vertical and horizontal compaction respectively.

Undoubtedly there are programs that are not amenable well to compaction. Our experiments show that these tend to be highly recursive in nature. Theoretically, pure vertical compaction will not apply if all method calls are distinct and pure horizontal compaction will not apply if all objects are instances of different classes. Such execution traces are necessarily small, in practice, and hence, compaction is not needed for such traces.

Our labeling and compaction technique have been developed as a plug-in for the JIVE dynamic analysis and visualization system for Java. The technique depends only on the execution trace derived from the program execution and not on the Java source code. JIVE is a useful visualization and debugging system for Java, and therefore, the compaction technique has considerable practical impact.

As part of our current and future work, we are exploring compact representation of the run time behavior in which the state of the program is also taken into account. When programs have a repetitive behavior, as in the case of servers, we can construct a finite-state machine diagram that summarizes program behavior with respect to key variables of interest to the user.

APPENDIX: PROGRAMS USED IN EXPERIMENTAL STUDY

Later, we present a brief description of each of the programs presented in our experimental results given in Section 6. These programs were run through JIVE, which is available for download from http://www.cse.buffalo.edu/jive/update. It can be used as plug-in for Eclipse (Kepler or Luna or Mars) in conjunction Java (1.7 or 1.8) – the available versions of Eclipse and JDK, respectively, at the time of writing this paper. The full programs and the resulting execution traces and synthesized compact diagrams are given in the website http://www.cse.buffalo.edu/jive/compaction. Each directory at this website contains the information for one program and its execution. We have used PlantUML to render the compacted diagrams.

DOI: 10.1002/spe

The programs are briefly summarized as follows:

1. Recursion: This is a program with indirect recursive calls. This is a good example for to illustrate when compaction does not work.
2. AVLTree: This program is a menu-driven implementation of AVL Tree that supports insert, search, traversal, and count operations.
3. Kruskal: This program implements Kruskal's minimal spanning tree algorithm. The program reads the graph edges and corresponding weights from an input file and computes the MST edges.
4. Prims: This program implements Prim's minimal spanning tree algorithm. The programs reads the graph from the standard input in the form of weighted matrix and computes the MST edges.
5. Dijkstra: The program implements Dijkstra's single source shortest path algorithm. The program reads the graph nodes and edges and computes the shortest path between the two specified nodes.
6. External Iterator: This program implements a generic binary search tree. It constructs two trees with the same set of elements, but the elements are inserted in a different order. It then uses an external iterator to retrieve values, one at a time, from the two trees in alternation, and checks the retrieved values for equality.
7. Trie: The program implements the Trie data structure that supports read, put, and isEmpty operations. It reads input words from the user and builds a dictionary and allows the user to search for words based on prefix.
8. TreeGUI: This is a GUI implementation of binary search tree and supports insert and delete operations. The program once invoked accepts input values and plots the nodes of the tree visually using AWT. The user can also delete a node from the tree.
9. BST: A Binary Search Tree implementation with over 300 inserts performed. This is to gather and differentiate the compaction behavior of the small program generating large trace as against large program generating large trace.
10. MinRegExp: This program implements the computation of minimal regular expression given a string sequence as the input. This is to study the time and memory usage with respect to the string sequence pattern and size.
11. ProdCons: This implements the two variants of producer–consumer problem. The first one with proper synchronization construct including 'synchronized' methods and wait/notify primitives. The second one does not include these constructs.
12. DiningPhil: This program implements the classical dining philosophers problem. Each philosopher runs as a separate thread and executes hungry, pick, and drop methods in a repeated fashion.
13. Parser: This program is a recursive descent parser for a simple imperative language and generates Java byte-codes as output. Two versions of the program are written: one, fully recursive; the second, using iteration for the statement-sequence rule.
14. Compact Plugin: This program is the JIVE extension that constructs a compact sequence diagram given an execution trace. (That is, we ran the compaction program on itself.) It includes modules for event processing, call tree construction, computation of regular expression labels, compaction of the call tree, and drawing the compacted sequence diagram. We ran the program three times, each run on a different test program: ProdCons, Dijkstra, and TreeGeneric.
15. SpecJVM2008 (https://www.spec.org/jvm2008/): These benchmarks are designed to capture the computations of typical general-purpose applications. They are a suite of real-life applications that help measure the performance of a Java Runtime Environment (JRE), focusing on the performance of the hardware processor and memory subsystem. The benchmarks have minimal dependence on file I/O and do not exercise any network communication between machines.

DOI: 10.1002/spe

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their extensive comments and suggestions, which have greatly helped to improve the paper. We also thank the handling editor, Prof. Nigel Horspool, for his guidance during the revision of the paper.

REFERENCES

1. Gestwicki PV, Jayaraman B. Methodology and architecture of JIVE. *Proceedings of the Fifth ACM Symposium on Software Visualization (SoftVis '05)*, St. Louis, MO, US, May 2005; 95–104.
2. Lessa D, Chomicki J, Jayaraman B. A temporal data model for program debugging. *13th International Symposium on Database Programming Languages (DBPL '11)*, Seattle, WA; August 2011.
3. Lessa D, Jayaraman B. Explaining the dynamic structure and behavior of Java programs using a visual debugger (abstract). *Proceedings of the ACM SIGCSE*, Raleigh, NC, US, March 2012; 668.
4. Czyz JK, Jayaraman B. Declarative and visual debugging in Eclipse. *Proceedings of 2007 OOPSLA Workshop on Eclipse Technology EXchange*, New York, NY, USA, 2007; 31–35, ACM.
5. Cornelissen B, van Deursen A, Moonen L, Zaidman A. Proceedings of 11th European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands: IEEE Computer Society, 2007; 213–222.
6. Myers D. Improving scalability of tools incorporating sequence diagrams visualization of large execution traces. *Master's Thesis*, University of Victoria, British Columbia, 2011.
7. Eclipse. Eclipse Test and Performance Tools Platform. [Online; accessed 28-July-2006].
8. Hamou-Lhadj A, Lethbridge TC, Lianjiang Fu. Challenges and requirements for an effective trace exploration tool. *Proc.12th IEEE International Workshop on Program Comprehension*, Bari, Italy: IEEE Computer Society, v004; 70–78.
9. Jerding DF, Stasko JT, Ball T. Visualizing interactions in program executions. *Proceedings of the International Conference on Software Engineering (ICSE 97)*, Boston, MA, US, May 1997; 360–370.
10. Lange DB, Nakamura Y. Object-oriented program tracing and visualization. *Computer* 1997; **30**(5):63–70.
11. Sharp R, Rountev A. Interactive exploration of UML sequence diagrams. *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Budapest, 2005; 1–6.
12. Systä T, Koskimies K, Müller H. Shimba: an environment for reverse engineering Java software systems. *Software: Practice and Experience* 2001; **31**:371–394.
13. Watanabe Y, Ishio T, Ito Y, Inoue K. Visualizing an execution trace as a compact sequence diagram using dominance algorithms. *Proceedings of the 4th PCODA*, Belgium, October 2008; 1–5.
14. Stoye J, Gusfield D. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science* 2002; **270**(1–2):843–850.
15. Bennett C, Myers D, Storey M-A, German DM, Ouellet D, Salois M, Charland P. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution* 2008; **20**(4):291–315.
16. Cornelissen B, Zaidman A, Van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 2009; **35**:684–702.
17. Sneiderman B. The eyes have it: a task by data type taxonomy for information visualizations. *Proceedings of IEEE Symposium on Visual Languages*, Boulder, CO, 1996; 336–343.
18. Myers D, Storey M-A, Salois M. Utilizing debug information to compact loops in large program traces. *Proceedings of the CSMR'10*, Madrid, 2010; 41–50.
19. Taniguchi K, Ishio T, Kamiya T, Kusumoto S, Inoue K. Extracting sequence diagram from execution trace of Java program. *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, Lisbon, Portugal, 2005; 148–151.
20. Toda T, Kobayashi T, Atsumi N, Agusa K. Grouping objects for execution trace analysis based on design patterns. *Proceedings of Asia Pacific Software Engineering Conference*, Bangkok, 2013; 25–30.
21. Noda K, Kobayashi T, Agusa K. Execution trace abstraction based on meta patterns usage. *Proceedings of the WCRE '12*, Kingston, ON, 2012; 167–176.
22. Grati H, Sahraoui H, Poulin P. Extracting sequence diagrams from execution traces using interactive visualization. *Proceedings of the WCRE '10*, Beverly, MA, 2010; 87–96.
23. Briand L, Labiche Y, Leduc J. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering* 2006; **32**:642–663.
24. Noda K, Kobayashi T, Yamamoto S, Saeki M, Agusa K. Reticella: an execution trace slicing and visualization tool based on a behavior model. *IEICE Transactions on Information and Systems* 2012; **E95-D**(4):959–969.
25. Cornelissen B, Holten D, Zaidman A, Moonen L, Van Wijk JJ, Van Deursen A. Understanding execution traces using massive sequence and circular bundle views. *Proceedings of the 15th IEEE International Conference on Program Comprehension*, Banff, Alberta, BC, 2007; 49–58.
26. Trümper J, Bohnet J, Döllner J. Understanding complex multithreaded software systems by using trace visualization. *Proceedings of the 5th International Symposium on Software Visualization*, Salt Lake City, Utah, 2010; 133–142.

DOI: 10.1002/spe

27. De Pauw W, Lorenz D, Vlissides J, Wegman M. Execution patterns in object-oriented visualization. *Proceedings of the 4th COOTS*, Santa Fe, New Mexico, April 1998; 219–234.
28. De Pauw W, Jensen E, Mitchell N, Sevitsky G, Vlissides J, Yang J. Visualizing the execution of Java programs. *Software Visualization, LNCS* 2002; **2269**:151–162.

DOI: 10.1002/spe