Towards Program Execution Summarization: Deriving State Diagrams from Sequence Diagrams

Swaminathan Jayaraman
Kishor Kamath D
Department of Computer Science and Engineering,
Amrita Vishwa Vidyapeetham, India

Bharat Jayaraman
Department of Computer Science and Engineering,
State University of New York, Buffalo, USA
Email: bharat@buffalo.edu

***Abstract*—We propose a summarization technique that provides a clear and concise picture of the history of program execution with respect to entities of interest to a programmer. We develop our technique in the context of JIVE, a tool for Java execution visualization that renders execution states and history using UML object and sequence diagrams respectively. While these notations have been developed for specifying design-time decisions, the distinguishing aspect of our work is that we adapt their use for execution-time. Sequence diagrams tend to be long and unwieldy, and often exhibit a repetitive structure, hence we develop a novel procedure to summarize the sequence diagram in the form a state diagram with finite states. This summarization is user-driven, in that the user annotates the key variables of interest in the source code. This information together with an execution trace of the program for a particular input enables us to systematically construct a state diagram that summarizes the program behavior for that input. Using multiple execution traces, we show how an integrated state summarization can be obtained. Finally, by choosing different sets of variables, the user may view different summarizations, or perspectives, of the execution. This paper presents our technique along with experimental results from summarizing several different program executions in order to illustrate the benefit of our approach.**

***Keywords*—state diagram, sequence diagram, visualization, dynamic analysis, program behavior**

I. Introduction

The goal of our research is to enhance the runtime comprehension of object-oriented program execution. Towards this end, 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], [2], [3], [4], [5], [6], [7], [8]. JIVE is a dynamic analysis tool available as an Eclipse plug-in and incorporating traditional debugger features such as breakpoints, variable inspection, and stepping, as well as advanced features such as dynamic visualizations of program executions, forward and reverse stepping, and query-based debugging. This system is an outcome of several years of development effort [3], [4] and continues to be actively developed. JIVE is available as a plugin for Eclipse [1] and has been used extensively as a visual Java debugger [5], [8].

This paper focuses on JIVE's visualizations which are founded upon two important types of diagrams: object diagrams and sequence diagrams. The current state of execution in terms of the active objects and their interconnection are captured by the object diagram. The sequence diagram, on the other hand, captures the history of method interactions between the objects of the system. While the Unified Modeling Language (UML) uses these notations to specify design-time behavior, the novel aspect of our work is that we generate these diagrams during the execution time, thereby ensuring common notation between design and execution. In doing so, we have proposed a few extensions to the UML notation in order to represent runtime behavior better. Figure 1 is a screen-shot of the JIVE interface showing a fragment of the source code for the classic 'Dining Philosopher's Problem' along with a visualization of its execution – we discuss this example in more detail later in the paper.

A fundamental problem with visualizations, however, is that they tend to become unwieldy as the number of objects and interactions grow. The object diagram depicts the object structure at a particular point in execution time, and also shows the details within an object, especially the bindings for fields. The sequence diagram, on the other hand, depicts how the objects interact through method calls, but does not have any state information (figure 9). Due to the repetitive nature of program execution, sequence diagrams can become long and difficult to comprehend. We therefore present in this paper a novel technique for summarizing the behavior of sequence diagrams.

There could be many useful summarizations of a program, each with respect to a different set of entities of interest to a programmer. Therefore, we let the user specify, through a suitable interface at execution time, the key variables in various classes and/or objects. This information together with an execution trace of the program for a particular input enables us to systematically construct a *state diagram* that summarizes the program behavior for that input. Using multiple execution traces – either due to new inputs or due to nondeterministic behavior of the program – we show how an integrated state summarization can be constructed. Finally, by choosing different sets of variables, the user may obtain different summarizations, or perspectives, of the execution.

This paper presents our technique along with experimental results from summarizing several different program executions in order to illustrate the benefit of our approach. We also show how these summarizations provide greater insight into the working of the program, helping uncover subtle errors in programming. The rest of the paper is organized as follows: section 2 presents a brief comparison with related work; section 3 presents our summarization technique; section 4 presents experimental results; and section 5 presents conclusions and directions for further work.

978-1-4799-5173-4/14/$31.00 ©2014 IEEE

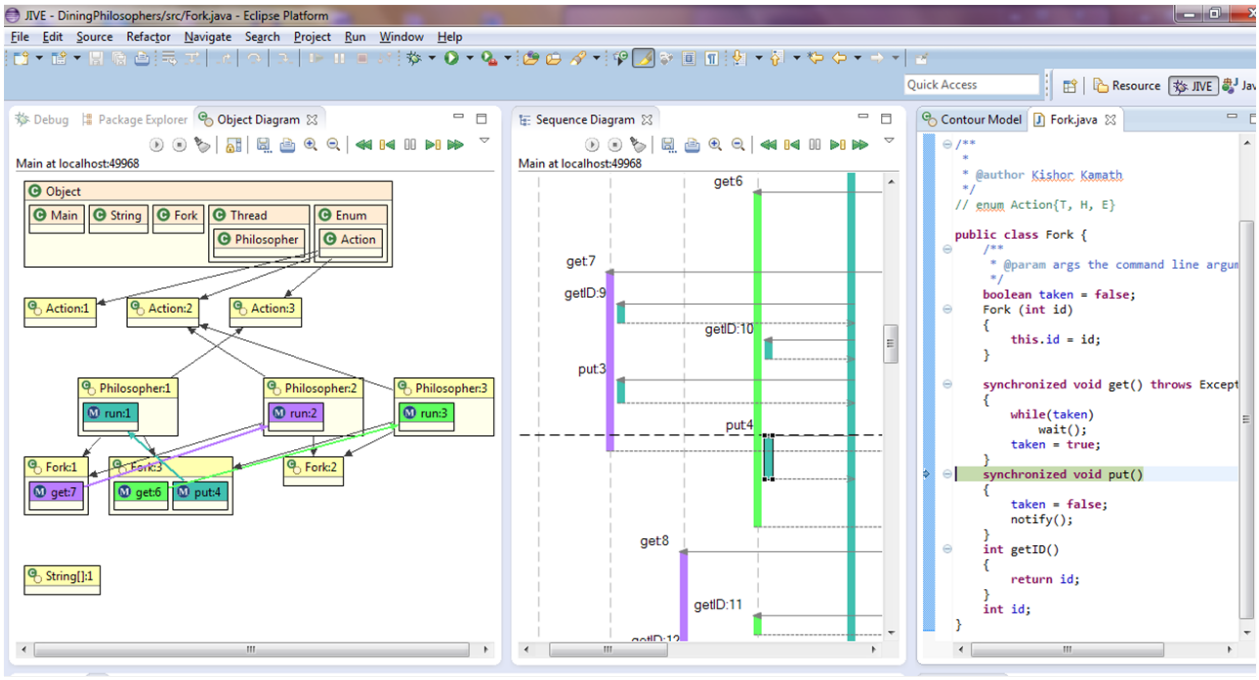

Figure 1. JIVE interface showing source code, sequence and object diagrams for the Dining Philosopher's problem with three forks.

II. Related work

We briefly compare research in two different contexts, static analysis and reverse engineering, which have some commonality with our work. In the context of dynamic analysis of programs, to the best of our knowledge, there has not been any published work on program summarization that constructs state diagrams automatically given an actual execution sequence and state variables of interest to the programmer.

There has been considerable interest in synthesizing state charts from sequence/scenario descriptions in the static context. Essentially, while producing design specifications, it is easier to model individual scenarios using sequence diagrams or through formal rules than producing state machines. On the other hand, it is easier to generate automated source code from state models. Hence, the philosophy behind these techniques were primarily centered around filling the gap between scenario diagrams and state diagrams, and the focus was to create models that capture the desired behavior [9] [10] [11] [12] [13]. However, there is no way to know whether the implemented program's behavior agrees with the desired behavior. In our approach, we work with an actual implementation of a program (in Java), and therefore the summary Zaidi [14] proposes an approach to reverse engineering sequence diagrams from execution traces of object oriented software. It uses the k-tail algorithm to extract a generalized labeled transition system which is converted into a sequence diagram by an intermediary mapping into regular expressions. Beyond offering a systematic and semantically well founded method, the approach can detect interaction operators in sequence diagrams, and merge multiple execution traces to obtain a single sequence diagram. Delemare [15] focus on the reverse engineering of UML 2.0 sequence diagrams in two stages that includes producing basic sequence diagram and then merging them to obtain high level sequence diagrams.

Noda [16] is aimed at producing a sequence diagram from a execution trace. By grouping the strongly correlated objects, visualization of the system's behavior in terms of intergroup interactions is achieved and this is helpful in grasping the big picture of the overall behavior of the system. SmarSET [17] is a summarization and user-driven analysis tool that observes a running program and creates a specialized database for compact representation of execution trace. The user can manipulate execution trace views in various ways using some restrictions, and also check assertions such as program invariants. In contrast with these approaches, we aim to construct a state diagram that summarizes execution based upon an execution trace and user-specified variables of interest.

III. Summarizing Program Execution using State Diagrams

We consider the Dining Philosopher's problem to illustrate our methodology of summarizing program execution. As is well-known, this is a classic illustration of concurrency control, resource sharing, and deadlock. Seated around a table are n philosophers with a bowl of spaghetti in the middle and a fork between two adjacent philosophers. Every philosopher needs the two adjacent forks in order to help himself with spaghetti from the bowl. After eating, he starts thinking until such time he feels hungry again and tries to acquire the two adjacent forks. All philosophers repeat this process indefinitely. The following points may be noted:

- A philosopher can be either thinking or hungry or eating. When a philosopher is thinking, he does not use any fork. When he is hungry, he tries to acquire the left and right forks one at a time. Once both forks are acquired, he starts eating. After eating, he puts down both the forks and starts thinking again.

Thread	Number	Event	Source	Details
Thread-1	282	Method Call	SYSTEM	caller=dining.philosophers.Philosopher:2#run:2, target=dining.philosophers.Philosopher:2#pickUp:3
Thread-2	283	Field Write	SYSTEM	context=dining.philosophers.Fork:2, taken=true
Thread-1	284	Method Entered	SYSTEM	
Thread-1	285	Line Step	SYSTEM	
Thread-2	286	Line Step	SYSTEM	
Thread-1	287	Field Read	SYSTEM	context=dining.philosophers.Philosopher:2, left
Thread-2	288	Method Exit	SYSTEM	returner=dining.philosophers.Fork:2#get:4, value=<void>
Thread-2	289	Method Retur...	SYSTEM	
Thread-2	290	Line Step	SYSTEM	
Thread-1	291	Method Call	SYSTEM	caller=dining.philosophers.Philosopher:2#pickUp:3, target=dining.philosophers.Fork:2#get:5
Thread-1	292	Method Entered	SYSTEM	
Thread-2	293	Field Read	SYSTEM	context=dining.philosophers.Action, E

Figure 2. A snapshot of raw event sequences from JIVE

- If a philosopher is eating, his neighbors cannot eat, since they will not be able acquire both their forks.
- All philosophers could become hungry simultaneously and acquire one fork each. This leads to a deadlock since no one can acquire the remaining fork.
- It is also possible that a philosopher may starve for a long duration because his neighbors always acquire the forks before him.

This problem can be modeled as a state transition system. We consider a case of three philosophers and three forks. Suppose each state were to represent a combination of philosopher's current action, then as each philosopher performs an action, a transition to a new state occurs. The implementation is given below:

Philosopher.java

```
enum Action{T, E, H}
class Philosopher {
  Action action = Action.T;
  Fork left, right;
  void feelHungry() {
    action = Action.H;
  }
  void pickUp() {
    left.get();   right.get();
    action = Action.E;
  }
  void putDown() {
    action = Action.T;
    left.put();   right.put();
  }
}
```

Fork.java

```
class Fork {
  boolean taken = false;
  void get()
  {
    while(taken)
      wait();
    taken = true;
  }
  void put()
  {
    taken = false;
    notify();
  }
}
```

Event Sequence. JIVE provides sequence of runtime events for a single run, executed with a specific input. It should be noted that executing the same program with different inputs will produce different set of events at runtime since the execution can take alternative paths. For a multi-threaded program, due to non-deterministic interleaving of threads, each individual run will produce different sequence of events at runtime even with the same input. In the case of dining philosophers program there are no inputs and each execution will provide different execution sequence since it is multi-threaded. Figure 2 shows the snapshot of the execution trace.

The events captured by JIVE include *new object instantiation*, *thread start/end*, *method call/exit*, *field read/write*, *local variable read/write*, *line step*, etc. Firstly, from the stand point of state diagram generation, only *field write* events and *method call/exit events* are useful. *Field write* operation occurs when an object comes into existence or when an object's field is modified. *Method call and exit* events contains the caller-callee information which enables us to record the stack trace at each *field write* instance. Hence, other types of events can be filtered out.

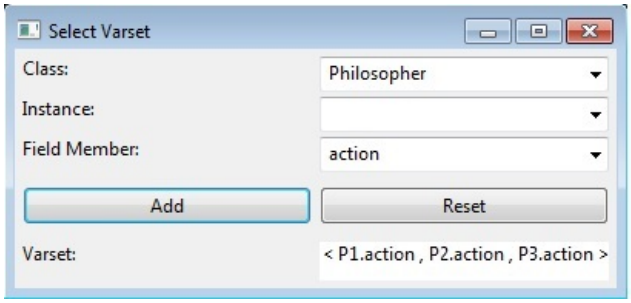

Figure 3. Interface for Specifying Variables of Interest

Variables of Interest. Our focus is on construction of state diagram based on the key variables of interest, which can be specified by the programmer using a graphical user interface, as illustrated in figure 3. To this end, we define *varset* as a n-tuple $\langle v_1, v_2, ..., v_n \rangle$ that captures the variables of interest as annotated by the user. Each v_i takes one of the following four forms.

1) *Object.Field*: In the above example, considering *F[1]*, *F[2]* and *F[3]* as instances of *Fork* class, the tuple $\langle$*F[1].taken, F[3].taken*$\rangle$ denotes that user is interested in the first and third forks' *taken* fields.
2) *Class.Field*: Considering *P[1]*, *P[2]* and *P[3]* as different instances of *Philosopher* class, the tuple $\langle$*Philosopher.action*$\rangle$ denotes the *action* field of all the three instances of *Philosopher* class. i.e $\langle$*P[1].action, P[2].action, P[3].action*$\rangle$.
3) *Object*: The tuple $\langle$*P[2]*$\rangle$ implies that the user is interested in all the fields of the *P[2]* object. i.e. $\langle$*P[2].action, P[2].left, P[2].right*$\rangle$.
4) *Class*: The tuple $\langle$*Fork*$\rangle$ represents that the user is interested in all the fields of every instance of Fork class. i.e. $\langle$*F[1].taken, F[2].taken, F[3].taken*$\rangle$

Secondly, since our focus is only on the *varset*, we only need a subset of *field write* and the *method call/exit* events that pertain to the *varset*. We obtain a projection of the trace with respect to the *varset* by first collecting the *field write*'s of all v_i's in *varset* along with the encapsulating method(s). In the process we capture modifications to the *varset* at the method level, i.e., if a method makes changes to multiple variables in *varset* they will be treated as a single change. The output of projection is a subset of events referred to as *projected events* that contains the set of values assumed by all the v_i's for every method instance.

Note that the *Select varset* dialog box in figure 3 allows the user to choose all the four options mentioned above. To select a class, the user needs to select the class name from the *Class* drop-down box. To select an object, the user can select both the class name and the (object) instance, denoted by *Instance* drop-down box. Finally a field member can be chosen from the *Field Member* drop-down box. Selecting only the class and field member will have the effect of including the field member of all the instances of the class. If the object instance is also specified in addition, it will have the effect of choosing the specific field of that instance.

Turning to our example, suppose *Philosopher.action* is chosen as the variable of interest, the *varset* is ⟨*P[1].action, P[2].action, P[3].action*⟩. The selection of *action* field of *Philosopher* class can be seen from the user interface in figure 3 above. A snapshot of the output produced by the projection is shown below:

#	Method	Values assumed by *varset*
1	Philosopher()	⟨P[1].action: T⟩
2	Philosopher()	⟨P[1].action: T, P[2].action: T⟩
3	Philosopher()	⟨P[1].action: T, P[2].action: T, P[3].action: T⟩
4	feelHungry()	⟨P[1].action: T, P[2].action: T, P[3].action: H⟩
5	pickUp()	⟨P[1].action: T, P[2].action: T, P[3].action: E⟩
6	feelHungry()	⟨P[1].action: T, P[2].action: H, P[3].action: E⟩
7	feelHungry()	⟨P[1].action: H, P[2].action: H, P[3].action: E⟩
8	putDown()	⟨P[1].action: H, P[2].action: H, P[3].action: T⟩
9	pickUp()	⟨P[1].action: E, P[2].action: H, P[3].action: T⟩
10	feelHungry()	⟨P[1].action: E, P[2].action: H, P[3].action: H⟩
11	putDown()	⟨P[1].action: T, P[2].action: H, P[3].action: H⟩
12	pickUp()	⟨P[1].action: T, P[2].action: E, P[3].action: H⟩
13	putDown()	⟨P[1].action: T, P[2].action: T, P[3].action: H⟩
14	pickUp()	⟨P[1].action: T, P[2].action: T, P[3].action: E⟩
	....	.....
	....	.....

A. State Diagram for Single Execution Sequence

The projection of the trace with respect to variables of interest, *varset*, provides a series of methods that were called during runtime and the corresponding changes it made to *varset*. The values of the variables in *varset* and the methods together help form the state diagram: methods represent the transition labels, and the values of variables in *varset* form the states. With reference to the *projected events* above, initially there are three constructor calls that create the *Philosopher* threads. Subsequently, a pattern of methods *feelHungry()*, *pickUp()* and *putDown()* executed by the three *Philosopher* threads continuously.

Since the program is executed continuously for a period of time, it can be noticed that the values assumed by the *varset* repeats every now and then. For example, lines 4 and 13 have the same values assigned to the *varset* though the methods that result in these values are different, i.e., feelHungry() and putDown() respectively. The unique set of values assumed by the variables in *varset* forms the state space. We also record the unique transitions associated with each pair of states.

Input: Projected events
Output: State graph $G_{n\times n}$
Initialize Graph to $G_{1\times 1}$ with a single state START;
Initialize k = 1;
current_state = START;
next_state = next_transition = NULL;
repeat
  Read the next line L_i from the projected events;
  next_state = getState(L_i);
  next_transition = getTransition(L_i);
  if *next_state* $\notin G$ **then**
    $G_{(k+1)\times(k+1)} = G_{k\times k} \cup \{\text{next\_state}\}$;
    G[current_state, next_state] = 1 ;
    Increment k
  end
  if *next_transition* $\notin G$ **then**
    G[current_state, next_state] = 1 ;
  end
  current_state = next_state ;
until *end of projected events*;

Algorithm 1: Construction of state graph

As we scan through the *projected events*, we construct the state graph as provided in Algorithm 1. The nodes denote the unique states and the edges denote the unique transitions and are labeled by the method names. The graph is represented in the form of nxn boolean matrix $G_{n\times n}$ where n represents the total number of states. If there is a transition from a state $S_i \rightarrow S_j$, then the value of G_{ij} is set to 1. Else 0. We include a special state START to represent the start state of the graph from where the first transition begins. As we read the projection output line-by-line we encounter three possibilities from the current state S_i.

1) *A new state*: This implies the line read reveals a new state S_j which has not been encountered before. Consequently, we add S_j to $G_{k\times k}$ and also a new edge $S_i \rightarrow S_j$ labeled by the method name. So the graph extends to $(k+1)\times(k+1)$ boolean matrix $G_{(k+1)\times(k+1)}$.
2) *A new transition*: This implies the line read reveals an already existing state S_j in the graph but the transition has not been encountered yet. Hence, we only add a new edge $S_i \rightarrow S_j$ from the current state to the already existing state.
3) *No new state or transition*: This implies the line read does not reveal a new state or a transition. So the graph $G_{k\times k}$ remains as it is.

The overall state diagram for a single run with two different varsets (i) ⟨Philosopher.action⟩ and (ii) ⟨Fork⟩ is shown in figures 4 and 5. In the figures the highlighted state (in yellow) represents the current state, the number tagged to the edge labels represent the number of times that transitions has occured.

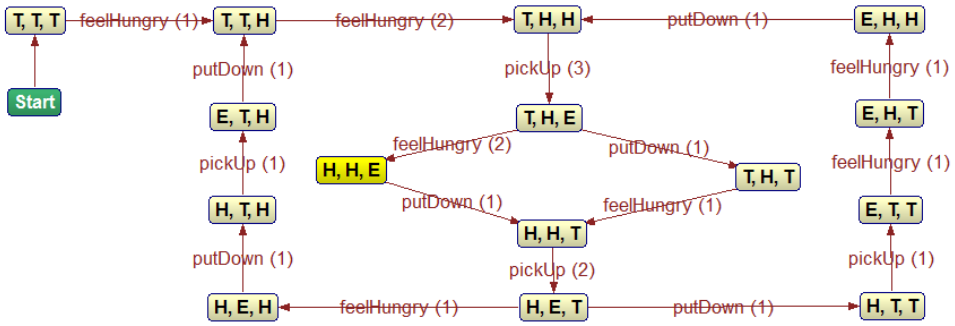

Figure 4. (a) State diagram for dining philosophers: *varset* = ⟨Philosopher.action⟩

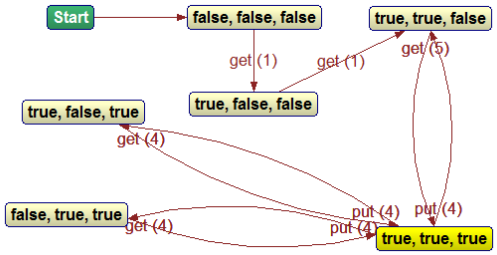

Figure 5. State diagram for dining philosophers: *varset* = ⟨Fork⟩

B. Merging State Diagrams from Multiple Executions

The state diagram from a single run typically does not capture all states and transitions. For example, in figure 5, the following states [H,E,T], [E,T,H] and the deadlock state [H,H,H] are not captured in this run. Also, transitions like [T,T,T] → [T,H,T] and [T,T,T] → [H,T,T] are also not captured. Hence we construct state diagrams for multiple runs and merge them in order to obtain an integrated state diagram. In merging the graphs, G_1 and G_2, for two state diagrams, we need to consider two cases:

1) G_1 and G_2 have the same set of states
   We perform a simple logical OR operation of G_1 and G_2. Since the graphs are represented as boolean matrices, this will ensure all the transitions from the graphs are included in the combined graph.
2) G_1 and G_2 do not have the same set of states
   Here, a simple OR will not suffice since the state space don't match. We first find the states in G_2 that do not exist in G_1 and add them to G_1. All the transitions are set to 0 for the newly added nodes. This gives a new graph G'_1. We perform a similar operation on G_2 to get G'_2. Now both G'_1 and G'_2 have the same set of states and hence the logical OR operation between G'_1 and G'_2 can be performed to get the integrated graph.

Algorithm 2 provides the procedure for merging two state graphs. This approach allows us to integrate state graphs from any number of runs thereby facilitating an integrated view of multiple executions in the form of single state diagram. This has several advantages in software testing since, getting an unified view can help detect lack of test cases that can reveal unexplored paths. It should be noted that the entire process of construction and coalescing of state diagrams can be done with different variables of interest.

Input: Graphs G_1 and G_2
Output: Coalesced state graph G
if $G_1.states \neq G_2.states$ **then**
  $G'_1 = G_1 \cup \{G_2 - G_1\}$;
  $G'_2 = G_2 \cup \{G_1 - G_2\}$;
  for *all newly added states in* G'_1 *and* G'_2 **do**
    Set transitions to 0;
  end
  G = $G'_1 \vee G'_2$;
end
else
  G = $G_1 \vee G_2$;
end

Algorithm 2: Coalesce two state graphs

C. Dealing with large state space

It is evident that whenever state variables take a restricted set of values, we can get concise state diagrams that facilitates in comprehending the behavior. However, if the state variables take on an unbounded range of values, for example integer or string variables, there would be an explosion of states. In order to deal with such scenarios, we provide ways to abstract the state diagram through user defined predicates. The user can define a predicate over a variable to construct an abstracted state diagram. Figure 6 demonstrates the case where a stack size which take any number of states based on its size is reduced to two states - empty or not. The predicate is defined as $size = 0$"

IV. Experimental Results

We have conducted several experiments on eight programs with different sets of key variables. The programs include three well-known synchronization problems (critical sections, readers-writers, and dining philosophers) and the simulation of five simple physical systems (an automatic teller machine, a telephone instrument, a vending machine, a sensor-driver traffic light, and an elevator). We carried out multiple runs

Example		ATM	Critical section		Readers Writers		Telephone	Vending Machine	Traffic	Elevator		Dining Philosophers	
Varset		atm.state	P1.state P2.state	P.state	R1.state R2.state	R1.state W1.state	Person.state	VM.state	Traffic.light	E.floor	E.direction	Phil.action	Fork.taken
Run 1	Total Events	214	199		741		2615	798	2778	4331		702	
	Projected Events	3	7	7	19	8	15	20	8	25	28	20	24
	# states	3	6	3	4	2	5	4	4	5	2	11	6
	# transitions	2	6	3	5	2	5	5	7	10	2	11	8
Run 2	Total Events	339	991		1510		1263	1071	4943	15535		2091	
	Projected Events	13	61	19	41	22	7	26	16	27	83	70	92
	# states	7	8	3	4	3	4	5	4	5	2	18	8
	# transitions	8	13	3	7	4	4	7	12	14	2	25	16
Merged Runs	# States	8	8	3	4	3	6	5	4	5	2	18	8
	# Transitions	9	14	3	7	4	9	7	16	16	2	26	16

Table I. SUMMARY OF RESULTS

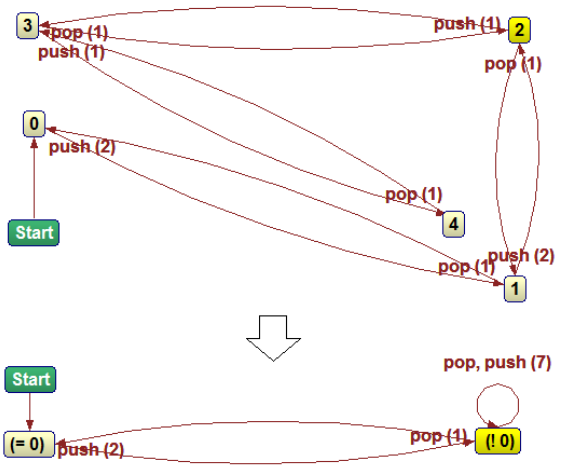

Figure 6. State diagram of a stack reduced to two states based on the predicate $size = 0$

of each program and merged the resulting state diagrams in order to produce a composite diagram. The table I provides the summary statistics of two runs and also their merged output. The snapshots of the state diagrams obtained during the experiments are included in the Appendix. The state diagrams, both single and composite, have provided good insight into the program and also helped detect flaws in the implementation, as explained below.

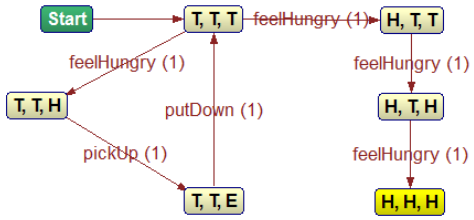

Figure 7. State diagram exhibiting the deadlock behavior

By examining the state diagram several facts about the program behavior can be ascertained. Firstly, the size of the state diagrams is significantly (two orders of magnitude) smaller than the corresponding sequence diagrams. The size reduction helps understand the overall program behavior in a clearer manner and gain insight into the working of the program. For instance, suppose we wanted to know if deadlock occurred, i.e., all philosophers are hungry and have one fork each. That such a scenario never happened in the state diagram can be readily inferred by a simple inspection. An alternate run of the same program does exhibit this behavior and the snapshot of the state diagram is provided in figure 7. The state diagram also clarifies immediately whether any philosopher starved. That this condition did not arise can also be inferred from the fact that every philosopher had got into the eating state. Additionally, the ability to synthesize the integrated view can tell us the extent of deviation with respect to the design-time state diagram.

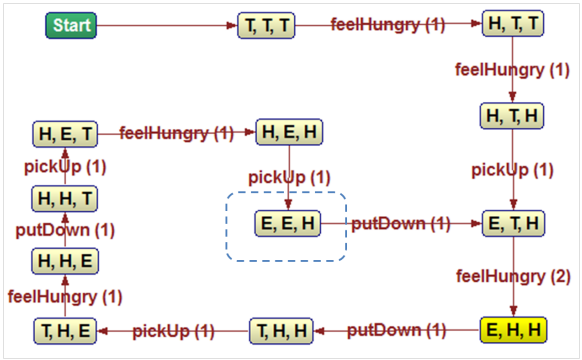

Figure 8. State diagram exhibiting the incorrect behavior

The state diagram can also help uncover flaws or peculiarities in the implementation. For the given dining philosophers code shown in section 3, it is necessary that when a philosopher has finished eating he first move into the thinking state before putting down both his forks. If, on the other hand, a philosopher first puts down both the forks and then moves to the thinking state, it is possible that, between these two actions, another philosopher thread acquires the freed-up fork and moves into the eating state. This leads to a situation where two adjacent philosophers are in the eating state at the same

time, a violation of the problem requirements. Figure 8 shows the state diagram obtained from such an implementation.

V. Conclusion and Future Work

State diagrams provide a succinct summary of the run-time behavior of many programs with a repetitive and cyclic behavior. We have illustrated how such diagrams can be constructed in a systematic way starting from a program's execution trace and user-provided variables of interest. A direct rendering of the execution trace in the form a sequence diagram results in a rather unwieldy structure which is hard to comprehend. As our experiments show, the state diagrams constructed are orders of magnitude smaller than the sequence diagrams and they also provide good insight into program behavior and help uncover subtle errors. By considering multiple execution sequences, we have shown how an integrated state summary can be obtained. We have also presented algorithms for construction and coalescing of state diagrams and substantiated the benefits through the results obtained from several experiments.

We also plan to carry out a more systematic study of the number of execution sequences required to construct a complete state diagram. This is directly related to how complete the test suite is. This problem may be more tractable in our case since we are focusing on certain variables of interest and these are generally restricted so as to have a limited range of values (for ease of comprehension).

Finally, we would like to note that our experimental platform, JIVE, has been crucial for carrying out this research because it provides the execution trace of a program in an easily accessible and usable manner. JIVE can carry out visualizations of a Java program with or without the source code, i.e., one can run a .java file, a .class file, or a .jar file. Essentially, JIVE builds upon Eclipse's debugging interface and provides a new 'perspective' (Eclipse terminology) with object and sequence diagrams. Since the summarization happens at execution time, it is possible to summarize program behavior even when the source code is not available.

References

[1] J. K. Czyz and B. Jayaraman, *Declarative and visual debugging in eclipse*, In Proceedings of the 2007 OOP- SLA workshop on eclipse technology eXchange, pages 31-35, New York, NY, USA, 2007, ACM.

[2] Paul V. Gestwicki, *Interactive Visualization of Object-Oriented Programs*, Ph.D. Thesis, University at Buffalo, June 2005.

[3] Paul V. Gestwicki, Bharat Jayaraman, *Interactive Visualization of Java Programs*, In Proceedings of the IEEE 2002 Symposium on Human-Centric Computing, Languages, and Enviornments (HCC '02), pages 226-235, September 2002.

[4] Paul V. Gestwicki, Bharat Jayaraman, *Methodology and Architecture of JIVE*, In Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis '05), pages 95-104, May 2005.

[5] Hani Girgis, Paul V. Gestwicki, Bharat Jayaraman, *Visual Queries for Interactive Execution of Java Programs* In OOPSLA Companion, pages 156-157, October 2005.

[6] Demian Lessa, *Temporal Model for Program Debugging and Scalable Visualizations*, PhD Thesis, University at Buffalo, September 2013.

[7] Demian Lessa, Jan Chomicki, Bharat Jayaraman, *A Temporal Data Model for Program Debugging*, In International Workshop on Database Programming Languages (In DBPL'11), August 2011.

[8] Demian Lessa, Bharat Jayaraman, *Explaining the Dynamic Behavior of Java Programs using a Visual Debugger (abstract)*, In ACM SIGCSE, March 2012.

[9] Kai Koskimies, Erkki Makinen. *Automatic synthesis of state machines from trace diagrams*, Software - Practice and Experience, Vol. 24 No. 7, Pages 643-658, 1994

[10] Abdolmajid Mousavi , Behrouz H. Far, Armin Eberlein, *An approach for generating state machine designs from scenarios*, In Proceedings of the 11th IASTED International Conference on Software Engineering and Applications, 2007.

[11] M. Sarma , R. Mall, *Synthesis of system state models*, ACM SIGPLAN Notices, v.42 n.11, November 2007

[12] Sebastian Uchitel, Greg Brunet, Marsha Chechik, *Behaviour Model Synthesis from Properties and Scenarios*, In Proceedings of the 29th International Conference on Software Engineering, p.34-43, 2007

[13] David Harel, Hillel Kugler *Synthesizing State-based Object Systems from LSC Specifications*, International Journal of Foundations of Computer Science, Vol. 13 No. 1, Pages 5-51, 2002.

[14] Y.-G. Gueheneuc, T. Ziadi, *Automated reverse-engineering of UML2.0 dynamic models*, In Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering, 2005.

[15] Romain Delamare, Benoit Baudry, Yves Le Traon *Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces*, In Proceedings of the workshop on Object-Oriented Reengineering at ECOOP, 2006.

[16] Noda, Kunihiro, Takashi Kobayashi, Kiyoshi Agusa, *Execution Trace Abstraction Based on Meta Patterns Usage*, In 19th Working Conference on Reverse Engineering (WCRE). IEEE, 2012.

[17] Theodorus Eric Setiadi, Ken Nakayama, Yoshitake Kobayashi, Mamoru Maekawa, *Interactive environment for smart summarization of execution trace*, In IEEE International Symposium on Communications and Information Technology, ISCIT 2005.

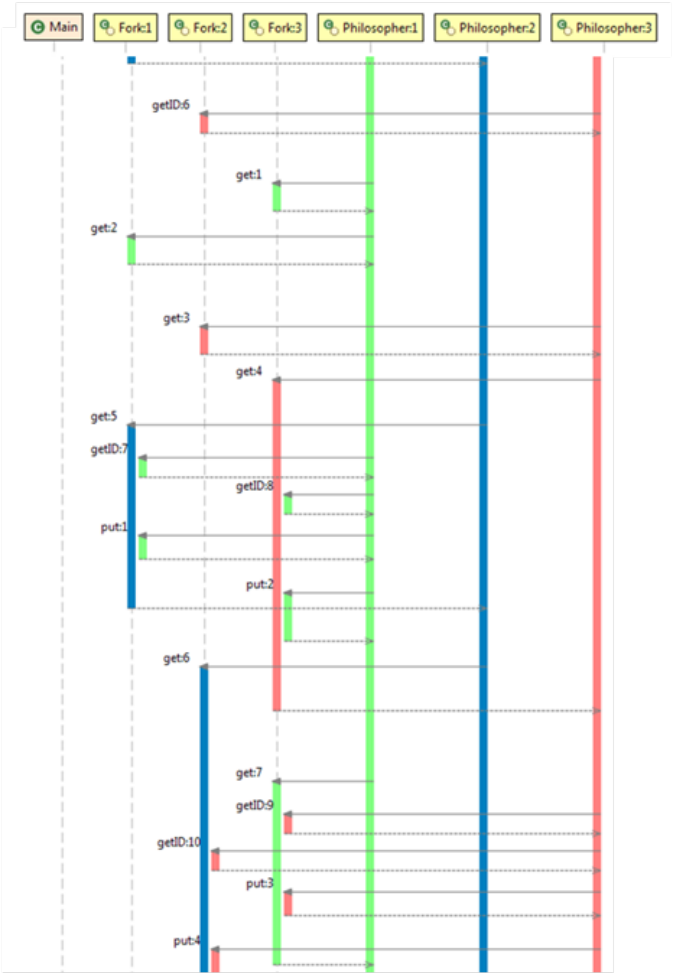

Figure 9. Fragment of Sequence Diagram for Dining Philosophers program