Runtime Visualization and Verification in JIVE

Lukasz Ziarek[1](✉), Bharat Jayaraman[1], Demian Lessa[1], and J. Swaminathan[2]

[1] Department of Computer Science and Engineering, State University of New York at Buffalo, Buffalo, USA
{lziarek,bharat}@buffalo.edu, demian@lessa.org
[2] Amrita Vishwa Vidyapeetham University, Coimbatore, India
swaminathanj@am.amrita.edu

Abstract. JIVE is a runtime visualization system that provides (1) a visual representation of the execution of a Java program, including UML-style object and sequence diagrams as well as domain specific diagrams, (2) temporal query-based analysis over program schedules, executions, and traces, (3) finite-state automata based upon key object attributes of interest to the user, and (4) verification of the correctness of program execution with respect to design-time specifications. In this paper we describe the overall JIVE tool-chain and its features.

Keywords: Runtime visualization · Object · Sequence · State diagrams · Finite state model extraction · Runtime verification

1 Introduction and JIVE Overview

We present in this paper a tool called JIVE for runtime visualization and verification of Java and real-time Java programs running on the Fiji VM [9]. JIVE provides visual debugging, visual dynamic analysis through temporal queries, and visual model synthesis and validation for object oriented programs. The toolchain and associated tutorials and installation instructions are publicly available at: http://www.cse.buffalo.edu/jive/. JIVE is based upon a model-view-controller architecture; the controller component interfaces with the Java Platform Debugger Architecture (JPDA), an event-based debugging architecture, in order to receive debug event notifications such as method entry and exit, field access and modification, object creation, and instruction stepping. JIVE supports two modes of operation, an interactive mode where the user can debug while the program is executing, and an offline mode where a program execution trace (represented as a sequence of events) can be loaded and introspected. JIVE's form-based queries and its reverse step/jump feature allow past program states to be explored without restarting the program [5].

JIVE has been extend to support offline analysis of real-time Java programs. The extension is called JI.FI [2,3], and takes offline traces of events as input. Unlike the vanilla version of JIVE, JI.FI supports precise notions of time and assumes timestamps present in events are gathered from a real-time clock. The JI.FI system is agnostic to both SCJ and RTSJ, offering support for either specification's memory model [4] and linguistic constructs. Our initial work on JI.FI

Y. Falcone and C. Sanchez (Eds.): RV 2016, LNCS 10012, pp. 493–497, 2016.
DOI: 10.1007/978-3-319-46982-9_33

has resulted in some preliminary specialized visual representations of real-time Java programs, specifically focusing on scoped memory, a region based memory allocation strategy that is highly error prone. The true power of Ji.Fi lies in its temporal query analysis engine. By leveraging precise timestamps as well as the temporal database for storing execution events, Ji.Fi is able to detect schedule drift of periodic tasks due to contention on shared monitors between threads of differing priority. The Ji.Fi system does also offer a preliminary sequence diagram that can illustrate visually contended monitors and schedule drift.

Java Path Finder (JPF) [7] is a specialized virtual machine for Java that can simulate the nondeterminism inherent in features such as thread scheduling and selection of random numbers. Although JPF is a very powerful tool and incorporates several execution efficiencies, its textual output is not always easy to follow, especially for long executions. Jive provides a visualization mechanism for JPF's output, which we call the *scheduling tree diagram*. The scheduling tree diagram depicts the choices made (nodes) and the paths traversed by the JPF virtual machine in order to uncover a bug. The paths of this scheduling tree are traversed by the JPF virtual machine in a depth-first left-right manner, and the rightmost leaf node in the search tree corresponds to a property violation. The edges of the search tree are annotated with the JPF instructions that lead to a choice generation. The path leading to the property violation is shown by Jive in more detail using a SD, which summarizes at a high-level the calling sequence leading to the violation. Thus, the three diagrams (scheduling tree, sequence, and object) allow the user to progressively explore different levels of detail in the execution of a concurrent Java program, and together serve as a useful tool for understanding concurrency bugs.

2 Runtime Models: Visualization and Verification

While object and sequence diagrams are useful in clarifying different aspects of run-time behavior, they each have some limitations. Sequence diagrams do not have any state information while object diagrams may be too detailed and also do not convey a sense of how the state changes over time. To remedy these shortcomings, a state diagram is proposed as a more concise way to summarize the evolution of execution than either the object or sequence diagram. A state diagram is an especially appropriate visualization for the class of programs that have a repetitive behavior, especially servers and embedded system controllers.

In order to cater to different summarizations of execution, we let the user specify at a high level which attributes of which objects/classes are of interest. These are referred to as *key attributes* and they typically are a subset of the attributes that get modified in some loop. Given a set of key attributes and an execution trace of Java program for a particular input, we systematically construct a state diagram that summarizes the program behavior for that input. Each field write event in the execution trace could potentially lead to a new state in the diagram. Since the number of field writes is bounded by the number of events n, the complexity of state diagram construction is $O(n)$.

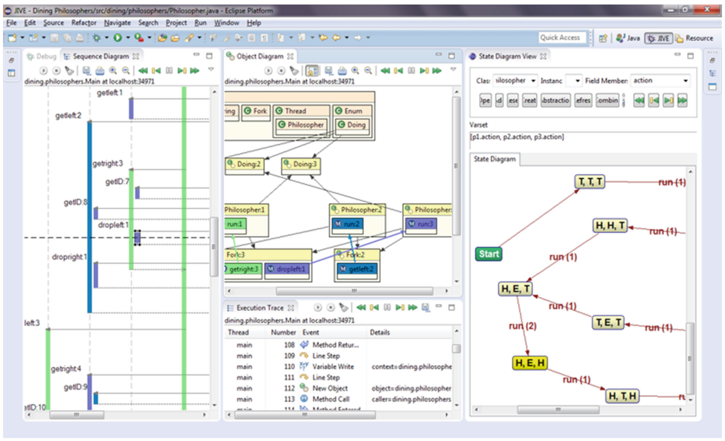

JIVE - Dining Philosophers/src/dining/philosophers/Philosopher.java - Eclipse Platform
Sequence Diagram
Object Diagram
State Diagram View
Execution Trace
dining.philosophers.Main at localhost:34971
Field Member: action
Varset
[p1.action, p2.action, p3.action]
Start
T, T, T
H, H, T
H, E, T
T, E, T
H, E, H
H, T, H
run (1)
run (2)

Figure 1(a)

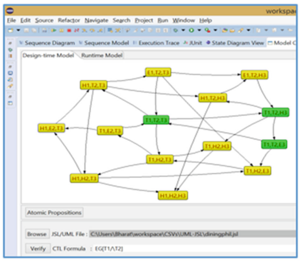


Figure 1(b)

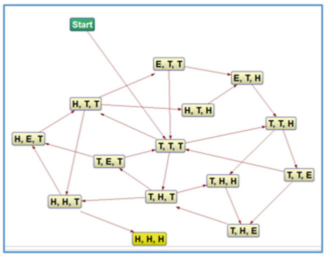
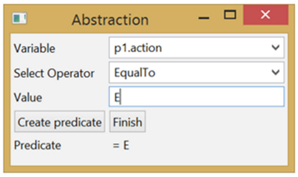


Figure 1(c)



Abstraction

Variable	p1.action
Select Operator	EqualTo
Value	E
Create predicate	Finish
Predicate	= E

Figure 1(d)

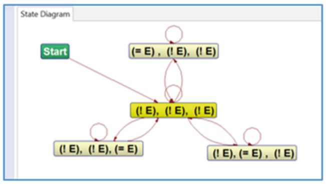


Figure 1(e)

Fig. 1. (a) JIVE user interface showing a fragment of sequence, object, and state diagrams, along with execution trace. (b) JIVE model-checking view showing the states for three dining philosophers and the result of checking EG[T1∧T2]. (c) Finite state model extraction from a Java execution of the three philosophers, with attributes of interest being the philosopher states. (d) Specifying predicate Abstraction in JIVE. (e) Reduced state machine after performing predicate abstraction WRT 'p1.action = E and p2.action = E and p3.action = E'.

We briefly mention some refinements that can help construct more concise and insightful state diagrams: (1) *Predicate Abstraction* helps reduce the state space by reducing the number of possible values for one or more key attributes. (2) *Range Reduction* is similar to Predicate Abstraction and is applicable for a totally-ordered set of values, e.g., integers. By grouping values in ranges, e.g., less than 0, equal to 0, and greater than 0, we can reduce the state space for the integer-valued attribute to just three values. (3) *Masking* some attributes allows us to capture the fact that a key attribute was changed during execution without regard to the value it was assigned to. (4) *Merging Multiple Runs* enables us to obtain more comprehensive state diagrams, as a union of smaller of finite-state machines.

In order to close the loop between design and execution, JIVE provides a consistency-checking capability. JIVE allows the design-time state diagram to be authored by an open-source UML tool, such as Papyrus UML (which is available as an Eclipse plug-in), or the state diagram may be defined textually using a simple notation, referred to as JSL, for JIVE State Language. Given a design-time state diagram, JIVE can check whether the runtime state diagram is consistent with the design by checking whether every state and every transition in the runtime state diagram is present in the design-time diagram. JIVE will highlight states and transitions in the runtime diagram that are not present in the design, thereby signaling a possible error in implementation. Since the runtime state diagram may not exercise all possible states and transitions, the consistency check is an 'inclusion' test rather than an 'equality' test of two state diagrams.

3 Conclusions and Future Work

In this paper we presented an overview of JIVE and its extensions. We described the latest additions to the JIVE toolchain, including generation and refinement of runtime models as well as verification and validation of those models against design time models. The system has been developed over a number of years and the website http://www.cse.buffalo.edu/jive is a repository of all information about the system, including instructions for installation and usage. We provide in Fig. 1 a few screen shots from the latest version of JIVE to illustrate the mechanism described in Sect. 2 of the main paper. For our future work we plan to extend the runtime models and design time models to include notions of time. This extensions, coupled with JI.FI will be particularly useful for validation of real-time system designs against execution traces.

TuningFork [1] is a visual debugger for real-time systems, and much like our JI.FI extension it provides basic visualizations over event streams. A number of tools for enhancing program comprehension of object-oriented programs have appeared over the last two decades. Jinsight [8] provides dynamic views for detecting execution bottlenecks (Histogram View), displaying execution sequences (Execution View), showing interconnections among objects based on pattern recognition algorithms (Reference Pattern View), and displaying profiling information for method calls (Call Tree View). Shimba [10] 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., subscenarios) sequences of events. Ovation [6] visualizes traces as execution pattern views, a form of interaction diagram depicting program behavior; it supports various levels of detail through filtering, collapsing/expanding, and pattern matching.

References

1. Bacon, D.F., Cheng, P., Frampton, D., Pizzonia, M., Hauswirth, M., Rajan, V.T.: Demonstration: on-line visualization and analysis of real-time systems with TuningFork. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 96–100. Springer, Heidelberg (2006)
2. Blanton, E., Lessa, D., Arora, P., Ziarek, L., Jayaraman, B.: JIFI: visual test and debug queries for hard real-time. Concurrency Comput. Pract. Exper. **26**(14), 2456–2487 (2014)
3. Blanton, E., Lessa, D., Ziarek, L., Bharat Jayaraman, J.: Visual test and debug queries for hard real-time. In: Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems. ACM, New York, October 2012
4. Cavalcanti, A., Wellings, A., Woodcock, J.: The safety-critical Java memory model: a formal account. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 246–261. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21437-0_20
5. Czyz, J.K., Jayaraman, B.: Declarative and visual debugging in eclipse. In: Proceedings of the 2007 OOPSLA Eclipse Technology eXchange Workshop (ETX 2007), pp. 31–35. ACM, New York (2007)
6. De Pauw, W., Lorenz, D., Vlissides, J., Wegman, M.: Execution patterns in object-oriented visualization. In: Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 1998), pp. 219–234, April 1998
7. Havelund, K.: Java PathFinder User Guide. NASA Ames Research, California (1999)
8. Zheng, C.-H., Jensen, E., Mitchell, N., Ng, T.-Y., Yang, J.: Visualizing the execution of Java programs. In: Diehl, S. (ed.) Software Visualization. LNCS, vol. 2269, pp. 151–162. Springer, Heidelberg (2002)
9. Pizlo, F., Ziarek, L., Blanton, E., Maj, P., Vitek, J.: High-level programming of embedded hard real-time devices. In: Proceedings of the 5th European conference on Computer systems, EuroSys 2010, pp. 69–82. ACM, New York (2010)
10. Systä, T., Koskimies, K., Müller, H.: Shimba–an environment for reverse engineering Java software systems. Softw. Pract. Exper. **31**, 371–394 (2001)