Consistency of Java Run-time Behavior with Design-time Specifications

Swaminathan Jayaraman
Dinoop Hari
Department of Computer Science & Engineering
Amrita Vishwa Vidyapeetham
Kollam, India

Bharat Jayaraman
Department of Computer Science & Engineering
State University of New York at Buffalo
Buffalo, USA

***Abstract*—We present a novel framework for formal verification of run-time behaviour of Java programs. We focus on the class of programs with a repetitive behaviour, such as servers and interactive programs, including programs exhibiting concurrency and non-determinism. The design-time specifications for such programs can be specified as UML-like finite-state diagrams, or Kripke structures, in the terminology of model checking. In order to verify the run-time behavior of a Java program, we extract a state diagram from the execution trace of the Java program and check whether the run-time state diagram is consistent with the design-time diagram. We have implemented this framework as an extension of JIVE (Java Interactive Visualization Environment), a state-of-the-art dynamic analysis and visualization tool which constructs object, sequence, and state diagrams for Java program executions. JIVE is available as an open-source plugin for Eclipse and makes available the execution trace for facilitating analyses of program executions. We have tested our extension on a number of programs, and our experiments show that our methodology is effective in helping close the gap between the design and implementation of Java programs.**

***Keywords*-finite-state machines, model checking, uml, design-time specifications, run-time consistency, temporal properties, visualization**

I. Introduction

Design and implementation are two important phases in software lifecycle, and, despite the best of efforts, software development is prone to errors in both these phases. While design errors arise due to the failure to capture the requirements correctly, implementation errors arise due to careless coding and improper use of programming constructs as well as due to a deviation of the implementation from the design. We are concerned here with minimizing design errors as well as detecting the deviation of the implementation from design. In the former case, we need to validate the design against its requirement specification, whereas in the latter case, we need to compare the behaviour of the implemented program against the design.

Model checking [1] has emerged as a powerful automatic method for verifying that a model specified in the form of a finite-state transition diagram satisfies a specification given in terms of a propositional temporal logic, such as CTL (Computation Tree Logic) or LTL (Linear-time Temporal Logic). For our purposes, we take this finite-state transition diagram as providing a high-level design of a Java program. Such a design is especially appropriate for a class of repetitive computations, such as servers and interactive programs, including programs that exhibit concurrency and nondeterminism. Model checking has been studied extensively for three decades, with many extensions and variations of the basic concept as well as implemented systems [2].

The main contribution of this paper lies in showing how the run-time behaviour of the implemented Java program can be compared with the design-time specification. It should be noted that the run-time behavior of these programs cannot be reduced simply to their computed output; there are behavioral properties such as absence of deadlock, 'liveness', etc., that are equally important. We therefore construct a finite-state transition diagram from the execution. Towards this end, we first let the programmer specify the key state variables of interest, and we use these state variables along with the execution trace of the program in order to construct a state diagram for a particular execution.

In our previous work [3], we have shown how the execution trace of a Java program together with a set of state variables of interest can help construct an execution-time state diagram for an execution. We implemented this approach as an extension to a state-of-the-art debugging and visualization system for Java, called JIVE (Java Interative Visualization Environment) [4], [5]. Figure 1 illustrates the kind of visualizations that JIVE supports; the problem being visualized is the Dining Philosophers problem (assuming three philosophers). Three types of diagrams are produced by JIVE: object, sequence and state diagrams.

This paper further extends our previous work (and JIVE) by showing how a run-time behavior can be compared with a design-time specification. The design-time state diagram is developed using UML and the model is imported into JIVE. The run-time state diagram is constructed by JIVE and the two state diagrams are checked for consistency. Through this comparison we determine whether the implementation is consistent with the initial design; and, if it does not, we also determine the extent of deviation of the implementation from the design. The result of comparison is depicted visually, highlighting the deviation.

The remainder of this paper is structured as follows:

978-1-4673-7948-9/15/$31.00 ©2015 IEEE

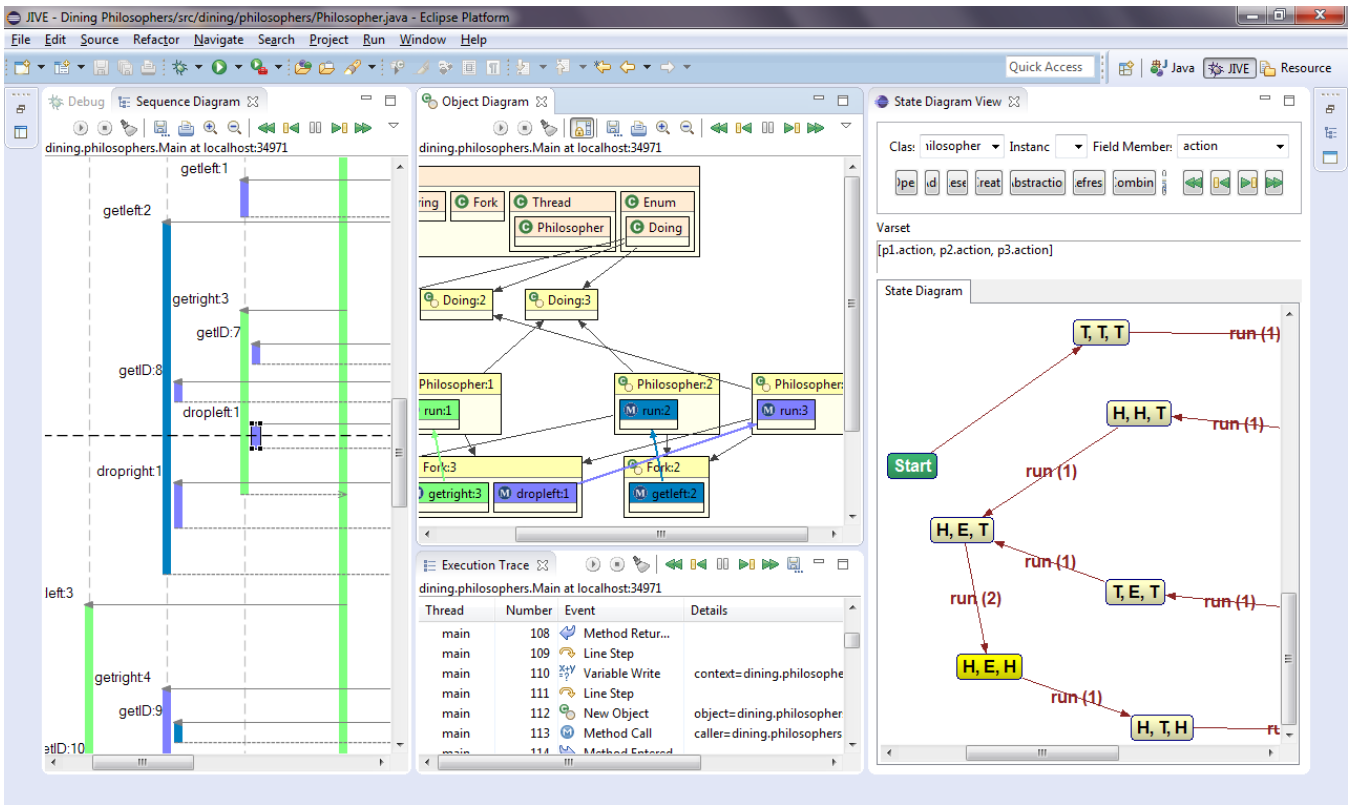

Figure 1. JIVE user interface showing fragments of the execution trace, sequence, object, and state diagrams for the Dining Philosophers problem (with three philosophers). In the State Diagram View, a state H,E,T means that philosopher 1 is hungry, philosopher 2 is eating, and philosopher 3 is thinking.

Section 2 presents closely related work in comparing design-time and run-time behavior of programs; section 3 presents our approach to this problem; section 4 presents experimental work; and section 5 presents conclusions and areas of further research.

II. Related Work

There have been independent efforts in verifying the correctness of (i) the design through the use of model checkers and (ii) the implementation through the use of assertions or contraints expressed through specification languages. Both these approaches use the notion of temporal properties to validate the correctness of their respective behavior. In this section we compare our work against some of these closely related work.

SLAM [6] analyzes and checks temporal safety properties of a C program using a specification language, primarily intended to check the correct working of Windows device drivers. Bandera [7] constructs the model back from Java byte code and then verifies the correctness properties expressed through temporal logic using model checkers such as SPIN, SMV or JPF. SPIN [2] verifies the design against its behavioral requirements using Linear Temporal Logic (LTL). The model checkers NuSMV [8], TAPAs [9] and GEAR [10] allow one to specify the temporal logic formulas using Computation Tree Logic (CTL). Thus, the correctness of the design could be established. However, the problem of correctness of the actual implementation remained unaddresed.

Java PathFinder (JPF) [11] is an explict-state software model checker which can check the correctness of an implementation through its execution. Specifications are given using temporal logic, and JPF uses the instrumented JVM to check whether the temporal properties are satisfied during execution. Java Modeling Language (JML) [12] allows specifying assertions, constraints and invariants in the source code which can be checked using instrumented Java virtual machine ESC/Java2. However, there is no explict design against which the consistency can be checked. Moreover, the above mentioned tools lack visualization and require the user to learn specification languages in order to use them. JIVE not only provides an easy-to-use approach to specify temporal properties, it also provides a platform to check run-time consistency with the design explicitly using the same notation.

Chavez et al. [13] proposed CCUJ as an automated test-based approach to check whether a Java implementation is consistent with the design expressed in the form of a UML class diagram. The class diagram is a static model and

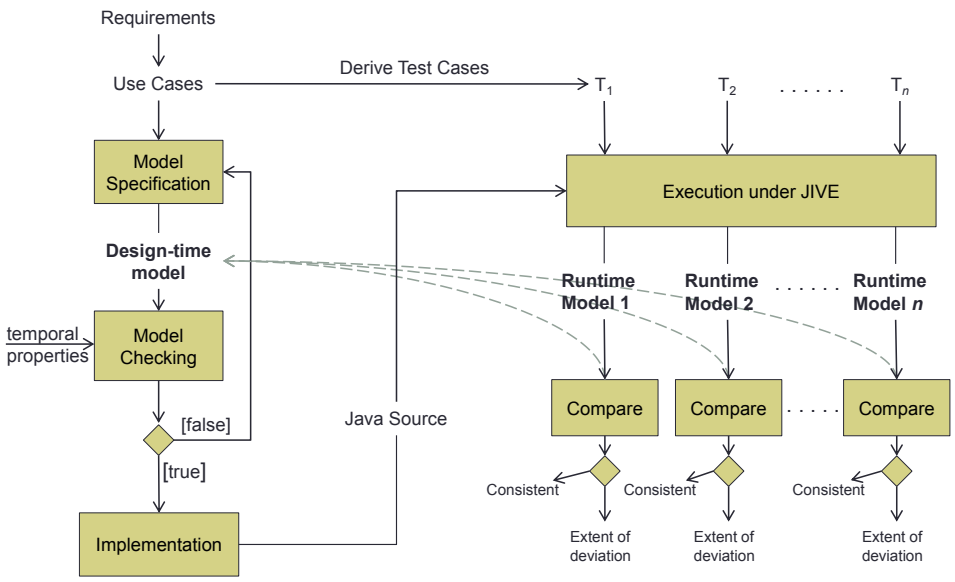

Figure 2. Design-time vs Run-time model validation approach

hence this approach cannot be used to verify the dynamic behavior of programs. Reactis [14] uses a Simulink model to specify the design and allows run-time verification of the implemented C program against this model. In contrast, we take a end-to-end approach to validate the implementation, thereby closing the loop between design-time specification and run-time behavior.

III. Checking Consistency of Run-time Model with Design-time Model

The overall framework to validate the execution against the design is given in Figure 2. As the figure illustrates, we first formulate the design-time model and verify its correctness. Once the design-time model is verified, we then develop the program that implements the design, execute the program under JIVE and reverse-engineer the run-time model. The run-time model is then compared against the design-time model for consistency; any inconsistency is highlighted in the design-time in order to depict the extent of deviation.

A common notation is the key to comparing the design-time and the run-time models. To this end we use the UML statechart notation. This has several advantages. (i) UML is a visual notation that is widely accepted and used by the academia and industry alike. (ii) The state machine can be viewed as a Kripke structure in the terminology of model checking literature [1] and thus we can formally check that it satisfies desirable temporal properties of interest. (iii) In our previous work [3], we have examined techniques to construct a state diagram from the execution trace of a Java program, and JIVE already supports this functionality. Hence, the focus of this paper is to extend JIVE's functionality to support end-to-end design and verification.

We use the classic Dining Philosophers problem to demonstrate our approach. We consider the case of three philosophers seated around a table with a bowl of spaghetti and a fork placed between adjacent philosophers. Any philosopher can either be in a Thinking, Hungry or Eating state. In the Hungry state, the philospher needs to pick the left and right forks in order to move into the Eating state. After eating, the philosopher puts down both the forks and moves into the Thinking state. This process is repeated indefinitely. No two adjacent philosophers can be in the Eating state at the same time, since the fork between them can be used by only one of them at any given time.

A. Modeling the System

As a first step, we use Papyrus [15], an Eclipse plugin, to model the dining philosophers problem. Though there are several UML tools available, the choice of Papyrus is motivated by the need to integrate with JIVE which is an Eclipse-plugin. This allows for seamless integration and extension. Figure 3 shows the design-time model for the dining philosophers problem. There can be at most 27 states since each of the three philosophers can be in any one of the three states: Thinking, Hungry and Eating. However, the states in which two or more adjacent philosophers are in the Eating state are not possible and hence invalid. Consequently, the following states do not exist in the design: [E,E,T], [E,E,H], [E,T,E], [E,H,E], [T,E,E], [H,E,E] and [E,E,E]. It should be noted that the state [H,H,H] represents a deadlock state since all three philosophers have picked one fork each and are in an indefinite circular wait. This state is undesirable and hence should not be in the design. The figure depicts all of the 19 valid states and 42 valid transitions.

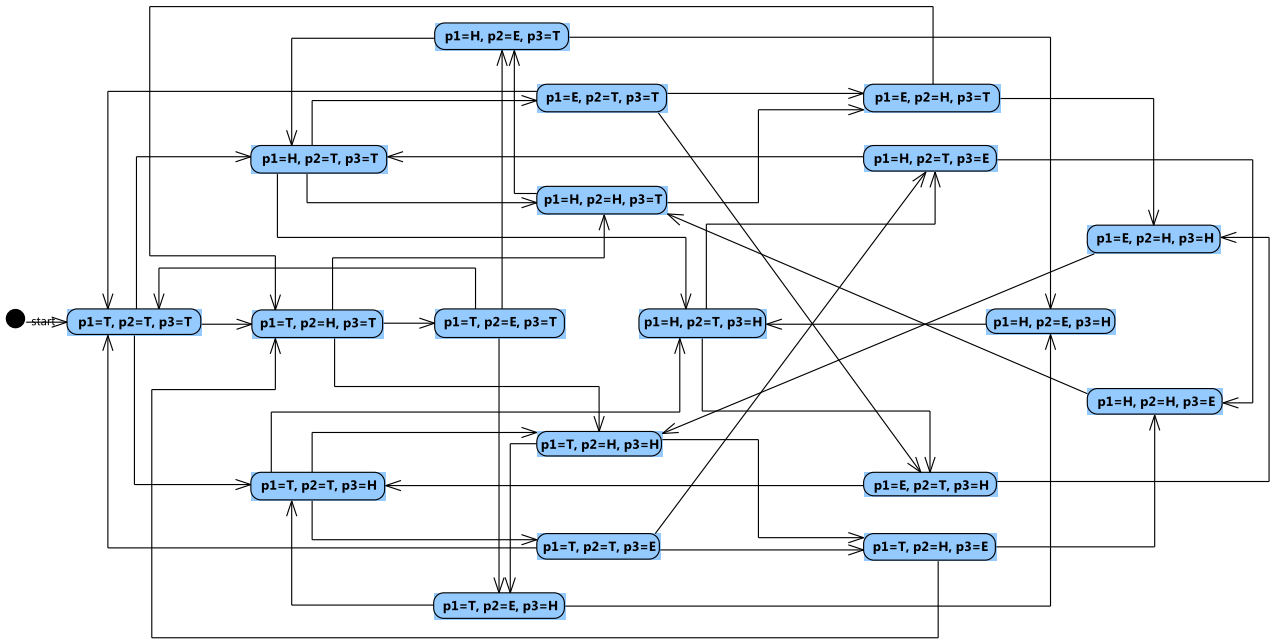

Figure 3. Design-time model for Dining Philosophers problem in Papyrus

B. Model Checking

The next step is to validate the correctness of the design. The state diagram created in Papyrus can be read by our model checking engine transparently and then subsequently verified. The JIVE model checker allows temporal properties, expressed using computation tree logic (CTL), to be specified using a simple graphical user interface. A temporal property is usually one or more variables that must take a specific set of values at certain point in time during the execution. It has three parts: *type, current state, property*. Table I lists some of the important properties.

Table I
TEMPORAL PROPERTIES

EX s p	property p holds in a next state of s
AX s p	property p holds in every next state of s
EF s p	along one path from s, p holds in some future state
AF s p	along all paths from s, p holds in some future state
EG s p	from s, p is true globally along one computation path
AG s p	from s, p is true globally along all computation paths

As the Figure 4 indicates, the temporal properties can be specified by picking the appropriate parameters from the drop-down lists. It can be noted that the drop-down lists are populated by the model checking engine automatically from the Papyrus model and is transparent to the user. This makes it an easy-to-use tool without the need for the user to learn the intricate syntax of specification languages.

The first drop-down box allows one to select the *property type*. The second one lists the *state* from which this property must hold true. The next set of boxes allow one to select the *property* that needs to be verified. The '+' button can be used to specify conjuction and disjuction of properties. For more on model checking, [1] can be referred. Figure 5 shows that *EF [p1=T,p2=T,p3=T] <p1=*,p2=E,p3=*>* is satisfied. i.e. from the state where all the philosophers are Thinking, there exists a path where the second philosopher gets into Eating state. The computation path is highlighted in *green*.

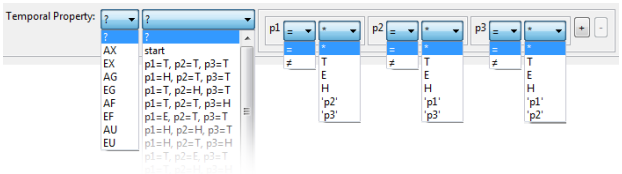

Figure 4. Specifying temporal properties in JIVE

The values populated in the boxes vary based on the data type. The data type is automatically inferred from the states. For instance, if it is an integer, the comparison boxes will also include $>$, $<$, $\leq$ and $\geq$. In addition, the user has the liberty to enter values or wildcards apart from picking the pre-existing values.

Failure of a property usually indicates that design is flawed or incomplete. This situation is caused by the absence of valid states/transtions or the presence of invalid

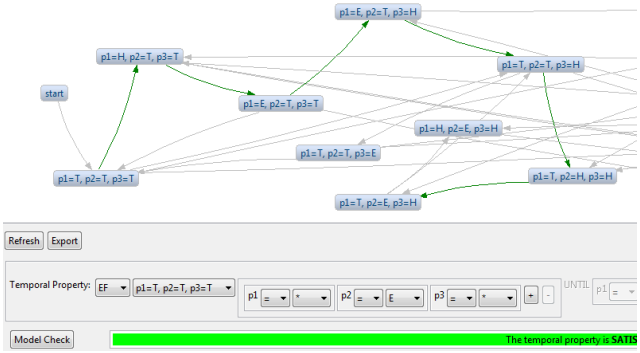

Figure 5. Model checking in JIVE

states/transtions in the design. In such a case, the design needs to be fixed. In rare cases, it can be due to incorrect specification of the temporal property itself. It must be noted that JIVE only provides a platform to specify and verify the properties. Formulating the correctness behavior through a right set of temporal properties is the responsibility of the user.

C. Construction of Run-time Model from Execution

Once the design-time model is verified, the next step is to translate the design into Java code (Figure 6) and execute it under JIVE. JIVE's state diagram feature [3] allows the user to project the state diagram with respect to key variables of his/her interest. Here we have chosen the key variables as {*p1.State,p2.State,p3.State*} in line with the design. This enables us to compare the design-time and run-time models in the same notation. The run-time model generated is shown in Figure 7. The run-time model will be different for different executions, since the inputs may vary, causing changes in the control flow of the program. The presence of non-determinism in a program may also contribute to varying run-time models for different executions.

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() {
    left.put();   right.put();
    action = Action.T;
  }
}
```

Fork.java

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

Figure 6. Java implementation of Dining Philosophers problem

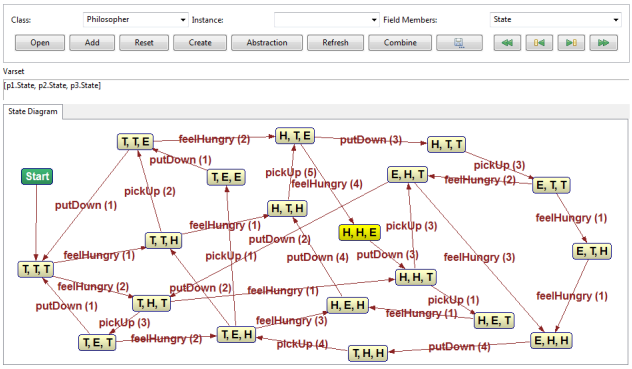

Figure 7. Run-time Model obtained using State Diagram feature of JIVE

D. Comparison of Design-time and Run-time Models

Once the run-time model is constructed, it can be checked against the design-time model for consistency. This problem can be considered as the comparison of two graphs to determine the similarity measure. Let G^D and G^R denote the design-time and run-time graphs respectively. We note that both G^D and G^R can be of different size to start with. Graph G^R may contain only a partial set of states and transitions present in G^D. This is so, since not all states of the design G^D may be traversed in a single execution. From a consistency point of view, assuming that design G^D is valid, we need to identify the illegal states and transitions present in G^R. Essentially two cases are possible during the comparison:

1) $G^R - G^D = \phi$. This denotes the fact that all the states and transitions found in G^R are also present in G^D. This implies that run-time model is *consistent* with the design.
2) $G^R - G^D \neq \phi$. This denotes the existence of at least one state or transition in the run-time model G^R that is not present in the design G^D. Such states or transitions are illegal. In such a case, the run-time model *is not consistent* with the design.

The algorithm for checking consistency is given in Figure 9. Beginning with the *Start* states of both graphs, we traverse the run-time graph G^R and compare against the design G^D. (i) Whenever we encounter states that are common to both graphs, we mark those states in G^D as *green*. (ii) When we encounter a transition in G^R that is not present in G^D, we add this new transition to G^D and mark it as *red*. (iii) Finally, when we encounter a new state in G^R that is not present in G^D, we add both the new state and transition to G^D and mark them both as *red*.

Once the algorithm completes, the run-time graph G^R is superimposed on the design G^D. And all the states belonging to the set $(G^R - G^D)$ are marked in *red*. These denote the set of illegal states and transitions. All the states that are common to both graphs $(G^R \cap G^D)$ are marked in *green*.

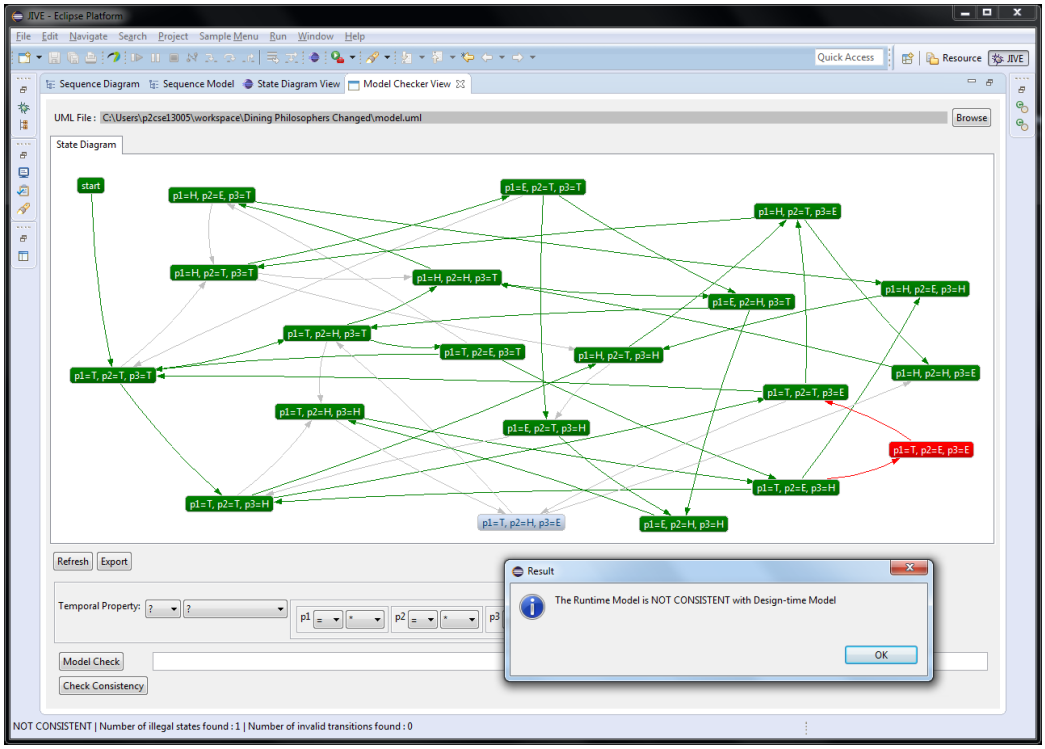

Figure 8. Verifying the Consistency of Run-time Model against Design-time Model

Algorithm COMPARE(G^D, G^R, $start^D$, $start^R$)
  Set $current^D \leftarrow start^D$
  Set $current^R \leftarrow start^R$
  for all $s \in$ nextState($current^R$) AND not visited **do**
    if $s \in$ nextState($current^D$) **then**
      Mark s in G^D as GREEN
    else if $s \notin$ nextState($current^D$) AND $s \in G^D$ **then**
      $t \leftarrow$ newTransition($current^D$, s)
      Mark t as RED
    else
      $s' \leftarrow$ newState(G^D, s)
      $t' \leftarrow$ newTransition($current^D$, s')
      Mark s' and t' as RED
    Mark s in G^R as visited
    COMPARE(G^D, G^R, $current^D$, $current^R$)

Figure 9. Algorithm: Compare Design-time vs. Run-time Models

The rest of the states ($G^D - G^R$) remain *blue*. These states denote that they were not seen during this execution.

Figure 8 depicts the design-time model superimposed with the run-time model from an execution of dining philosophers program. As can be inferred, the diagram clearly captures the presence of an *illegal state*: [T,E,E] denoting the fact that philosophers 2 and 3 are simultaneously in Eating state. With reference to the program code, there is an error in the order in which a philosopher moves from the Eating to Thinking state. As the red dot in the code in Figure 6 indicates, a philosopher first puts down the forks and then moves into the Thinking state. Since the other philosopher is waiting on the fork becoming free, there is a possibility that the waiting philosopher is scheduled between these two actions, picks the free fork and moves to the Eating state causing a transition to the illegal state [T,E,E]. This invalid state is highlighted in *red* in the output which could have gone unnoticed otherwise.

IV. Experimental Results

We carried out numerous experiments on well-known synchronization problems such as Dining Philosophers, Critical Section, Readers Writers and simulations of physical systems like Automated Teller Machine, and Traffic Light controller. We compared multiple implementations of a design against its design-time model in order to validate our method. These experiments provided valuable insights into the implementation revealing errors like the one depicted in Figure 8.

In somes cases, we could uncover flaws in the design itself while checking for the consistency. For instance, in the dining philosopers problem, the state transition made by a single philosopher with his left and right forks is given in Figure 10 (left). The consistency check revealed the presence of an apparently invalid state: [c1=PICKED, p1=T, c3=FREE] as shown in Figure 10 (right). That is, in

our model we assumed that, while the philosopher (p1) is in the Thinking state, both his forks must be free. We did not consider the possibility of the next philosopher (p3) holding the fork (c1). This revealed the incompleteness of the design which was clearly highlighted by the output.

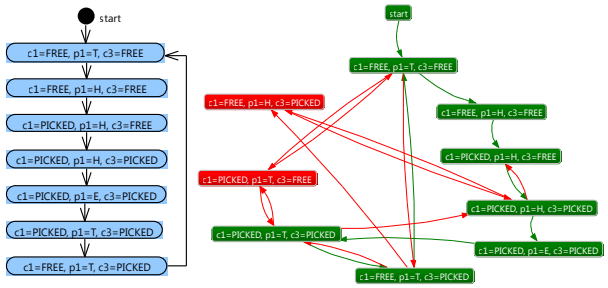

Figure 10. Incompleteness of design uncovered

Similarly while running the other experiments, we were able to identify such mistakes that can occur during the design and/or implementation.

V. Conclusion

We presented a novel framework to verify the run-time behavior of Java programs that exhibit repetitive behavior. The UML state diagram serves as a concise visual specification of the behavior of such programs, and we showed that JIVE [3] can be used to construct the run-time state diagram back from the program execution. This paper shows how the design-time and run-time models can be compared. If the run-time model is not consistent with the design-time model, the states and transitions of the run-time model are highlighted to point out the difference.

We have implemented this framework as an extension to JIVE, experimented it with many programs in order to demonstrate that this methodology is effective in closing the gap between design-time specification and execution-time behaviour. We presented the algorithm to perform this consistency check. In some cases, we were also able to catch flaws in design and correct them. Thus, this framework serves as an end-to-end verification mechanism for the Java programs that have repetitive behavior.

We plan to combine execution models extracted from multiple runs (e.g., based on the test suite for the program) and then verify the consistency of the unified run-time model against the design-time specification. This in turn will take us closer to a complete verification of the program. We would like to note that JIVE has served as an indispensable platform for carrying out this research. It is available as an Eclipse plugin from www.cse.buffalo.edu/jive and can also be used to construct visualizations for Java programs even when their source codes are not available.

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. on Prog. Lang. and Sys.*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[2] G. J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[3] S. Jayaraman, D. Kishor Kamath, and B. Jayaraman, "Towards program execution summarization: Deriving state diagrams from sequence diagrams," in *Seventh Intl. Conf. on Contemporary Computing*. IEEE, Aug. 2014, pp. 299–305.

[4] P. Gestwicki and B. Jayaraman, "Methodology and architecture of JIVE," in *Proc. of the 2005 ACM symp. on Software visualization*, ser. SoftVis '05. ACM, 2005, pp. 95–104.

[5] D. Lessa, J. K. Czyz, and B. Jayaraman, "JIVE: A Pedagogic Tool for Visualizing the Execution of Java Programs," University at Buffalo, Tech. Rep., Dec. 2010.

[6] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, ser. POPL '02. ACM, 2002, pp. 1–3.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting Finite-state Models from Java Source Code," in *Proc. of the 22nd Intl. Conf. on Soft. Engg.* ACM, 2000, pp. 439–448.

[8] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A New Symbolic Model Verifier," in *Computer Aided Verification*, ser. LNCS. Springer, 1999, vol. 1633, pp. 495–499.

[9] F. Calzolai, R. De Nicola, M. Loreti, and F. Tiezzi, "TAPAs: A Tool for the Analysis of Process Algebras," in *Trans. on Petri Nets and Other Models of Concurrency I*, ser. LNCS. Springer, 2008, vol. 5100, pp. 54–70.

[10] "GEAR - A Game-based Model Checking Tool," http://jabc.cs.tu-dortmund.de/modelchecking/, accessed: June, 2015.

[11] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Automated Software Engg.*, vol. 10, no. 2, pp. 203–232, Apr. 2003.

[12] P. Chalin, J. Kiniry, G. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," in *Formal Methods for Components and Objects*, ser. LNCS. Springer, 2006, vol. 4111, pp. 342–363.

[13] H. M. Chavez, W. Shen, R. B. France, and B. A. Mechling, "An approach to testing Java implementation against its UML class model," in *Model-Driven Engg. Lang. and Sys.*, ser. LNCS. Springer, 2013, vol. 8107, pp. 220–236.

[14] "Software Testing and Validation with Reactis," http://www.reactive-systems.com/, accessed: June, 2015.

[15] "Papyrus," http://www.eclipse.org/papyrus/, accessed: June, 2015.