Extracting Static and Dynamic Structural Information from Java Concurrent Programs for Coverage Testing

Rafael R. Prado, Paulo S. L. Souza, George G. M. Dourado,
Simone R. S. Souza, Julio C. Estrella, Sarita M. Bruschi
Department of Computer Systems, University of Sao Paulo, ICMC
Sao Carlos, Brasil
Email: rafaelrp, pssouza, georgemd, srocio, jcezar, sarita @icmc.usp.br

Joao Lourenco
Universidade Nova de Lisboa, NOVA-LINCS
Caparica, Portugal
Email: joao.lourenco@fct.unl.pt

***Abstract*—This paper proposes novel techniques for the extraction of structural information from the source code of Java concurrent programs for their coverage testing. Such techniques differ from others because they consider synchronization flow among processes/threads, distinct paradigms of communication/synchronization, method calls and pointer manipulation. The structural information gathered from the source code is kept in a test model based on a Parallel Control Flow Graph (PCFG) and helps the generation of an instrumented source code, used for a future generation of trace files and to replay the concurrent execution. The results show the techniques can generate both an instrumented code and a PCFG for Java concurrent programs effectively, extracting static and runtime information required for structural testing.**

keywords – Testing tools, Concurrent software, Parallel Software, Coverage, Structural testing, Java

I. Introduction

Software testing is a major concern in the development of complex software systems and, in the current multicore era, concurrent program testing verifies the quality of this application domain. However, the presence of multiple interacting threads and/or processes make the test models, criteria and tools used in the context of sequential software unsuitable for the concurrent setting.

The TestPar project (http://testpar.icmc.usp.br) [1], [2], [3], [4], [5] aims at the creation of a full infrastructure capable of providing test models, criteria and tools for the structural testing of concurrent programs. The current structural testing criteria developed for the project consider both communication and synchronization paradigms (shared memory and message passing) in a unified way [1].

The ValiPar software testing tool [3], [5], [6] analyzes and modifies the source code to extract structural information from a concurrent program statically and at runtime so that proper support can be offered for the testing activity. The previous instances of ValiPar offer support to the TestPar structural testing models and criteria, considering either shared memory or the message passing paradigm individually. MPI [5] and BPEL [6] ValiPar instances support the message passing paradigm, while Pthreads [3] ValiPar instance supports shared memory. Such previous instances do not consider calls of methods and pointers required for a deeper coverage testing of multi-paradigm concurrent programs.

The approaches related (described in Section II) are characterized mainly by their focus on specific interaction (communication/synchronization) paradigms or consideration of only theoretical aspects. The literature lacks software tools for the structural testing of concurrent programs that support the usage of both synchronization and communication paradigms on a same concurrent program and in a broader sense. None of the solutions found describe the practical aspects on how to fill in those testing models with testing elements, i.e. how the instrumentation and graph generation of typical multi-paradigm concurrent programs can be implemented.

This paper presents novel techniques to provide more complete sets of required elements and trace files for the structural testing of concurrent programs that include aspects, such as pointers, method calls and use of both synchronization and communication paradigms on the same program. It proposes a new PCFG (Parallel Control Flow Graph) generation and new support for trace generation to make static and runtime information available and compatible for the structural testing of concurrent program. The techniques have been implemented in the ValiPar software testing tool for Java concurrent programs, described in this paper.

The paper is organized as follows: Section II addresses studies related to our topic, highlighting their main features and gaps; Section III presents the TestPar test model and tools and discusses the identification of Java concurrent program flows; Section IV describes the extraction of structural information for PCFG and the trace generation; Section V evaluates the techniques implemented in the ValiInst module; finally, Section VI reports the main conclusions and future work directions.

II. Related Work

Different concurrent software testing approaches related to our study can be found in the literature. Taylor et al. [7] propose the Concurrency State Graph to map the concurrency states (nodes) and transition between states (edges). Yang et al. [8] describe a Parallel Program Flow Graph (PPFG) to represent flows of concurrent programs. Chen et al. [9] propose a hybrid approach to deal with atomicity violations by recording the data flow at runtime and using an intraprocedural static analysis to gather data from a source code. New executions of the program are guided by a summary of such analysis which determines source code unexplored braches. The HAVE (Hybrid Atomicity Violation Explorer) software testing tool offers the support required for the detection of atomicity violations on shared memory (multi-threaded) Java programs. Christakis and Sagonas [10] conducted a static analysis to detect usual types of message passing errors in languages related to the dynamic process creation and asynchronous message passing.

JaBUTi [11] is a unit testing tool for sequential Java programs that uses the program's bytecode to extract structural information and instrument it to generate trace files during the test execution. It handles only control and data flow information of the tested program. Besides, it tests each method individually (unit testing), which may be not enough to reveal concurrency bugs.

The ConTest software testing tool implements the Java testing of multi-threaded programs by exercising possible interleavings to find concurrency-related bugs [12]. CHESS tool is based on the definition of concurrency scenarios and uses model checking to cover all thread schedules in a systematic way [13]. ConTest and CHESS tools test concurrent programs, however, they do not use the structural testing approach or handle concurrent programs that synchronize and communicate using shared memory and message passing paradigms together.

Some other frameworks and tools help the testing and analysis processes. Soot [14] is a framework widely used for Java bytecode manipulation that offers a large variety of static analysis resources. It provides algorithms for an effective analysis of tested program that can be applied to the structural testing activity. However, the actual program's structural representation and strategies must be provided by the developer of the tool for a successful comparison of static and runtime extracted information.

JUnit (http://junit.org) is a unit testing framework that helps the developer to write repeatable tests for Java program. Although it can be used for structural testing, it does not establish a model or test criteria to be covered or used in a systematic testing activity. The software tester must identify the elements to be tested and apply a suitable testing strategy.

JML is a behavioral interface specification language that enables the specification of invariants and pre- and postconditions of a module. It can be used by tools, such as assertion-checking compilers and unit testing tools for the verification and validation of the tested software behavior [15].

Souza et al. [16] provides a systematic review on concurrent software testing, which complements the related works with a larger set of references.

III. Structural Testing of Concurrent Programs

The structural testing of concurrent programs is based on well-defined testing steps [16]. Initially, the program source code is analyzed and instrumented to extract all the required information (according to a test model). The static analysis of the program results in a PCFG (Parallel Control Flow Graph) and the instrumentation enables the tracing of model information at runtime. The static information is used in the derivation of required elements and the runtime information is used for the measurement of the actual coverage of test case executions.

The test model used here was proposed in [1]. It organizes the information extracted from the source code as a PCFG that represents control, data and synchronization flows. If only the control flow is considered, the PCFG can be defined as a set of disconnected subgraphs (control flow graphs), each one representing a thread of the tested system. A control flow graph (CFG) consists of a set of nodes and edges. Edges link two sequential nodes belonging to the same thread graph.

The synchronization flow of a concurrent program is represented by operations of synchronization and communication and also synchronization edges among threads and processes. These operations are classified and clustered according to their semantics (sender, receiver, sender-receiver, blocking, non-blocking, synchronous and asynchronous) [2].

The data flow is represented by use and definition operations over data items (i.e. memory regions). Computational and predicate use already defined for sequential programs [11] occurs in both a computation statement and a condition associated with control flow statements, respectively. A use can be a shared use if the data item is shared among threads, or a message use if the data element is associated with the transmission of messages [1].

Information is extracted from a source code as sets of tuples. For example, a data flow tuple is composed of a data operation (definition or use) and a target memory position. Required elements are generated based on sets of tuples, according to structural testing criteria. Testing criteria act as predicates guiding the choice of test cases that systematically increase the coverage of required elements

The ValiPar structural software testing tool automates those testing activities for the structural testing of concurrent programs. It consists of four modules, namely ValiInst (instrumentation and PCFG generation), ValiElem (generation of required elements), ValiExec (test case execution) and ValiEval (testing criteria coverage evaluation).

The techniques proposed in this paper were implemented into the ValiInst module, instantiated for Java. Java was chosen because of its wide use in academic and industrial contexts and its libraries related to concurrency.

The control and data flows of Java programs have been extensively investigated for sequential programs [11]. The control flow considers method calls and conditional and repetition structures, whereas the data flow is based mainly on local variables, class and instance fields. Each variable has a data type (primitive or object) and can be handled by either a value or a reference.

The synchronization flow of Java concurrent programs considers the use of both shared memory and message passing paradigms. These resources are provided by the standard library or language built-in structures. class is commonly used for the creation of threads through the calling of the method. The method waits for the end of a thread execution. Java provides synchronized blocks and methods, which have the semantics of a mutex, and monitor methods (, and) to offer the semantics of condition variables. Other operations with a high diversity of semantics for shared memory can be found in the package, which provides the semantics of semaphores, barrier and locks.

The message passing paradigm is supported by TCP and UDP sockets. The interactions among processes can be synchronous/asynchronous and blocking/non-blocking and communication involves one or more senders and receivers. The semantics are available in , , and classes.

IV. PCFG Generation and Instrumentation

The structural testing of concurrent programs requires the identification of all control, data and synchronization flows, both statically and at runtime. For example, PCFG must reliably represent the whole concurrent program being tested. Similarly, the trace file produced during the instrumented program execution must represent the dynamical aspects executed. The static and runtime information concerning these flows must be compatible with each other to enable a coverage analysis after the generation of trace files and required elements.

The Java concurrent program presented in Listings 1, 2 and 3 has been used to describe, without loss of generality, our solution for the automation of PCFG generation and code instrumentation addressing the compatibility issues related to the program flows. It illustrates a master-slave program that consists of two processes and three threads, which interact through both message passing and shared memory paradigms.

Listing 1: Master process of the master-slave program. It sends a token to the first slave program and waits for a token from the second slave.

```
 1 // Master (Process 0 Thread 0)
 2 class Master {
 3   public static void main(String[] args) {
 4     DatagramSocket master = new DatagramSocket();
 5     DatagramSocket server = new DatagramSocket(5000);
 6     int token = 2;
 7     for (int i = 0; i < 2; i++) {
 8       MySocket.send(token, master, 4000);
 9       token = MySocket.receive(server) + 1; }
10     }
11   } // main
12 }
```

The master class (thread 0 in process 0) starts transmitting a token in line 8 to Slave 0 (thread 0 in process 1) in line 9. Slave 0 retransmits the received token (line 10) to Slave 1 (thread 1 in the same process - line 21). Finally, Slave 1 (line 22) transmits the token back to Master (line 9). The threads in the same process use a shared buffer properly synchronized by a binary semaphore (created in line 5). Slave 0 updates the shared buffer and notifies (release) Slave 1 by depositing a token in (line 10). The processes communicate with each other through a UDP socket in a blocking and synchronous way (lines 8, 9, 9 and 22).

Listing 2: Slave process of the master-slave program

```
 1 // Slave 0 (Process 1 Thread 0)
 2 class Slave0 {
 3   public static void main(String[] args) {
 4     DatagramSocket socket = new DatagramSocket(4000);
 5     Shared.semaphore = new Semaphore(0);
 6     Slave1 consumer = new Slave1();
 7     consumer.start(); // thread creation
 8     for (int i = 0; i < 2; i++) {
 9       Shared.buffer = MySocket.receive(socket) + 1;
10       Shared.semaphore.release();
11     }
12     consumer.join();
13   } // main
14 }
15
16 // Slave 1 (Process 1 Thread 1)
17 class Slave1 extends Thread {
18   public void run() {
19     DatagramSocket socket = new DatagramSocket();
20     for (int i = 0; i < 2; i++) {
21       Shared.semaphore.acquire();
22       MySocket.send(Shared.buffer, socket, 5000);
23     }
24   } // run
25 }
```

Listing 3: MySocket implementation used in both master and slave processes

```
 1 class MySocket {
 2     public static void send(int token, DatagramSocket
           s, int port) {
 3        InetAddress ip = InetAddress.getLocalHost();
 4        byte[] buff =
              Integer.toString(token).getBytes();
 5        s.send(new DatagramPacket(buff, buff.length,
              ip, port));
 6     } // send
 7
 8     public static int receive(DatagramSocket s) {
 9        DatagramPacket pkt = new DatagramPacket(new
              byte[20], 20);
10        s.receive(pkt);
11        return Integer.parseInt(new
              String(pkt.getData()).trim());
12     } // receive
13 } // class MySocket
```

A. Challenges for the Control Flow Automation

The first automation challenge is related to the control flow reported at runtime and generated by the static analysis. method (line 2), for example, is used by both Master and Slave1 threads (i.e. in different parts of the source code) and its body cannot be ignored in the static analysis. Besides, the individual testing of and does not necessarily reveal potential synchronization and communication defects. The interaction among , and must also be taken into account in the testing of a concurrent program as a whole.

Indeed, for a better testing of concurrent programs, each method must be considered in the context in which it was called.

The structural testing of concurrent programs addresses those requirements by viewing the concurrent program as a PCFG that contains a single control flow. Each method call in the PCFG is represented by its method body. This representation makes each model element explicit and eases the generation of required elements based on the program's data, control and synchronization flows.

Some challenges related to coverage evaluation are raised. The testing tool must verify in trace files if each required element, generated statically, has been covered. Therefore, the runtime information must correspond to the statically generated information. This is not a trivial process once methods called multiple times must be considered individually, as addressed above.

A first solution would be to initially transform the program by applying an in-lining process to the actual program to expand each method to its body. The posterior PCFG generation and program instrumentation would be based on the transformed program and the information on PCFG would correspond directly to the information reported at runtime. However, the method considerably increases the size of the instrumented program and code transformation complexity.

Instead of such a transformation, our solution addresses this issue by introducing the hierarchical organization of *node ids* and PCFG in-lining. These concepts help the generation of unique *node ids* that can be reported statically and at runtime in a compatible way. Therefore both the size of the resulting instrumented code and the instrumentation process complexity are reduced in comparison to the previous solution and no actual code in-lining is required.

This approach reduces both the size of the resulting instrumented code and the instrumentation process complexity in comparison to the previous solution and requires no actual code in-lining.

The PCFG generation is based on the original code and separated into 2 steps. First, the static analysis translates each method body into a partial Control Flow Graph (CFG). Each method call or block of continuous instructions inside the partial CFG is mapped to nodes and labeled with a *partial node id* (a integer), which makes them unique within the partial CFG.

The second step, i.e. the PCFG in-lining, consists in recursive replacements of the method call nodes inside each initial partial CFGs (i.e. the and methods) by their partial CFG, which results in a CFG. The *partial node id* assigned to the method call node is added at the beginning of each node id of its partial CFG. Finally, PCFG is created based on the CFGs of all threads. Therefore, each *node id*, associated with the corresponding *thread id* and *process id*, is unique on the concurrent program.

Figure 1 shows the PCFG for the program in Listings 1, 2 and 3 after the application of the technique. Nodes 3.1, 3.2 and 3.3 from process 0 thread 0 and nodes 4.1, 4.2 and

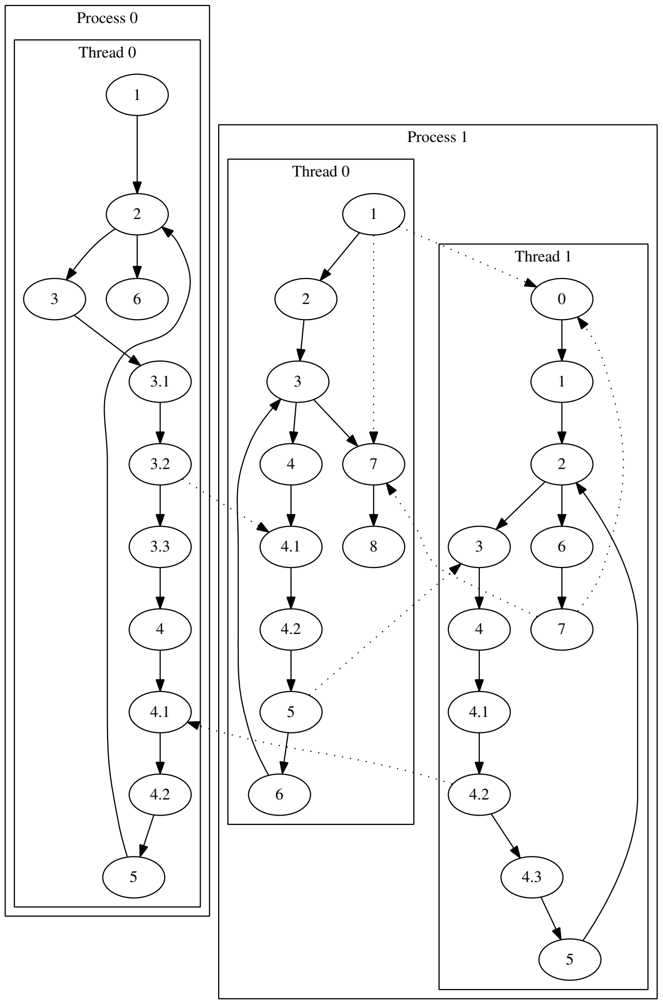

Fig. 1: PCFG representation of Listings 1, 2 and 3

4.3 from process 1 thread 1 represent the method body, while nodes 4.1 and 4.2 from process 0 thread 0 and nodes 4.1 and 4.2 from process 1 thread 0 represent . These nodes are composed of two numbers. The first denotes the partial node id of the method calls and the second represents the partial node id of the code blocks inside the method bodies.

Note (nodes 3.1 from process 0 thread 0 and 7.1 from process 1 thread 1) and (nodes 4.1 from process 0 thread 0 and process 1 thread 1) were replicated due to the in-lining process. This organization enables the derivation and evaluation of the required elements in the testing activity.

The instrumentation process enables a compatible *node id* generation by the insertion of instructions for the report of changes in the program control flow as events. This transformation causes no side effects to the tested program once it does not modify its original control flow.

method is inserted for each control flow change and method call.

Additionally, and are inserted before and after each method call. The two latter operations, respectively, push and pop a node id on a stack of *partial node ids*. Whenever a method is called, the instrumented program inserts (pushes) the method call *partial node id* in the stack and removes (pops) it after the return. Given the information provided by this stack, only the *partial node id* component must be reported in a new node event. The actual *node id* is generated at runtime through the concatenation of prefixes from the stack and the reported *partial node id* component.

As an example, consider the method call on line 8 of Listing 1. The result of this line instrumentation can be analysed in Listing 4. Firstly, the instrumenter signalizes a "new node" event with partial node "3" and then the "method call" event. Inside , the "new node" events are reported with partial node ids "1", "2" and "3". The actual node ids "3.1", "3.2" and "3.3" are generated based on this information at runtime.

The technique helps the testing of typical concurrent programs in cases whose synchronization and communication primitives must be analysed and are present in the method calls. Because the focus of the structural testing of concurrent program is on the testing of synchronization and communication among threads of a program, it is not the objective of this technique to give support to all aspects of programming languages, such as object orientation features. The full support of some language constructs, such as recursive calls, is not trivial for the structural testing of concurrent programs because the number of recursion levels is determined at runtime. In such a case, the tester can adopt a conservative approach and inform the number of recursion levels for each recursive method call in the PCFG generation and program instrumentation processes. The support to polymorphism in the testing activity is discussed in [17], [18].

Listing 4: Example of a control flow instrumentation

```
// Master main method (Process 0 Thread 0)
Visitor.visitNewNode("3");
Visitor.visitBeforeMethodCall();
MySocket.send(token, master, 4000);
Visitor.visitAfterMethodCall();

// Method Mysocket.send() instrumentation
class MySocket {
 public static void send(int token, DatagramSocket
      s, int port) {
  Visitor.visitNewNode("1");
  InetAddress ip = InetAddress.getLocalHost();
  byte[] buff = Integer.toString(token).getBytes();
  Visitor.visitNewNode("2");
  s.send(new DatagramPacket(buff, buff.length, ip,
       port));
  Visitor.visitNewNode("3");
 }
}
```

B. Challenges for the Data Flow Automation

The trace file and PCFG must contain data flow operations targeting uniquely identified memory positions. The identifiers of variables cannot be the only label component for the structural testing of concurrent programs, as two or more identifiers can reference the same object. Besides, the same method can be called multiple times which generates different memory positions.

We propose the data label attribution based on the allocation site to consider this scenario and support both the instrumentation and PCFG generation. A data label consists of *process id*, *thread id* from the selected graph and *node id*, where the data are allocated. All the components of the data label can be found in both run and static time. The definition and use of variables are mapped into operations over the composed data labels. Listing 5 gathers a sample of data flow information from the program in Listings 1, 2 and 3.

ValiInst concomitantly identifies data flow and control flow on both instrumentation and PCFG generation. All necessary information (e.g. *process id*, *thread id* and *node id*) is provided to the data operation identified until the parallel control flow ends.

The instrumentation performed for the data flow is also based on events. Instructions are inserted in the original source code for allocation, definition and use events to provide information in the trace files.

For example, when a variable is allocated, method is called with runtime information. At this point, the label is generated from current *node id*, *thread id* and *process id* and associated with the runtime information. Each operation over a variable identified by runtime information is reported as an operation over a previously created label.

Note the described process and inserted instructions do not modify variables of the tested program, which avoids undesirable side effects in the concurrent program behavior. The possibility of race conditions on the tested concurrent program is not eliminated by the instrumentation process.

To consider pointer handling, ValiInst ignores the program's control-flow graph and assumes instructions are executed in any order during the pointer analysis (or reference). The pointer analysis is also context-sensitive because it considers each method call individually. A more efficient static analysis may be implemented in a future version to improve the testing process.

C. Challenges for the Synchronization Flow Automation

Synchronization flow from concurrent Java programs consists of built-in language resources (e.g. synchronized blocks and methods) and classes, such as , and , which address different synchronization and communication paradigms and semantics.

The identification of such primitives cannot be entirely embedded in the testing tool's code. Indeed, a large number of primitives must be handled and others may appear in future versions of Java language. Besides, each primitive has its attributes, such as a cluster, a blocking nature (blocking or non-blocking) and a transmission nature (sender, receiver or sender-receiver).

Our solution is based on the development of a synchronization's specification that translates every primitive to testing model information. The resulting document contains all relevant information for the handling of methods or built-in features with semantics of synchronization and communication. The role of the testing tool, in this case, is to identify the primitives and generate PCFG and instrumented code based on the described properties.

The synchronization specification helps the implementation of flexible and extensible testing tools. It can explore synchronization operations compatible with the underlying model without changing the testing tool. This is an advantage in comparison to approaches that consider only built-in synchronization or implement support to a set of operations directly in the testing tools code. The analysis of concurrent programs assisted by the synchronization specification generates a uniform synchronization flow, reproducible statically and at runtime.

The instrumentation also modifies the concurrent program to enable the controlled execution. Instructions and , respectively, are inserted before and after the operation, with no modifications in their original semantics. As a result, the execution of selected synchronizations can be forced and a given execution can be replayed.

V. Evaluation

The following evaluation aims at the verification of consistency and performance aspects of the application of the proposed techniques. The ability of ValiPar to detect concurrent defects, as race conditions is related to the selection of a test model and testing criteria that address these problems. This subject, which is not the objective in this study, is presented and evaluated in [1], [2], [3], [4], [5] for the TestPar test model.

The consistency aspects are related to the correctness of the resulting set of structural information produced by the application of the techniques to concurrent programs of different complexities. The performance aspects are related to how these techniques affect instrumentation, PCFG generation and instrumented program execution in terms of execution time and size of instrumented files. This information helps in the determination of the viability of the techniques in the testing activity as a whole.

The validation involved eight concurrent Java benchmarks, which represent usual programming structures on concurrent Java programs of distinct complexity levels (see Table 1). The concurrent programs were developed according to a different number of threads and processes, number of bytecode instructions, synchronization and communication paradigms. The number of threads and processes ranged from 3 to 441 and 1 to 21, respectively. Both synchronization paradigms were considered and the number of bytecode instructions ranged from 244 to 1718.

Producer-Consumer program (PC) considers 50 producers and 50 consumer threads communicating through a shared buffer. Greatest Common Divisor (GCD) is a master-slave program that evaluates the GCD of three numbers. Token Ring program (TR) is a simulation of the ring network topology that uses both synchronization paradigms. Master-Slave program (MS) is described in Section IV. Non-blocking Message Passing program (NB-MP) represents a client-server program that uses non-blocking sends and receives considering communication events. Multithreaded Token Ring program (MT-TR) also simulates a ring network, however each process creates extra threads that modify a shared variable. Prime Server program (PS) is a numerical benchmark for the generation of prime numbers that uses a client-server approach. Finally, Matrix Multiplication program (MM) solves a matrix multiplication using processes and threads.

Each benchmark was submitted to ValiInst for PCFG generation and program instrumentation and the result were evaluated. The PCFG generated should contain all program flows, identify the data with unique labels, each node with unique node ids and translate the synchronization operations into a uniform representation. Furthermore, the execution time of ValiInst and the resulting instrumented programs were evaluated to show the length of these activities.

ValiInst and the concurrent programs were executed in an Intel R Core TM i5-4440 (3.10GHz) machine of 8 GB RAM, running Java SE 1.7 in an Ubuntu 14.04 Linux operating system. The tool could handle several combinations of parameters. The results are provided in Table 1.

Concerning the generation of structural information, our results show the instrumentation and PCFG generation produced consistent information for the test model and enabled effective verification of the coverage. The generation of structural information for the smaller program (NB-MP) produced 329 tuples, while the generation for the larger one (TR) produced 50,542. All the generated information is consistent with what was expected and shows ValiInst and the proposed techniques can produce a large volume of valid information.

To exemplify the ValiInst execution and the proposed techniques, consider Master-Slave program (at Listings 1, 2 and 3). The application of the proposed techniques is demonstrated in Listing 5, which shows part of the Slave1 program flow after the PCFG generation process. The program's control flow is correctly translated into a set of nodes and edges. Every node id, in conjunction with process id and thread id, is unique when the whole concurrent program is considered.

Listing 5: Sample of flows generated for the Slave1 thread from Master-slave. The program flow is composed of control flow edges (2 node ids), data flow operations (a node and a data label) and synchronization flow operations (node, semantics and cluster).

```
1 Control flow edges: (4, 4.1), (4.1, 4.2), (4.2, 5)
      ...
2 Definitions: (2, p1t0n1), (6, p1t0n1), (1, p1t0n1)
      ...
3 C-uses: (1, p1t0n1), (7, p1t0n1), (6, p1t0n1) ...
4 Synchronizations: (5, BS, SEMAPHORE) ...
```

The data flow is also translated into a correct set of data tuples. Every data tuple consists of an operation, a data label and the node/edge where it occurred. Data operation *Definition*

TABLE I: Detailed description of the concurrent programs adopted and impact of the instrumentation process. Programs adopted: Producer-Consumer (PC), Greatest Common Divisor (GCD), Token Ring (TR), Master-Slave (MS), Non-blocking Message Passing (NB-MP), Multithreaded Token Ring (MT-TR), Prime Server (PS) and Matrix Multiplication (MM).

Parameters	**PC**	**GCD**	**TR**	**MS**	**NB-MP**	**MT-TR**	**PS**	**MM**
Threads	101	4	441	3	3	13	7	13
Processes	1	4	21	2	3	4	4	5
Paradigm	SM	MP	Both	Both	MP	Both	Both	Both
Original Bytecode file size	1718	551	1170	239	244	710	403	602
Instrumented Bytecode file size	6815	1405	4333	773	773	1958	1387	1938
ValiInst Execution Time ms	1245	383	1948	337	330	529	521	551
Original Program Execution Time ms	64	42	232	2018	31,5	52	49	50
Inst. Program Execution Time ms	619	62	1410	2024	59	143	148	135
Data Flow Tuples	11935	571	30119	269	236	1344	707	1770
Control Flow Tuples	4279	176	16979	69	87	611	362	641
Sync Flow Tuples	701	12	3444	10	6	87	50	67
Total of Tuples	16915	759	50542	348	329	2042	1119	2478

(6 p1t0n1) (line 2 of Listing 5), in this context, represents the definition of a variable allocated in process 1, thread 0 and node 1 (forming the data label) at node 6.

Finally, the program's synchronization flow is correctly converted to generic operations with the actual synchronization and communication semantics. For example, represents a blocking send operation that matches other operations of *SEMAPHORE* cluster. The resulting tuple can be observed in line 4 of Listing 5. This synchronization operation occurred at node 5 of thread 0 of process 1 and consists of a (i.e. a blocking send operation) in the *SEMAPHORE* cluster.

Listing 6 shows part of the code of Slave 1 (described in Listing 2) after the instrumentation process. The inserted code (lines 1, 2, 4, 5 and 7) tracks and reports the program's flows. For ValiInst, the tracking process is conducted through the class, which receives the runtime information, translates it into model information and produces the trace considering the current process, thread and node ids. in Listing 6 reports the use and definition of variable , represented at runtime by its actual name and statically by its allocation site.

Listing 6: Sample of the instrumented code of a thread from Master-slave

```
1 ThreadVisitor.visitGetStaticObjField(semaphore);
2 ThreadVisitor.visitBeforeSend(1, "BS");
3 semaphore.release();
4 ThreadVisitor.visitAfterSend();
5 ThreadVisitor.visitNewNode("6");
6 i++;
7 ThreadVisitor.visitInc("Slave1.main(String)V", "i");
```

Listing 7 shows the trace file after the execution of the instrumented code in Listing 6. The first field of each line in Listing 7 contains the event type followed by the node where it occurred and a logical time stamp (which differentiates multiple executions of a certain node inside a loop, for example). Line 1 shows a send event that occured at node 5 of thread 0 in process 0 and whose time stamp is 141. The remaining fields are specific for each type of event. The synchronization events require operation (i.e. send, receive or send-receive) and cluster (i.e. semantic group of the operation) fields, while the data events require a data label field.

The trace file, in conjunction with statically generated model information (Listing 5), is used in the evaluation of the structural testing criteria. Each event reported corresponds to a tuple in the model information because of the application of the techniques.

Listing 7: Sample of the generated trace file of a thread from Master-slave

```
1 SendEvent,   <5^{0,0},141>, SEMAPHORE, BS
2 NodeEvent,   <6^{0,0},142>
3 UseEvent,    <6^{0,0},143>, CUSE, p0t0n1
4 DefinitionEvent, <6^{0,0},144>, p0t0n1
```

The operational cost of the instrumentation and PCFG generation can be summarized by three main aspects: time required for the generation of both PCFG and instrumented program, execution time of the instrumented program and number of additional bytecode instructions on the instrumented program. The total execution time of ValiInst module changed from 330 to 1948 milliseconds (Table 1). Moreover, the instrumentation process resulted in compiled files with 280% more instructions than the original ones (Bytecode (Original) and Bytecode (Inst.) rows in Table 1). Finally, the execution time of the instrumented program was, on average, 350% longer than the original one (Original program and instrumented program execution time rows in Table 1).

Both execution time and number of instructions increased because the structural testing activity requires the report of all data, control and synchronization operations to be conducted. Therefore, for each original operation, one or more operations are inserted and executed for keeping track of the application's current state. Although the testing activity is expensive, costs related to the execution time and number of instructions of the instrumented code do not affect the structural testing because the main focus is on the feasibility and availability of a higher coverage rate.

VI. Conclusions

The effective structural test of concurrent programs requires the instrumentation and PCFG generation of the whole concurrent program for the extraction and report of all program flows. Although several testing approaches can effectively test concurrent programs, none of their implementations as testing tools support the use of both synchronization and communication paradigms on the same program. Instructions on how to structure static and runtime information about the program flows are often not presented.

The techniques presented in this paper addressed such problems by describing a simpler but efficient instrumentation and PCFG generation processes and implementing them for the structural testing of concurrent Java Programs. The structure keeps the compatibility between the information in the trace file and the parallel control flow graphs, therefore, the coverage of selected test cases can be evaluated. The implementation of the techniques in ValiInst has shown our approach is effective when typical concurrent programs are tested.

The results show a stable behavior of ValiInst (even for larger programs), as all different Java concurrent programs could be analyzed and instrumented and the source codes were successfully represented in PCFGs.

Acknowledgment

The authors acknowledge Brazilian funding agencies FAPESP, CAPES and CNPq for their financial support (processes 2013/05750-8 and 2013/01818-7).

References

[1] P. S. Souza, S. S. Souza, M. G. Rocha, R. R. Prado, and R. N. Batista, "Data flow testing in concurrent programs with message passing and shared memory paradigms," *Procedia Computer Science*, vol. 18, no. 0, pp. 149 – 158, 2013, 2013 International Conference on Computational Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050913003219

[2] P. S. Souza, S. R. Souza, and E. Zaluska, "Structural testing for message-passing concurrent programs: anextended test model," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 21–50, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.2937

[3] F. S. Sarmanho, P. S. Souza, S. R. Souza, and A. S. Simão, "Structural testing for semaphore-based multithread programs," in *Proceedings of the 8th International Conference on Computational Science, Part I*, ser. ICCS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–346.

[4] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simo, and A. C. Hausen, "Structural testing criteria for message-passing parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 16, pp. 1893–1916, 2008. [Online]. Available: http://dx.doi.org/10.1002/cpe.1297

[5] A. C. Hausen, S. R. Verglio, S. R. S. Souza, P. S. L. Souza, and A. S. Simo, "A tool for structural testing of MPI programs," in *8th IEEE Latin-American Test Workshop (LATW 2007)*, Cuzco, Peru, 2007, pp. 1–6.

[6] A. T. Endo, A. S. Simao, S. R. S. Souza, and P. S. L. Souza, "Web services composition testing: a strategy based on structural testing of parallel programs," in *Testing: Academic Industrial Conference - Practice and Research Techniques (TAIC-PART 2008)*, Windsor, UK, 2008, pp. 3–12.

[7] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 206–215, Mar. 1992. [Online]. Available: http://dx.doi.org/10.1109/32.126769

[8] C.-S. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '98. New York, NY, USA: ACM, 1998, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/271771.271804

[9] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller, "Have: detecting atomicity violations via integrated dynamic and static analysis," *Lecture Notes in Computer Science (LNCS)*, vol. 5503, pp. 425–439, 2009.

[10] M. Christakis and K. Sagonas, "Detection of asynchronous message passing errors using static analysis," *Lecture Notes in Computer Science (LNCS)*, vol. 6539, pp. 5–18, 2011.

[11] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro, "Coverage testing of java programs and components," *Sci. Comput. Program.*, vol. 56, no. 1-2, pp. 211–230, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2004.11.013

[12] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," *IBM Syst. J.*, vol. 41, no. 1, pp. 111–125, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1147/sj.411.0111

[13] M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software," Microsoft Research, Tech. Rep. MSR-TR-2007-149, November 2007. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70509

[14] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[15] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of jml tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, 2005.

[16] S. R. S. Souza, M. A. S. Brito, R. A. Silva, P. S. L. Souza, and E. Zaluska, "Research in concurrent software testing: A systematic review," in *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2011), in conjunction with International Symposium on Software Testing and Analysis (ISSTA 2011)*, Toronto, ON, Canada, 2011, pp. 1–5.

[17] R. T. Alexander and A. J. Offutt, "Criteria for testing polymorphic relationships," in *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ser. ISSRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 15–. [Online]. Available: http://dl.acm.org/citation.cfm?id=851024.856208

[18] A. Rountev, A. Milanova, and B. Ryder, "Fragment class analysis for testing of polymorphism in java software," *Software Engineering, IEEE Transactions on*, vol. 30, no. 6, pp. 372–387, June 2004.