Distributed System Resource Racing Conditions Automated Testing Method

Robertas Jasaitis[1], Jonas Prapuolenis[1], Eduardas Bareiša[1]

[1]*Kaunas University of Technology, Studentų st. 50-406 LT-51368, Kaunas, Lithuania*

***Abstract* – Most applications today are designed to be networked. It is natural that such applications operate on some data. Such applications are multi threaded by nature and it is a common situation when few clients are using the same application at the same time. So it is possible that those users operate with the same data. It is possible that in such an operation some data might be treated incorrectly or some client instance may operate on outdated data. In order to avoid such situations proper consistent testing should be performed. Therefore it is complicated to perform such testing manually. Automated tool which would help to solve this problem is wanted. In this paper we are presenting an automated testing method that is able to detect problems related to resource racing conditions in a distributed system.**

***Keywords* – Distributed systems, resource racing conditions, automated testing.**

1. Introduction

Distributed system architects and developers spend much more time on the system architecture comparing to non distributed systems architects and developers. The same is true when talking about distributed and non distributed systems testing. This is because distributed systems are much more complicated by nature comparing to conventional non distributed systems. There are so many technologies dedicated for distributed systems development. We are focusing on Java based ones for our researches so only those will be considered in this paper.

Three of the most usable frameworks are described below:

Struts 2 - Apache Struts 2 is an elegant, extensible framework for creating enterprise-ready Java web applications. The framework is designed to streamline the full development cycle, from building, to deploying, to maintaining applications over time. [4]

JSF - JavaServer Faces (JSF) is a java based web application framework. JSF became a standard for development of server-side user interfaces for Java EE application. It uses a component based approach. JSF uses JavaServer Pages (JSP) as its display technology, but it can also support other technologies such as XUL and Facelets. [5]

Spring framework - Spring is a java based framework dedicated for modern enterprise applications development on any kind of deployment platform. The framework provides comprehensive programming and configuration model. “A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.” [3]

One may think that these frameworks should help developers and testers to make their products of better quality, but the reality is a bit different. The framework surely improves development performance but also brings in some complexity. All these frameworks have their benefits and drawbacks so there is no most popular and most commonly used one. Also there is no one unified testing method that would be able to test applications developed using different frameworks. That leads to naturally coming solution to get back to the roots of these frameworks.

As we mentioned we are focusing on Java based distributed system frameworks. All these frameworks are based on Java Enterprise Edition (Java EE) platform. The Java EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the application components that make up a Java EE application are installed on various machines, depending on the tier in the multitiered Java EE environment to which the application component belongs [2]. On a Java EE platform there are 4 main components that has their own place in the distributed system:

Client-tier components run on the client machine.

Web-tier components run on the Java EE server.

Business-tier components run on the Java EE server.

Enterprise information system (EIS)-tier software runs on the EIS server [2].

Such a distributed system might be visualized as displayed by Fig. 1

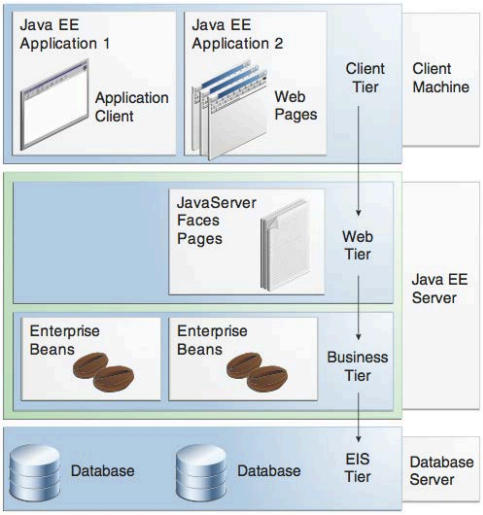

Figure 1.Multitiered distributed application [2].

Java EE distributed system might actually be a base platform for many existing Java based frameworks. So when talking about distributed systems testing, targeting such a system is logical and sensible decision. By doing so we are automatically targeting any framework created on a such base platform.

Now when we defined our target platform we can take a look at the main problems that arise while developing and testing such systems. Also what problems are targeted by the non distributed system automated testing algorithms and which are not.

Distributed system is a complex system by it's nature. Firstly such a system is concurrent by nature so sharing resources between system components becomes a complex task. Secondly such a system uses network communication for the resource sharing so automatically faces communication problems: communication problems while locating other component (for example server); timing problems when response comes too early or too late; incomplete or incorrect response when connection is damaged during communication and so on. So resource racing conditions are very common and also can lead to serious system functionality malfunction. That is why we chose to address resource racing conditions while distributed system components interact which each other over the network.

So to solve the problem the description of our distributed system interpretation should be defined firstly. There are number of different interpretations of what the distributed system is. In our context we chose to use the following distributed system definition.

A distributed system consists of a collection of autonomous components running on the same or different computers, connected through a network and distribution middleware, which enables those components to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility [1].

2. Intuition of the algorithm

As we mentioned earlier our main goal is to detect resource racing related problems when distributed system components communicates to each other over the network. In order to solve this task we split it into two main subtasks:

Distributed system components interaction between each other simulation covering all possible communication variations

Recourse racing conditions detection and problems related to these conditions detection

More details on each of the subtasks provided below.

What is a simulation of distributed system components interaction? What are the differences when solving this problem for the non distributed system and distributed system? These questions naturally arises when we start thinking about our earlier defined first subtask. In this section the answers to questions are provided in detail.

Firstly we start our explanation with simple concurrent application example. An example of such a system is provided by Fig. 2.

Thread 1	Thread 2

`void run() {`
`  print "[T1]";`
`}`
	`void run() {`
	`  print "[T2]";`
	`}`

Figure 2. Visualized concurrent application [13].

In the sample application there are 2 threads running concurrently. Each of the threads does as simple task as printing text. The Thread 1 prints [T1] while Thread 2 prints [T2]. In the application there is a natural racing condition. So we are never sure which text will be printed first. For some amount of tests the output can be [T1][T2], while for others [T2][T1]. This is obviously unpredictable and not acceptable for the testers. In order to cover all the paths through the application we may need unpredicted number of tests.

Such problems are often solved using model checking technique. This technique is able to detect such a racing situations and cover all paths through the application by controlling the application execution. If application provided by Fig. 2 would be passed to model checker algorithm the model checker would actually make an execution graph and execute it by covering all the paths. Possible execution graph of the sample application is provided by Fig. 3.

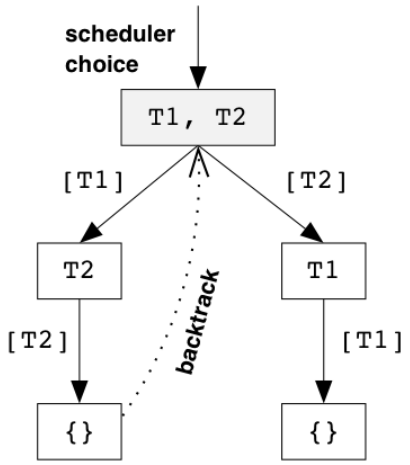

Figure 3. Concurrent application execution graph (space exploration) [13].

So the model algorithm detected racing condition marked by root node (T1, T2) and provides a choice situation. In case T1 is chosen, then the next choice is only T2 and application finishes its task. Then model checker detects that some other uncovered paths exists and executes "backtrack" operation and restores application to the choice condition node. Then the only choice is to choose T2 path which leads to T1 and application finishes its task. Model checked now knows that there are no more uncovered paths and finishes its work.

This is obviously not a problem to implement such a model checker for the non distributed systems as model checker is running on the same machine as all its components so it has control over all of those components. The situation in the distributed system case is totally different. The components of the distributed system are not under control of the same model checker so the model checker is not able to control all of the system components. Let us provide an example. We use an e-shop system as an example. Application deployment diagram is provided by Fig. 4.

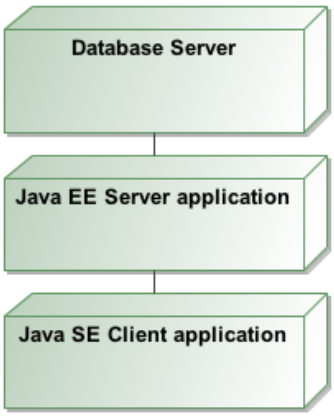

Figure 4. E-shop application deployment diagram

As we can see from the diagram the application client and server components are deployed on separate machines and communicate to each other via network. Our sample application is able to list goods to be sold on the client side and the client is able to buy the chosen item. Server side component is responsible for providing items to be sold and also mark an item as sold. The application use case diagram is provided by Fig. 5

Figure 5. E-shop application use case diagram

Principal implementation and possible execution graph is provided by Fig. 6. As we can see from the execution graph there is no possible resource racing condition.

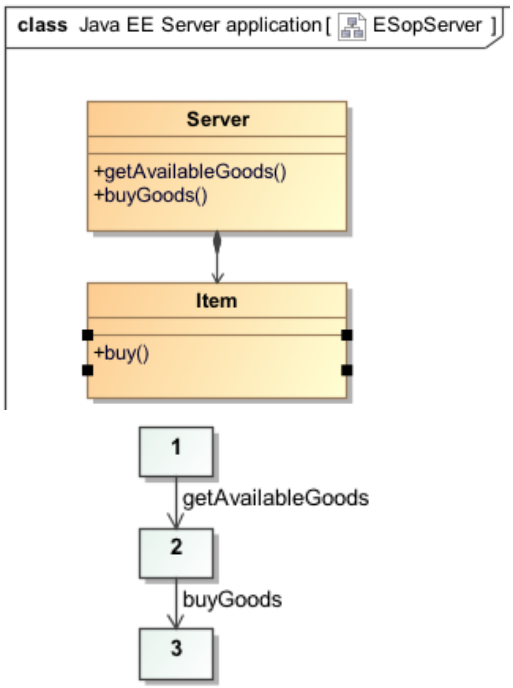

Figure 6. Principal server implementation class diagram (top) and possible execution graph (bottom)

What would happen if two or more clients would be interacting with the same server at the same time looking for the same item to buy? That would definitely lead to a situation where resource racing condition may happen. So the execution graph of the situation when two clients are interacting to the same system is provided by the Fig. 7.

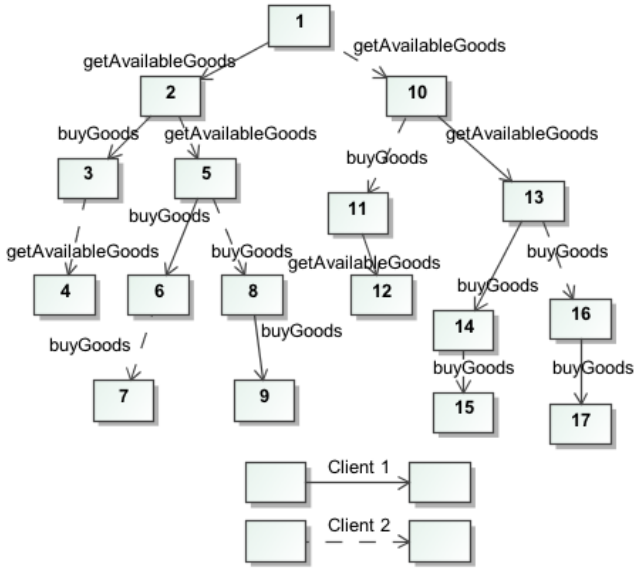

Figure 7. Execution graph of the situation when two clients are interacting to the same system

You may think that we can cover this case using earlier mentioned model checking algorithm but you would be wrong. It's because model checker algorithm works with single instance of the component (application) and has no control of the other instances of the same component. Also the model checker has no control on the remote components. So the question now is how to simulate such a situation and run model checker algorithm under the simulated situation.

Here we propose our algorithm to solve this problem. As the implementation we use Java Path Finder (JPF)[6] model checking tool developed by Nasa. But as we mentioned this model checking tool is not able to simulate two or more application instances and model check those instances interacting together. So our solution is to extend the tool. The tool should start two instances of the component and find split decision points. In our case these decision points are connection to other components calls.

As soon as model checker reaches the connection code in the application it marks it as a decision point. And it does the same for all the other connection code presence in the application. So as our model checker now knows when to make a decision how the model checker interacts with the other instance of the application communicating with the same remote component at the same time?

We solve this problem in a most simple and least expensive (in a perspective of model checker changes). All we do is we create two separate threads in the code. We make each of the thread to start the new instance of the application under test. Then we start both of the threads. So what we get is we have two application instances running concurrently under the same model checker using the same kind of decision points. All the rest is done by model checker. What we have to do now is to make the model checked detect problems in the code and notify the tester. All about this is provided in the next section as we came up to our second subtask.

So now we need to help model checker detect resource racing conditions and problems in the code related to that. There is no such tool at the moment that would be able to solve this totally automatically. Our proposed method is not an exception. But we tried to make a tool requiring least tester effort as possible. To achieve that we take an advantage of Unified Modeling Language (UML) class diagrams.

In our case UML class diagrams can be used for both: the client application components or the server application components. According to object oriented programming encapsulation concept "the data and the implementation code for the object are hidden behind its interface" [7]. "Encapsulation hides internal implementation details from users" [7].

The fragment of possible class diagram of our e-shop system server side component is provided by Fig. 6. Take a closer look at an Item class diagram. As we can see the class has "buyGoods()" method. This method is very important when implementing it because it is responsible for database change to mark the item as bought: unavailable for other buyers. So this method is directly related to application resources so naturally can lead to resource racing related conditions.

Imagine the following situation. User A launches our sample e-shop application. The user makes some search in the application and gets the list of available goods. One of available goods there is an item C. Then concurrently some other user B launches the same application on other computer and performs the same search. The user B gets the same list as the user A and so an item C is on both lists. What happens if both users decide to buy the same item C? The resource racing condition appears. And this is dependent on the application implementation whether the situation is handled correctly. Our task is to help developers in such situations and find possible faulty implementation parts. Ordinary testing can never lead to a situation when two clients communicating with the same remote components are trying to manipulate with the same resources.

In order to develop automated testing method which would be able to detect this type of failure we chose to use UML class diagram extension. The proposed extension should be as simple as possible, should not require significant architect input and should be compliant with existing UML 2.0 [8] or above modeling tools. To meet such requirements we chose to use method stereotypes as UML class diagram extension. We propose using new <<final>> stereotype for each setter method that could be called only once for the given value. In our case it's "buy()"

method of server component Item class. The Item class UML diagram with the stereotype is provided by Fig. 8.

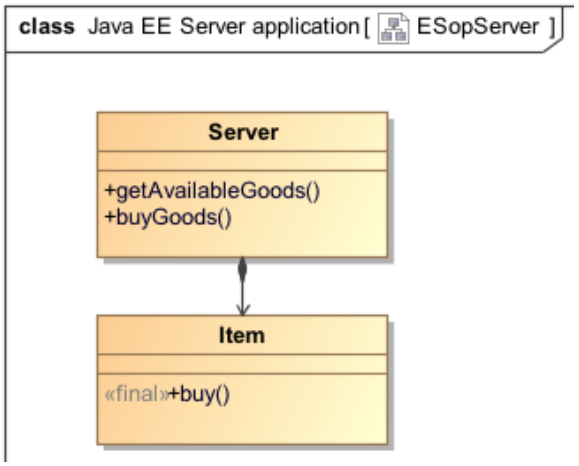

Figure 8. Server component class diagram

Having this new stereotype set in the diagram we can analyze the class diagram provided in XMI format [9] of the system under test and automatically generate additional constraints for the <<final>> method. The constraint checks if any value has been set for the given object. The test oracle: test passes in case some value is being set for the first time and fails vice verse. We will be using assertions provided by Java 1.5 and above for this purpose [10].

In addition to <<final>> stereotype we propose to use two more new setter method stereotypes: <<intermittent>> and <<unique>>.

<<intermittent>> failures usually happens when some value is consistently changing. For example a limited number of used discount coupons should always increase until it reaches maximum allowed value, while in a incorrect implementation the number returned which means current number of used coupons can be increased at the same time by two clients and then set on the server as that new number of used coupons (which is increased only by one while should be increased by two).

<<unique>>, as it name says, is used in a case when setter method can only be set to a value that have never been set before while executing the test. This can be useful when registering some objects using ids and the same objects (id) cannot be registered twice.

Using simple UML stereotypes is a convenient way for the system architect to ensure his or her design is resource racing safe. Also this kind of testing meats the Test Driven Development (TDD) principles that becoming more and more popular [11].

3. Implementation

Our method implementation is divided into two parts: model checker and the resource racing conditions detector. So we start with the model checker part.

As we mentioned in previous section our model checker is implemented as JPF tool extension. The system under test (SUT) is wrapped in two separate threads that just creates an application instances and start them concurrently. This is done by injecting new threads code in the application source. The code snippet that is used for this purpose is provided by the table 1.

Table 1. Threads injection code snippet.

From	To

public static void main(String[] args) {
/* Application entry code goes here */
}
	public static void main(String[] args) {
	Thread t1 = getMainRunnable(args);
	Thread t2 = getMainRunnable(args);
	t1.start();
	t2.start();
	}
	
	private static Thread getMainRunnable(final String[] args) {
	return new Thread(new Runnable() {
	@Override
	public void run() {
	/* Application entry code goes here */
	}
	});
	}

The other part of the implementation is the decision point detection in the SUT. These decision points are used by JPF model checker when executing the SUT. As we mentioned in the previous section we use communication with remote components code as decision points. So whenever JPF detect the following code in the SUT it knows it is a decision point. These decision points later are bookmarked in the JPF virtual machine state and so JPF is able to get back to the decision points when needed. All the other model checking is done automatically by the JPF model checker.

URL example = new URL("http://www.example.com/");
URLConnection connection = example.openConnection();

The resource racing condition detection relies on the code generation in the SUT depending on the methods and stereotypes found in the SUT class diagrams.

For setters and getters used in system under test (and of course in the system class diagrams) we require to use JavaBean setters and getters naming convention [10]. We need to know about the naming convention because it is needed for test generation. JavaBeans are widely used worldwide and this naming convention is actually usually used in Java applications.

The implementation of test generator activity diagram is provided by Fig. 9.

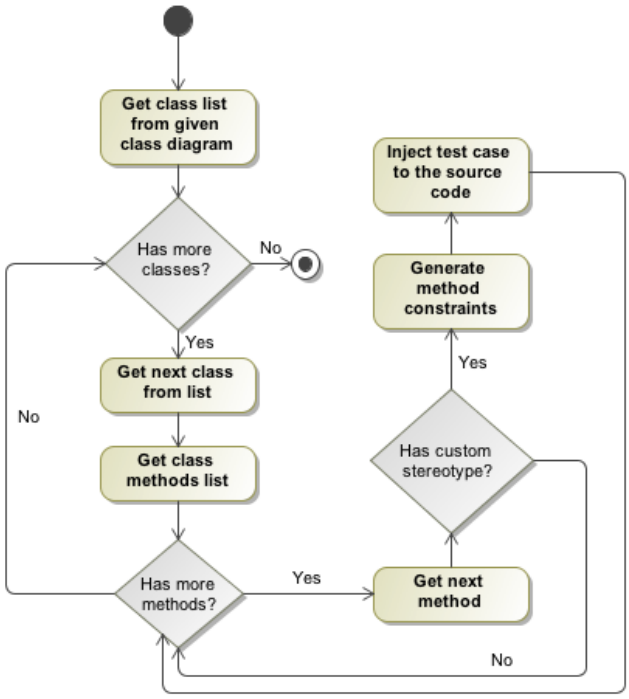

Figure 9. Test generator activity diagram

Code constraint generation activity depends on the stereotype applied for the method. The pseudo code which is generated for each of the defined stereotypes is provided in table 2. We use <methodName> key in the code which means the method name without "set" suffix and lower case first letter. So the method "setReserved" would become "reserved". We also use <getter> key which is the getter of the setter method that we are analyzing as per JavaBean specification. And finally we use <parameter> key which means the object's or primitive type's parameter name which is passed in a setter method.

Table 2. Method constrains pseudo code

Stereotype	Generated pseudo code
<<final>>	private static Vector<Object><methodName> = new Vector<Object>(); <method header taken from source code> { assertFalse(<methodName>.contains (this)); <methodName>.add(this); <method body from source> }
<<intermittent>>	<method header taken from class diagram> { assertTure(!this.equals(this.<gette r>)); <method body from source> }
<<unique>>	private static Vector<Object><methodName> Values = new Vector<Object>(); <method header taken from source code> { assertFalse(<methodName>Value s.contains(<parameter>)); <methodName>Values.add(<para meter>); <method body from source> }

Note that each class that contains methods with any of given new stereotypes should override equals method from Java Object class.

When the test case code is injected in the source code the application is compiled. After that we launch our distributed system model checking algorithm which now uses the source with the test cases and simulating two client model checker execute all possible clients and servers communication paths.

4. Experiments

Two experiments were performed to analyse the efficiency of test method. We used the seat reservation system mentioned in previous work and a simulated ATM-bank client server application. Automated test method results was compared to basic JUnit tests that could be written manually by a tester. Test method results are gathered by generating mutants of these applications and analysing how much of them both tests detected. Parameters of tested applications are provided in table 3. Note that

we are only analysing server side of tested applications, therefore only information about server side is given.

Mutants were generated by modifying server source code in following ways:

Deleting assign statement or switching variables;

Replacing arithmetic operation one with another;

Replacing boolean relation with inverted value;

Table 3. Mutation testing summary

Test application	**Seat reservation system**	**ATM-bank system**
Lines of code	219	318
No. of classes	5	6
No. of classes covered by testing	2	3
No. of lines of code in these classes	98	149
No. of generated mutants	14	17
Applied stereotypes	<<final>> once	<<intermittent>> twice; <<unique>> once

As shown in table 3 mutation testing revealed that our automated test method finds about half as much mutants as does a manually written JUnit test.

By analyzing test results further we found that some automatically detected mutants did not trigger exception of stereotype. Instead they failed in some other ways including NullPointerExceptions, NegativeArraySizeExceptions or others. Exceptions that occurred on client and on server during automated testing have been separated from exceptions caused by applied stereotypes and are also shown in Fig. 10.

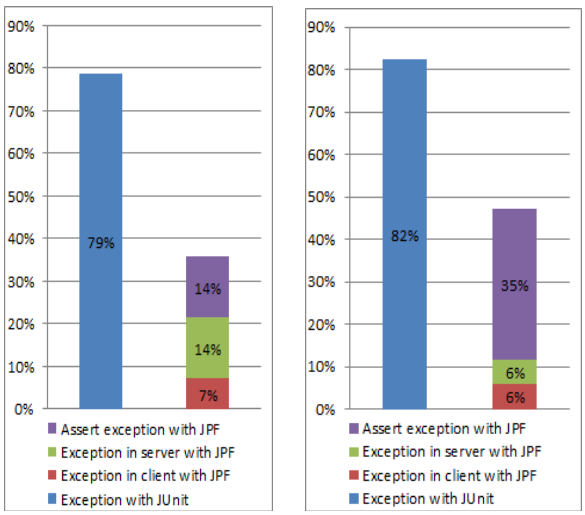

Figure 10. Percent of mutants detected by automated test method and JUnit

It is obvious that JUnit detected as twice more code mistakes compared to our suggested method. But this is an expected result as JUnit is a universal manual testing method and our suggested method is automated resource racing conditions detection method. Not all mutants caused resource racing issues so they were not detected by our proposed testing method.

It is very hard to compare time required to create JUnit test to our automated tests as manual test creation time depends on the engineer who is writing tests. But it is obvious that manual testing requires much more resources compared to automated testing.

We noticed that automated test method tends to fail to find mutants that have had arithmetic operations replaced. For example <<intermittent>> stereotype did not detect that bank account balance increased when user was withdrawing money, but JUnit test successfully detected the error. However this was not an error caused by resource racing.

5. Limitations of the method

The only problem of the method that we can think of is its dependency on the correctness of the UML class diagram. If the class diagram is incorrect in any form the correct implementation of the code might be treated as faulty. So software architects should be very careful when deciding what proposed stereotypes to use on the methods of the class diagram.

6. Conclusions and future work

Currently we have used two threads creation method to simulate two application instances. Although according to JPF specifications it is not the best and most correct way to achieve this. The problem with this method is that some data may be mixed between those threads which should never happen when running two separate clients. The data may be mixed in case of use static class variables or similar situations. The correct way to achieve this is to rewrite the depth first search (DFS) method of JPF and use custom choice generators. The rewritten DFS method should be clever enough to be able to jump between two separate clients states in the specified order and timing. This is not a trivial task and this was not realized yet and hence it's in our future work list.

References

[1]. Sabu M. Thampi. *Introduction to Distributed Systems.* L.B.S Institute of Technology for Women. Trivandrum, Kerala, India-695012.

[2]. *Java EE Technical Documentation*. Access via the Internet: http://docs.oracle.com/javaee/

[3]. *Spring Framework*. Access via the Internet: http://www.springsource.org/spring-framework

[4]. *Apache Struts 2*. Access via the Internet: http://struts.apache.org/release/2.1.x/index.html

[5]. *JavaServer Faces Technology*. Access via the Internet: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[6]. Nasa. *Java PathFinder Wiki*. Access via the Internet: http://babelfish.arc.nasa.gov/trac/jpf/ wiki

[7]. Oracle. *Object-Oriented Programming Concepts*. Access via the Internet: http://docs.oracle.com/javase/tutorial/java/concepts/object.html

[8]. Object Management Group (OMG). *UML Specification*, Version 2.0, 2010

[9]. Object Management Group (OMG). *MOF 2.0/XML Metadata Interchange Mapping Specification*, Version 2.1.1, 2007

[10]. Brian A.M., Jeffrey M.V. *Programming with Assertions: A Prospectus. IT Professional*, 2004. 6(5): p. 53-59

[11]. Kent Beck, Addison-Wesley Longman. *Test Driven Development: By Example*, 2002, ISBN 0-321-14653-0, ISBN 978-0321146533

[12]. Oracle. *JavaBeans Spec*. Access via the Internet: http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

[13]. Artho C., Leungwattanakit W., Hagiya M., Tanabe Y. *Efficient model checking of networked applications.* M. Hagiya, and Y. Tanabe. In Proc. TOOLS EUROPE 2008, volume 19 of LNBIP, pages 22–40, Zurich, Switzerland, 2008. Springer

Corresponding author: Robertas Jasaitis
Institution: Kaunas University of Technology, Lithuania
E-mail: jasaitis.robertas@gmail.com