日本ソフトウェア科学会第 26 回大会 (2009 年度) 講演論文集

Abstract Model Checking of Web Applications Using Java PathFinder

Vinh Cuong Tran Yoshinori Tanabe Masami Hagiya

Due to the interleaving of clients and servers, verifying web applications is a hard task. Bugs may occur only on particular scenarios, but testing all of them manually is almost impossible. To overcome the difficulty, we propose a framework for source code model checking of web applications. We use abstraction techniques to avoid state explosion. When model checking is conducted, various libraries including those for database access are replaced with their abstract versions so that the code of the target application can be verified without modification. Web clients are replaced with driver classes, which automatically generate possible scenarios with the help of the model checker. In this paper, we report a first verification attempt based on the proposed framework with a small web application built on the J2EE framework. We create abstract classes for servlets, HTTP requests and responses, SQL result sets, and so on, and the driver is executed by the model checker Java PathFinder to test all possible execution scenarios.

1 Introduction

With the rapid spread of the internet, more and more systems have been re-engineered into web applications (WA), such as online banking systems and e-payment systems. Since WAs are multi-threaded by nature, model checking is a powerful technique for verification of WAs. In this paper, we propose a framework for verification of Java WAs at the source code level. Specifically, we deal with WAs that are implemented using the J2EE servlet technology [2].

Although much work has been done on model checking of WAs, most of the previous work focuses on verifying the relationship among web pages in a WA [7][3].

MCWEB [3] models web pages in a WA with a graph called *webgraph*. Every node in the webgraph represents a web page with edges, each of which is labeled with the name of a frame or an anchor. MCWEB can analyze path properties of pages in the WA, such as the connectivity of pages, i.e., whether there is a path from every page to the home page, and cost properties, e.g., the number of pages to be downloaded.

The work by Donini, et al. [7] employs Computation Tree Logic to verify the design of a WA written in UML using the SMV model checker [8].

In contrast to the above approaches, our framework directly verifies Java implementation of a WA using Java PathFinder (JPF), which is a general-purpose extensible model checker for Java bytecode [11]. When model checking is conducted, various libraries including those for J2EE servlets are replaced with their abstract versions, while the source code of the WA is not changed. The abstract versions of the servlet libraries automatically analyze possible links between servlets. Web clients are replaced with driver classes, which automatically generate possible scenarios with the help of the model checker.

As far as we know, the most extensive work on verification of WAs was done in the development of the SAVE environment by Fujitsu Ltd [9]. In their environment, a WA is model-checked together with drivers and stubs, where the drivers are automatically generated to cover a suitable set of user inter-

Java PathFinder を用いたウェブアプリケーションの抽象モデル検査

チャン ヴィンクオン, 田辺 良則, 萩谷 昌己, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo

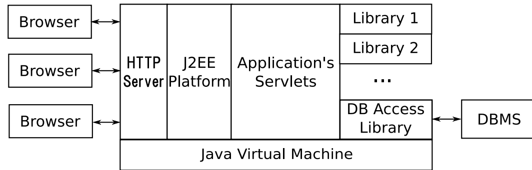

Fig. 1 WA deployed in the J2EE framework

actions and the stubs implement external functions such as databases and file systems. Their verification relies on symbolic execution of JPF, which was recently introduced [4].

On the other hand, we only use the ordinary finite model checking of JPF, which is stable and efficient. Abstraction is realized in the abstract versions of libraries, including the drivers and the stubs, e.g., the abstract library for database access. For each kind of abstraction, we prepare the abstract libraries that realize the abstraction. For example, in the case study presented below, we realize the specific-generic abstraction to verify the behavior of a student in a report-submission web application. In our framework, although the source code of the WA is not changed, the entire application can be model-checked using JPF, yielding a finite state space.

2 Framework

The structure of a WA, which is implemented using the servlet technology, can be depicted as in Figure 1. A WA basically consists of a set of classes called servlets, which usually extend the HttpServlet class in the J2EE framework. Clients can access the WA, which is deployed in a web server, through applications such as browsers and applets. Between the WA and the web server is the J2EE platform, which provides a container for servlets [1]. The container provides environmental information in the web server, extracts inputs from a client, and sends outputs to the web server on behalf of servlets. Beside the libraries provided by J2EE, the WA may use other libraries, e.g., the JDBC for accessing a database management system. All the application classes and framework classes on the server side are executed by a normal Java Virtual Machine (JVM), which is usually called *host JVM.*

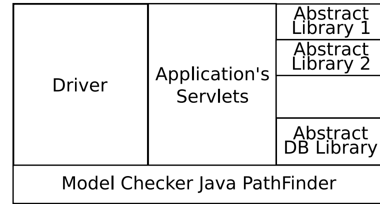

Fig. 2 WA verified by Java PathFinder

When we verify the WA by JPF, we replace the libraries in the J2EE framework and other libraries including the library for database access with the corresponding abstract libraries. Note that the source code of the servlets of the WA is not touched for verification.

We explain the abstract libraries used for the case study in the next section. They realize the specific-generic abstraction as we mentioned in the previous section.

In order to verify a WA, it is necessary to cover all possible scenarios of interactions between the clients and the server. We gather the clients (browsers), the HTTP server and the J2EE platform in Figure 1 into one system, called the driver, which simulates all (or more than) possible scenarios between the clients and the server, and behaves as the abstract container, by calling the servlets of the WA and interpreting (abstract) outputs from the servlets.

The entire system is then executed for verification on a customized JVM, which is a part of JPF.

In summary, the proposed framework uses Java as the only language both for implementation and abstraction, and uses JPF as the only tool. We hope that this monolithic nature will make the framework practical.

3 Case Study

To show the rationale behind our approach, we choose to verify a simple real-world report-submission system. The system allows students to submit reports, and a teacher to grade submitted reports. Students' data and submitted reports are stored in a database table. After login, the teacher gets a list of reports which have been submitted but not graded. The fragment of code in Listing 1 is an implementation of a servlet that displays the

Listing 1 A fragment of a teacher's servlet

```
public class TeacherHomeServlet extends HttpServlet {
  protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
          throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      try {
        out.println("<html>");
        out.println("<head>");
        out.println("<title先生のページ></title>\n");
        out.println("</head>\n");
        out.println("<body>\n");
        out.println("<h2> 先生のページ</h2>\n");
        try {
          String query="SELECT S.stuid FROM stmana S "
          + "WHERE S.grade='-1'";
          Object flag = getServletContext().getAttribute("flag");
          if (flag != null && (Boolean)flag ) {
            ResultSet result = executeQuery(query);
            while (result.next()) {
              String stuid = result.getString(1);
              out.println("<a href=\""
              +response.encodeURL("gradeReport?sid="+stuid)
              +"\">"+stuid+"</a><br>");
            }
            out.println("<br>");
            result.close();
          }
          else{
            out.println("レポートが提出されていません<br>\n");
          }
          out.println("<a href=\""
          + request.getContextPath()
          + "/logoutログアウト\"></a><br>\n");
          out.println("</body>");
          out.println("</html>");
        } finally {
          out.close();
        }
      } catch (SQLException ex) {
        Logger.getLogger(TeacherHomeServlet.class.getName())
        .log(Level.SEVERE, null, ex);
      }
    }
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
      processRequest(request, response);
    }
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
      processRequest(request, response);
    }
    public String getServletInfo() {
        return "Teacher Home servlet";
    }
}
```

reports that have not been graded.

Beside the default detection of JPF for deadlocks, uncaught exceptions, etc., we want to check if there is any student whose report is not graded when the student list displayed to the teacher is empty and all students have finished submission. To reduce the size of state space significantly without overlooking possible bugs, we employ the specific-generic abstraction explained in the next section.

4 Abstraction

Abstraction is not only useful but also crucial in model checking [5], because it can reduce an infinite state space into a finite one. In this work, we apply abstraction to the driver and the libraries in the J2EE framework and for database access.

4.1 Specific-Generic Abstraction

In the report-submission system, since there can be any number of students who submit reports and there can be any number of reports submitted, the state space of the system is virtually infinite without abstraction.

The specific-generic abstraction is a method that focuses on one particular element of a set, which is called the Specific object, and maps the remaining elements to one object, which is called the Generic object. In the report-submission system, we want to check if there exists any student whose report has been submitted but has not been graded yet. The specific-generic abstraction is applicable because the specific student that the method focuses on is represented by the Specific Object, while other students are represented by the Generic object. A set of students is then abstracted to a set of the Specific object and the Generic object.

4.2 Abstract classes in the report-submission system

Whenever the teacher queries for reports that have not been graded, there are two possible cases. The first case is that the specific student has not submitted a report, or the report of the specific student has already been graded. Thus the result of the teacher's query does not contain the report of the specific student. The second case is that the specific student has submitted a report and the report has not been graded yet. In both cases, the generic student may or may not have submitted a report. Thus, we need only two states as the contents of the abstract version of the database.

The (concrete) database access class is abstracted to a class, which has the same interface as the concrete one. The `executeQuery` method of the abstract database access class checks whether a query is UPDATE, INSERT or SELECT, and then takes appropriate actions on the database. For a SELECT query, one of the two objects of a class implementing the `java.sql.ResultSet` interface is returned. Each object corresponds to the two states of the database. The object is then returned to the servlet that makes the SELECT query. Since

Listing 2 A class implementing ResultSet

```
public class ResultSetWSpecific implements ResultSet {
  enum State {BOTH, GENERIC_ONLY, EMPTY };
  private State curState;
  private String returnString;
  public ResultSetWSpecific() {
    this.curState = State.BOTH;
    this.returnString = null;
  }
  public String getString(int columnIndex) {
    if (columnIndex > 1)
      assert false;
    return returnString;
  }
  public boolean next() throws SQLException {
    boolean ret = curState != State.EMPTY;
    switch (curState) {
    case BOTH:
      int nextState = Verify.getInt(0,2);
      switch (nextState) {
      case 0:
        returnString = genericString;
        break;
      case 1:
        returnString = specificString;
        curState = State.GENERIC_ONLY;
        break;
      case 2:
        returnString = specificString;
        curState = State.EMPTY;
        break;
      default:
        assert false;
      }
      break;
    case GENERIC_ONLY:
      if (Verify.getInt(0,1) == 0) {
      } else {
        curState = State.EMPTY;
      }
      returnString = genericString;
      break;
    case EMPTY:
      returnString = null;
      break;
    }
    return ret;
  }
}
```

these abstract classes conform to the J2EE API, we do not need to rewrite the source code of the WA. Listing 2 shows one of the classes that implement the `ResultSet` interface.

After receiving the abstract result of the query, the servlet calls the `next` method on the result to retrieve its elements. Each element contains a student as the owner of the report, and the student is determined by a transition system depicted in Figure 3.

The servlet then generates the output to the client by calling the `print` and `println` methods on the writer associated with the response of the servlet. The writer class is also replaced with an abstract class for verification. The `print` and `println` methods of the abstract class parse the output on-the-fly to get the relevant information for building the input to the next servlet.

Unimportant inputs to servlets are abstracted to a dummy string, and outputs from servlets are parsed on-the-fly as explained above and immediately discarded after relevant information is obtained.

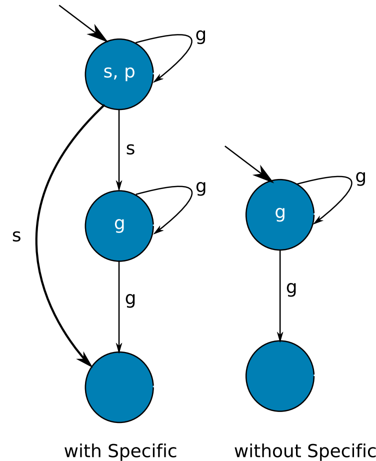

Fig. 3 Transition System of Abstract Student Set.

5 Result

We prepared two versions of the application. In one version, a bug was implemented so that in some sequences of submission, a report may not be graded by the teacher. In the other version, the bug was fixed. Our system detected the bug in the first version almost immediately.

For the second version, the model checker ran for four minutes and thirty-four seconds to explore all 2360 states, and reported that there was no bug remained. The experiment was conducted on a laptop PC with PentiumM 1.7GHz CPU with 1GB memory.

6 Future work

Through the case study, we have identified several pieces of work that are needed to be done before model-checking real-world WAs in the proposed approach.

In the case study, scenarios of interactions between the clients and the server are generated, as the output from a servlet is parsed to build the input to the next servlet. Although the generation of scenarios was automatic, their coverage depended on an initial setting, which specifies the order of execution of servlets, the number of threads, and so

on. Methods to help define an appropriate initial setting need to be investigated.

As for abstraction, we need to identify and provide those kinds of abstraction that are useful for verifying WAs. Automation of abstraction is also possible as in Bandera [6]. In particular, we need a method to automatically abstract a database by specifying how to abstract attributes of the database.

In the current framework of abstraction, in which concrete classes are replaced with abstract classes, we need to replace basic data with objects in order to apply abstraction to them. We can also use symbolic execution of JPF for integers [4].

However, even more and more advances are made in abstraction, it is not possible to verify all existing WAs in our framework. Instead, we plan to provide a well defined set of concrete/abstract libraries together with design patterns of WAs, so that WAs developed by the design patterns can be verified with the abstract libraries without modification of their source code.

In our previous work, we implemented the LTL library to make it possible to verify LTL properties using JPF [10]. We can immediately apply our LTL library to the proposed framework to verify liveness properties of such as "if a student submitted report, then that report will eventually be graded". We can also use the LTL library to avoid errors caused by lack of fairness in thread scheduling.

References

[1] Java platform, enterprise edition. `http://java.sun.com/javaee/technologies/javaee5.jsp`.

[2] Java servlet technology. `http://java.sun.com/products/servlet/`.

[3] Luca De Alfaro. Model checking the world wide web. In *Computer Aided Verification*, pages 337–349. Springer-Verlag, 2001.

[4] Saswat Anand, Corina Pasareanu, and Willem Visser. JPF–SE: A symbolic execution extension to java pathfinder. In *TACAS*, volume 4424 of *LNCS*, pages 134–138. Springer, 2007.

[5] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[6] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *In Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM Press, 2000.

[7] Francesco M. Donini, Marina Mongiello, Michele Ruta, and Rodolfo Totaro. A model checking-based method for verifying web application design. *Electr. Notes Theor. Comput. Sci.*, 151(2):19–32, 2006.

[8] K. L. McMillan. The SMV system. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[9] Sreeranga P. Rajan, Indradeep Ghosh, Oksana Tkachuk, Mukul R. Prasad, Praveen K. Murthy, Ryusuke Masuoka, Tadahiro Uehara, Kazuki Munakata, Kenji Oki, and Hirotaka Hara. Software applications validation environment: SAVE. In *Fujitsu Scientific & Technical Journal (FSTJ)*, volume 43, 2007.

[10] Vinh Cuong Tran, Hideki Hashimoto, Yoshinori Tanabe, and Masami Hagiya. Verification of java programs under fairness assumption. In *25th Japan Society of Software Science and Technology Annual Symposia*, 2008.

[11] Willem Visser, Klaus Havelund, and Guillaume Brat. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12, 2000.