Secure Provision and Consumption in the Internet of Services

FP7-ICT-2009-5, ICT-2009.1.4 (Trustworthy ICT)

Project No. 257876

www.spacios.eu

Deliverable D2.2.2
Combined black-box and white-box model inference

Abstract

This deliverable presents how ASLan++ models are extracted from Java source code. It then describes how to combine black-box and white-box model inference by using information extracted from source code, such as the alphabet of input/output events, to guide learning. As an additional development, we introduce an approach that compares the white-box model with one obtained by deep web crawling, identifying executions that bypass the presentation layer and could denote vulnerabilities.

Deliverable details

Deliverable version: *v1.0* | Classification: *public*
Date of delivery: *31.03.2013* | Due on: *31.03.2013*
Editors: *IeAT, UNIVR* | Total pages: *32*

Project details

Start date: *October 01, 2010* | Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT/TUM, UNIGE, SAP, Siemens, IeAT

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

Contents

List of Figures

1 Introduction

Black-box and white-box model inference techniques both have specific strengths and weaknesses. They are complementary in that we can counter-balance one's limitations with the other one's strengths in order to obtain better, more precise models, in a less costly manner.

We first describe in detail our method for model extraction from source code. Then we present two approaches to combine it with black-box model inference, for two different purposes. First, we can interface the two methods and use the white-box approach to provide information to black-box model inference, such as the alphabet of input/output events, and oracle information. Additionally, we compare a black-box model learned by crawling with a source-extracted model in order to detect potentially unintended behaviors and vulnerabilities.

2 White-Box Model Extraction

This section describes jModex, a tool for white-box extraction of behavioral models for JSP/Servlet web applications. Models are first generated in an extended finite state machine format and then converted into ASLan++. jModex attempts to combine the effects of statements as much as possible, following the idea of large-block encoding [1]. In short, it captures the behavior of the system by recording the execution conditions and state changes on every distinct execution path through the analyzed program.

This section is intended as a standalone description of jModex, including both previously implemented parts and updates representing the current development status of the tool. Therefore the exposition includes fragments already included in other deliverables, e.g., [4, 5].

2.1 jModex Components

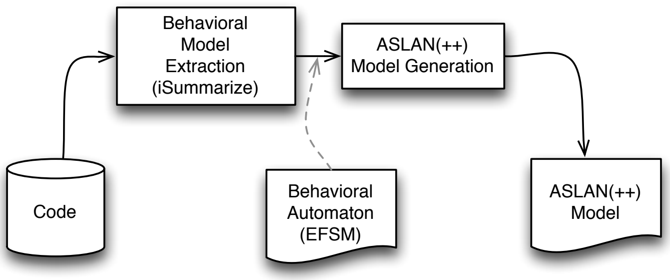
Behavioral Model Extraction (iSummarize)
ASLAN(++) Model Generation
Code
Behavioral Automaton (EFSM)
ASLAN(++) Model

Figure 1: jModex Components

Figure 1 presents the two main components of jModex, which correspond to the two main steps of the ASLan++ model extraction process:

- iSummarize - starting from the application code, a behavioral model (called iSummarize model) of the application is extracted in the form of an extended finite state machine (EFSM). The obtained model is further used by the second jModex component, but it can also be used by other tools as shown in different sections of this deliverable.
- ASLan++ Generator - using a set of translation schemas, this component translates the iSummarize model into an ASLan++ model of the application.

2.2 iSummarize - Capturing Execution Paths

iSummarize analyzes each method from the investigated application recording the updates of relevant state variables and the conditions under which these updates are performed. As a result, it builds an EFSM for the analyzed method. Figure 2 exemplifies in more detail the extraction process of the iSummarize model for the function shown in Listing 1.

Listing 1: A Source Code Example

```
1  public void _jspService(HttpServletRequest request,
2                                       HttpServletResponse response)
3                                       throws IOException, ServletException {
4          ServletOutputStream out = response.getOutputStream();
5          String aVar = "No Link!";
6          out.println("<html>");
7          out.println("<body>");
8          if(request.getParameter("update").equals("false")) {
9                  out.println(aVar);
10         } else {
11                 request.getSession().setAttribute("seen", "true");
12                 out.println("<a href=\"B.jsp?name=exec\"");
13                 out.println("Click here!");
14                 out.println("</a>");
15         }
16         out.println("</body>");
17         out.println("</html>");
18 }
```

The process starts by building the control-flow graph of the analyzed method, using the WALA analysis infrastructure[1]. Next, the graph is traversed in a depth-first order and the various execution paths are determined and analyzed instruction-by-instruction in reverse execution order.

Initially, at the end of the analyzed method, the final state of the method's EFSM is created. Going backward, on encountering an assignment to a relevant state variable, the update is recorded on every possible execution path leading from the assignment to the method exit/final state. Figure 2A shows the status of the iSummarize model construction after analyzing lines 16 and 17. For some analysis goals, capturing the output produced by a web application might be relevant. Thus, iSummarize records these updates and, consequently, it determines that by executing the *_jspService* function the strings "</body>" and "</html>" will be sent to the user.

When a join point is encountered (e.g., immediately before line 16), the paths/transitions going from that point to the final state of the method must be propagated backwards through each possible joining branch. As a result, the single transition from Figure 2A will be duplicated: the two copies will record the constraints/updates on the *false* branch of the *if* statement from line 8 and on the *true* branch, respectively. Figure 2B shows the iSummarize

[1] http://wala.sourceforge.net

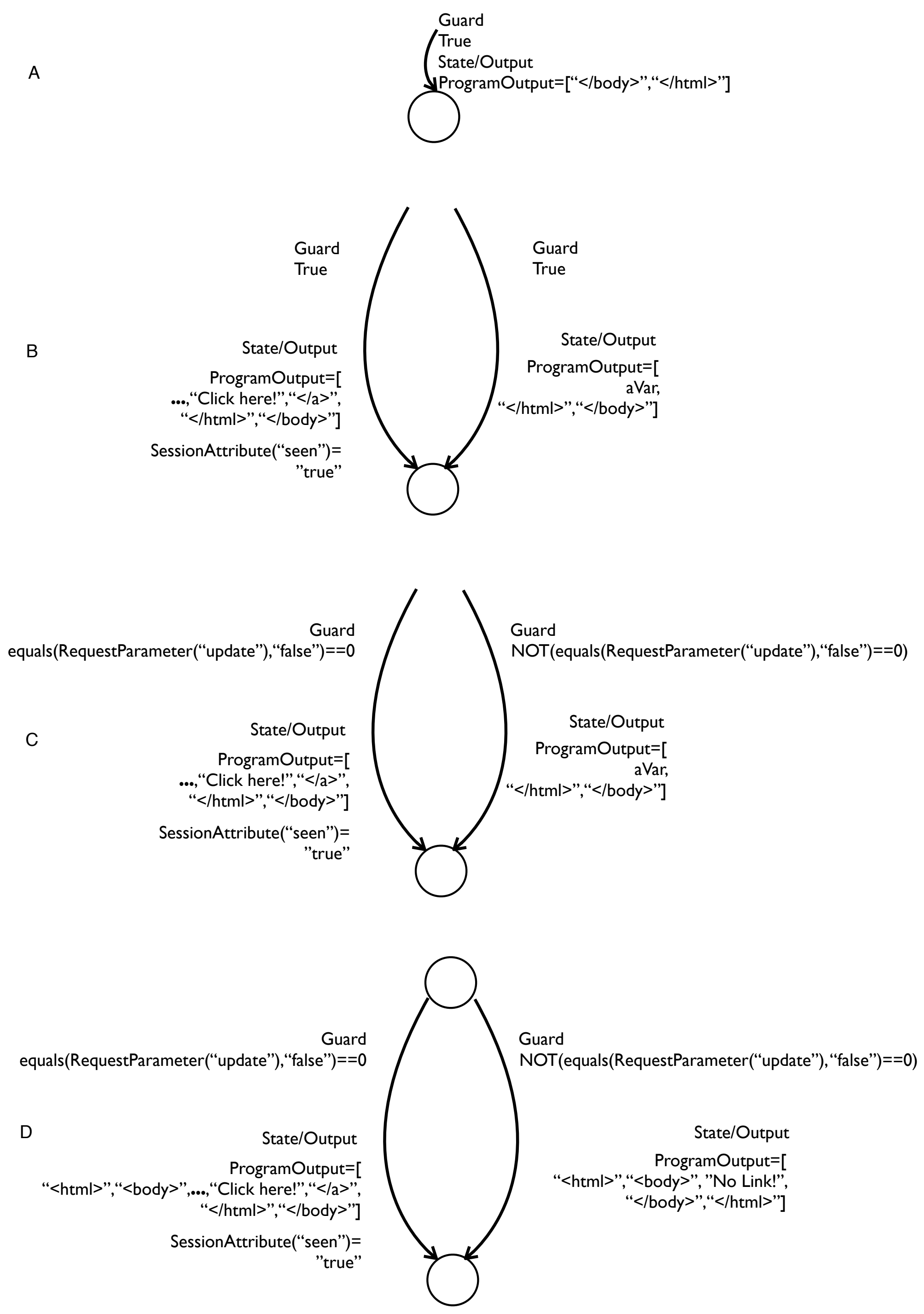
A
Guard
True
State/Output
ProgramOutput=["</body>","</html>"]
B
Guard
True
State/Output
ProgramOutput=[
...,"Click here!","</a>",
"</html>","</body>"]
SessionAttribute("seen")=
"true"
Guard
True
State/Output
ProgramOutput=[
aVar,
"</html>","</body>"]
C
Guard
equals(RequestParameter("update"),"false")==0
State/Output
ProgramOutput=[
...,"Click here!","</a>",
"</html>","</body>"]
SessionAttribute("seen")=
"true"
Guard
NOT(equals(RequestParameter("update"),"false")==0)
State/Output
ProgramOutput=[
aVar,
"</html>","</body>"]
D
Guard
equals(RequestParameter("update"),"false")==0
State/Output
ProgramOutput=[
"<html>","<body>",...,"Click here!","</a>",
"</html>","</body>"]
SessionAttribute("seen")=
"true"
Guard
NOT(equals(RequestParameter("update"),"false")==0)
State/Output
ProgramOutput=[
"<html>","<body>", "No Link!",
"</body>","</html>"]

Figure 2: Exemplifying the Model Building Process

model after both true and false branches are traversed. On the true branch, the value of variable *aVar* is sent to the user. On the false branch, some other string values are sent and there is an update to another relevant state variable for a JSP/Servlet application: the session attribute named *seen* which takes the string value “true” (line 11) on this execution path/transition.

When encountering an *if* decision (e.g., line 8), iSUMMARIZE checks to see if it is relevant i.e., the system state/response depends on the branch taken. If not, the instruction is ignored and the transitions for the true and false branch are jointly kept for further backward propagation. When the *if* is relevant, the constraints/guards for the transitions from that execution point are changed by performing a conjunction with the condition of the *if* statement (or its negation, respectively)[2]. Furthermore, all transitions (from both true and false branches) are propagated backward to the beginning of the method. As a result, the guards of the exemplified iSUMMARIZE model are updated as in Figure 2C (the equals operator tests the equality of two strings, and returns 1 when the strings are equal and 0 otherwise).

For a JSP/Servlet application, another relevant aspect are the request parameters, representing inputs sent (typically) by the user to the application. Since these inputs affect the application behavior, iSUMMARIZE captures the usage of such values using the *RequestParameter* construct.

The algorithm continues until the depth-first order traversal of the method control-flow graph is finished. Figure 2D shows the final form of the extracted iSummarize model. On each possible transition the output of the application will contain the strings “<html>” and “<body>” due to lines 6 and 7. The beginning of the method corresponds to the initial state of the EFSM.

When building the EFSM, every assignment operation triggers a substitution in iSUMMARIZE: every occurrence of the assigned variable in the current EFSM is substituted with the expression on the right side of the assignment. For instance, the assignment in line 5 triggers the substitution of variable *aVar* with the string “No Link!”. The variable appears in the automaton of Figure 2C on the transition corresponding to the true branch of the decision on line 8. As a result of the substitution, Figure 2 shows that when taking the mentioned transition the string “No Link!” is sent to the user.

Handling loops. The previously described algorithm captures as EFSM transitions all relevant paths of a loop-free method. However, when a loop is encountered, we cannot enumerate all paths. To handle loops, a state (control point) for the loop header is inserted in the automaton under construction.

[2]The true/false branches might be interchanged by WALA, for instance, by using an inequality operator instead of equality.

Moreover, the computation of the guards and updates associated with the transitions leaving the header is repeated until a fixpoint is reached (no further change appears). The reason is that inside the loop, we need to capture all relevant variables and their updates that directly or indirectly affect any relevant state variable and/or any variable controlling the loop. If on a loop path, we find that a variable is used to update a relevant state variable, the analysis has to be re-executed by considering the influencing variable as a new relevant state variable.

Managing infeasible paths. The approach used to build the iSummarize model for a method captures every possible loop-free path through that method. However, some of these paths might not be feasible. For instance, when we have two consecutive non-nested *if* statements, four distinct execution paths are possible. But, when the *if* conditions cannot be simultaneously true, the path passing through the true branches of both statements will never be executed (i.e., it is unfeasible). To detect and eliminate such paths, iSummarize tries to directly identify simple contradictions in the path guards (e.g., $1 > 4$ is always false, etc.). Additionally, our tool can use the external Z3[3] theorem prover in order to decide on the satisfiability of a logical condition guarding the execution of a path.

From individual methods to the entire system. The model extraction algorithm described above is applied to each method from the target system, producing a set of EFSMs, one for each method. To then build a model for the entire application, iSummarize performs the following steps:

1. The JSP/Servlets entry points (the *_jspService* methods) are detected and a context-insensitive call-graph is built starting from these entry points, also using the WALA analysis infrastructure.

2. The graph is then traversed in a depth-first order, applying for each reached method the EFSM extraction algorithm after every function invoked by the method has been processed. When a method invocation is encountered during the EFSM construction, the behavioral automaton of the invoked method is integrated/in-lined in the automaton of the invoking method. This action includes the substitution of formal parameters from the invoked automaton with the actual parameter values, the substitution of the caller's variable capturing the return value

[3] http://z3.codeplex.com/

with the returned value of the called method, etc. For this in-line expansion, the call-graph must be traversed in depth-first order. As a consequence of inlining, the tool does not handle recursion.

Following these analysis steps, iSummarize builds an inter-procedural EFSM for each entry point of the investigated application. These automata can be used for further analysis goals, e.g., modeling the application in ASLan++ (the next section describes how they are combined in this case).

iSummarize **Meta-Model.** Figure 3 depicts the entities used to represent an iSummarize model and their relations.

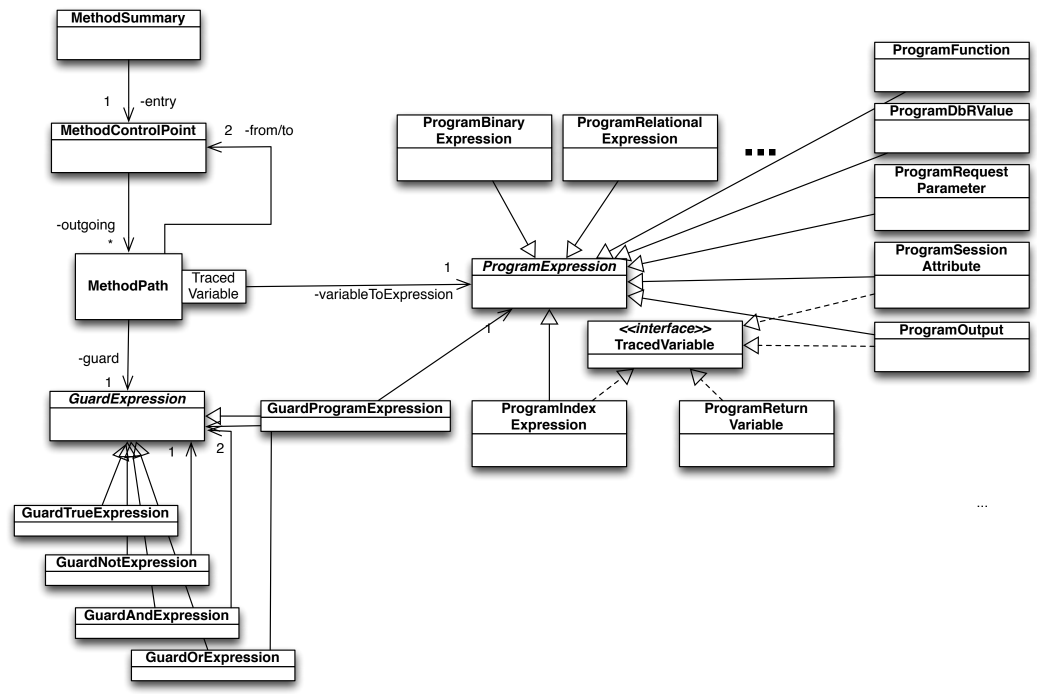

Figure 3: iSummarize Meta-Model

iSummarize represents as first-class entities the behavioral automata (i.e., *MethodSummary*) and their main components: states (i.e., *MethodControlPoint*) and transitions (i.e., *MethodPath*). A *MethodSummary* has one entry control point/state, a *MethodControlPoint* can have any number of outgoing transitions, while a *MethodPath* connects two states.

Each *MethodPath* has a guard condition (i.e., *GuardExpression*) representing the logical condition that enables the execution of that path. The nodes *GuardNotExpression*, *GuardAndExpression*, *GuardOrExpression* and

GuardTrueExpression represent the usual logical connectors and the value true, respectively. A special logical expression that can appear in a guard is the *GuardProgramExpression* which represents a predicate extracted from the program (e.g., *ProgramRelationalExpression* objects correspond to comparisons between numerical values in the program).

A *TracedVariable* corresponds to a relevant variable whose updates need to be captured. We capture updates to fields, relevant local variables (i.e., *ProgramIndexExpression* objects corresponding to local variables) and return variables (i.e., *ProgramReturnVariable*). In particular, for JSP/Servlet applications, we capture modifications to session attributes (i.e., *ProgramSessionAttribute*) and to the output of the program (i.e., *ProgramOutput*).

For every *MethodPath*, the tool associates every *TracedVariable* that is updated on that execution path with the actual *ProgramExpression* denoting the assigned value. Expressions of this kind can be numerical operations (i.e., *ProgramBinaryExpression*), relational expressions (i.e., *ProgramRelationalExpression*), constants or local variables (i.e., *ProgramIndexExpressions*), etc. In particular, for a JSP/Servlet application, request parameter inputs are represented as *ProgramRequestParameter* objects.

Values coming from a database via SQL queries are detected and represented by iSummarize in the behavioral model as *ProgramDbRValue*. Values returned by functions/methods defined in external libraries (i.e., outside the analyzed application and thus not analyzed in-depth) are recorded as *ProgramFunction* objects. This type of object memorizes the invoked function together with the invocation arguments.

Other features and current limitations. The user of the tool can decide to ignore during the analysis a given method from the system under investigation, e.g., replace the method with the identity transform. Such an abstraction must be chosen carefully to preserve soundness, without omitting aspects that are important for the goal of the analysis. Nevertheless, such a feature is important for reducing the size of the extracted model.

At present, iSummarize does not consider paths corresponding to exception propagation, polymorphic invocations or non-static field accesses. These could conceptually be treated in the same way. For instance, to add exception paths the analysis needs to follow and compose exception edges (provided by WALA, but currently ignored by iSummarize) in the method control flow graph. Guard conditions need to be added to execution paths capturing the implicit generation of an exception (e.g., $X == null$ for a *NullPointerException* exception when invoking a method on a reference X); in contrast, $X != null$ must be added to a normal execution path.

2.3 Generating the ASLan++ model

We next describe the ASLan++ Converter component of jModex, which takes a set of EFSMs built by the iSummarize component and transforms them into an ASLan++ model of the investigated application.

Integrating several EFSMs. Given a JSP/Servlet application, jModex builds a behavioral automaton for each of its distinct components (i.e., JSP / Servlet) starting from its entry point. All these distinct automata are translated into a single ASLan++ entity corresponding to the entire application. We describe how these different sub-models are integrated into a single one.

In a JSP/Servlet application, the components can be invoked by a user/intruder in any possible order, without enforcing a particular invocation protocol (e.g., following the request links displayed by the presentation layer). Therefore, to obtain the overall model, all sub-models are jointly enclosed in a *while(true)* loop in which one sub-model is selected for execution based on the user request. A special sub-model is inserted before this *while* loop, corresponding to the initialization phase of the application.

Next, we describe how a single iSummarize automaton / a single component sub-model is translated into ASLan++.

Building the Control Tree. To translate a given automaton, the converter first has to decide on the control structure of the generated ASLan++ model. In other words, the converter must decide which control statements (e.g., *select*, *if*, etc.) will be used and how they will be composed to represent the EFSM semantics and to minimize duplication in the ASLan++ code. For this purpose, the converter performs a structural analysis [3] reducing the EFSM graph using a set of basic structures (regions) associated with specific ASLan++ control statements. As a result, a control tree is obtained, capturing the required control statements and their chaining and/or nesting. Figure 4 exemplifies this process for the automaton from Figure 2D.

Converting Expressions. Once the control structure of an ASLan++ model has been established, the remaining task is to translate the guards and the state updates from the control tree into ASLan++ statements. The guards result in branching conditions for the control statements while the updates are translated into assignments in the bodies of the control statements. In the following, we will show how guards/update expressions are represented in ASLan++ by jModex, i.e., how various entities from the meta-model (Figure 3) are mapped into ASLan++ constructs. Since some of these entities represent the state of the application and thus, we also explain how they are expressed as ASLan++ state symbols.

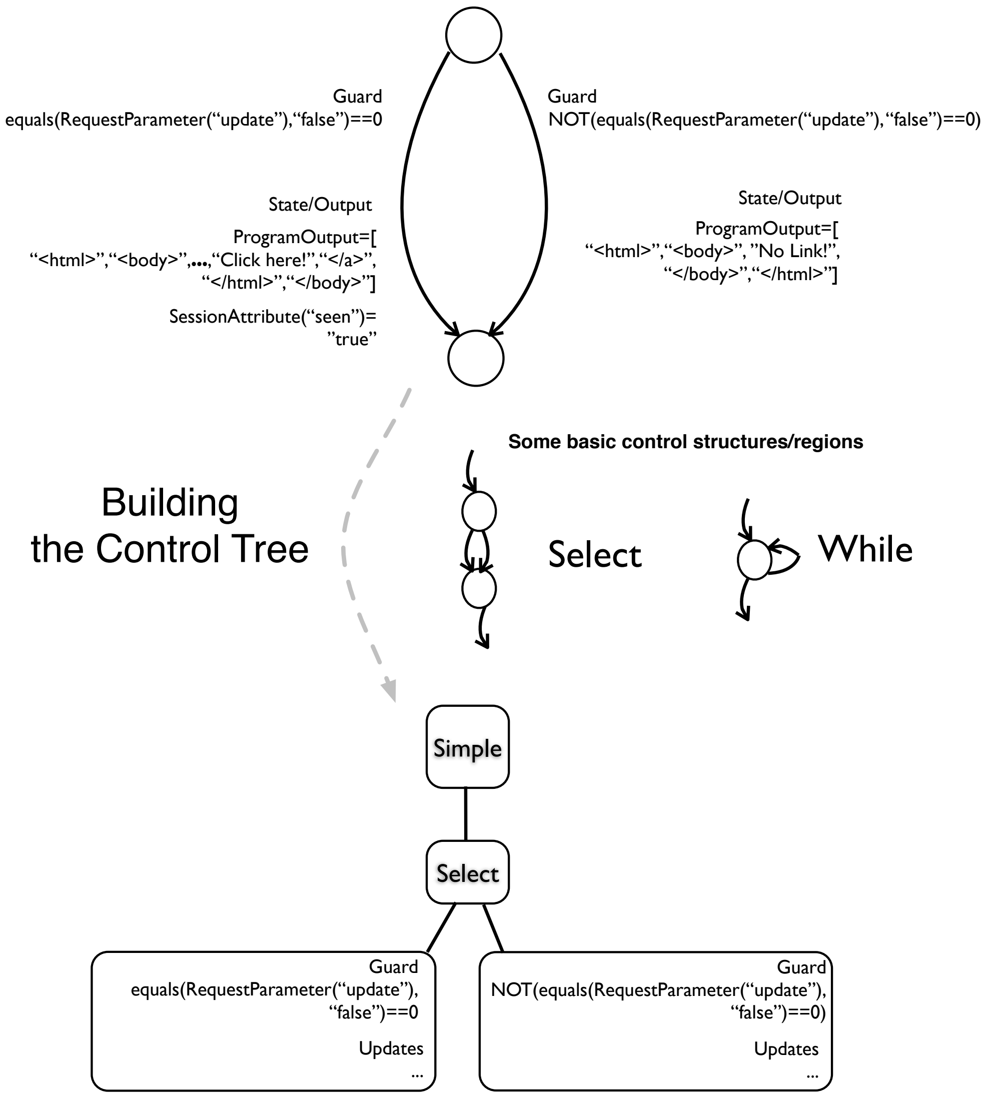
Guard
equals(RequestParameter("update"),"false")==0
Guard
NOT(equals(RequestParameter("update"),"false")==0)
State/Output
ProgramOutput=["<html>","<body>",...,"Click here!","</a>", "</html>","</body>"]
SessionAttribute("seen")= "true"
State/Output
ProgramOutput=["<html>","<body>", "No Link!", "</body>","</html>"]
Building the Control Tree
Some basic control structures/regions
Select
While
Simple
Select
Guard
equals(RequestParameter("update"), "false")==0
Updates
...
Guard
NOT(equals(RequestParameter("update"), "false")==0)
Updates
...

Figure 4: Control Tree Example

Values Correspondence and Operations

- Numeric values: ASLan++ only supports natural numbers. As a result, we have to abstract operations with real/integer values. One option is to create a particular ASLan++ type for numbers (e.g., numeric) and to treat the required numerical operations (e.g., *ProgramBinaryExpression*) as uninterpreted functions over this type. A refinement would be to declare a distinct ASLan++ type for each numerical Java type (e.g., int or double) in order to specialize how each class of values is managed during model checking.

- String values are treated as ASLan++ messages. String concatenation is converted into message concatenation while the equality test is translated as message equality. In some cases, we may define special functions or predicates representing other string-related operations (e.g., equals ignoring case).

Translating Non-State Related Entities

- The relevant constants identified by jModex in a program (i.e., objects of *ProgramIndexExpression* type in Figure 3 associated with constants) are expressed in ASLan++ as constants of corresponding types.
- Request parameter: in a JSP/Servlet application, the objects of type *ProgramRequestParameter* are simple named inputs of type string that the client provides in its request. In ASLan++ we represent them as a set of message pairs (i.e., Params : message.message set). The application entity receives the actual values for these parameters (i.e., a concrete set of message pairs) as a message sent by the client entity.
- Program function: Some functions called in the application code are not analyzed by jModex (e.g., library methods). In these cases, the function return value is represented by a *ProgramFunction* expression (in Figure 3). These expressions are translated to ASLan++ as applications of uninterpreted functions.
- Database values: as suggested by the name, these expressions represent data that come from a database (*ProgramDbRValue* in Figure 3). For such a value, we capture the expressions corresponding to (i) the SQL query producing the value and (ii) to the position in the resulting table. This information can be used in ASLan++ to express database values as results obtained by applying uninterpreted functions.

Translating State-Related Constructions.

- Temporary Variables (i.e., *ProgramIndexExpression* instances in Figure 3 corresponding to local variables) are expressed in ASLan++ as variable symbols of corresponding types, local to the application entity.

- Session attribute: they represent a set of named state variables of the application (i.e., *ProgramSessionAttribute* in Figure 3). In ASLan++ they are expressed as a local set of message pairs (i.e., Sess : message.message set). Accesses/assignments to the state attributes are made using the *contains* function, while attribute deletion can be implemented using the fact retraction construct.

Current Limitations. Currently, the strongest limitation of jModex is that it does not properly capture updates to the database; only read accesses are expressed using uninterpreted functions. Since for some analysis purposes, updates are also relevant, they must also be handled. We plan to eliminate this problem by modeling relevant parts of the database. Moreover, we plan to evaluate the possibility of employing coarser abstractions (e.g., representing and translating the entire output of the application is not relevant in many cases) and to optimize the translation of the expressions identified as relevant.

Example. Listing 2 shows the ASLan++ model produced by jModex for the code from Listing 1. The model is generated abstracting away the HTML output but shows the decision structure based on request parameters.

The *Application* entity (line 15) represents the modeled web application and the *while(true)* loop (line 23) corresponds to the server loop (i.e., it waits for requests from clients). The first *select* is used to model the invocation of a particular JSP/Servlet as discussed in Section 2.3 (paragraph "Integrating several EFSMs"). Since in our example we have just a single JSP/Servlet, we have just a single *on* block (line 25) associated to this single component of the application. The *entry* message represents the invocation of the exemplified service, and the received content provides the values of the request parameters sent by the user (i.e., Params at line 25).

The second *select* statement (line 26) distinguishes between the various execution paths that can be followed in the exemplified program/automaton. When the value of the parameter named "update" (update in lines 27 and 28) is not equal with the string "false" (sfalse constant in line 28) the session attributes of the system (i.e., *Sess*) are updated (line 33) as in the case of the original program from Listing 1. The *if* statements ensure that i) the updated attribute (i.e., seen constant in lines 30 and 33) will not have more than one value in the session set and ii) the updated attribute will not have the null value (i.e., it must exist in the session)[4]

[4]In this case, the second *if* condition is always true. With a simple satisfiability analysis of guards (here, comparison of different constants) the model can be further simplified.

The second path in Listing 1 (i.e., when the “update” parameter is “false”) does not appear in the ASLan++ model because it does not affect the state of the system (we do not capture the output changes in this example) and thus, the path can be eliminated from the model (i.e., we would have an alternative *on* with an empty body which is irrelevant in this case).

Listing 2: The Generated ASLan++ Produced by jModex

```
specification AnExample
channel_model CCM
entity Environment {

 symbols
 entry(message.message set):message;
 seen:message;
 oNull:message;
 strue:message;
 update:message;
 sfalse:message;

 entity Session(S:agent) {

   entity Application(Actor:agent,U:agent,Sess:message.message set) {

     symbols
     Params:message.message set;
     Seen:message;
     Update:message;

     body {
          while(true) {
              select {
                     on (?U*->*Actor:entry(?Params)): {
                        select {
                               on(Params->contains((update,?Update))
                                       & !(?Update = sfalse)):{
                                    if(Sess->contains((seen,?Seen))){
                                       Sess->remove((seen,Seen));
                                    }
                                    if(strue!=oNull) {
                                       Sess->contains((seen,strue));
                                    }
                               }
                        }
                     }
              }
          }
     }
   }

   body {
     new Application(S,i,{});
   }
 }

 body {
   any A. Session(A) where A!=i;
 }
}
```

3 Combining Black-Box and White-Box Inference

3.1 Overview and benefits

Extracting models statically, from the source code of a system, generally produces models that over-approximate the real of the system behavior. Complementarily, when a model is learned at runtime, by repeatedly querying the system, the resulting model may not contain all feasible system behaviors. However, the search space of learning a model can be efficiently limited when source code is available for the system under learning. This can be done by using relevant information from the statically extracted model.

In the following, we assume that the models inferred are in both cases Mealy machines with parameterized guard conditions, and that the learned system behaves deterministically, i.e., for each sequence of input events it produces exactly one sequence of output events.

3.2 Providing abstractions to black-box inference

The model extracted by means of white-box inference can be used to provide several relevant abstractions to the black-box inference process, abstractions that are specific to the domain of white-box analysis. In a black-box only inference approach, such abstractions as transition guards can only be obtained approximatively, in an empirical manner, at the end of the inference process, while by employing a white-box analysis of the application they can be made available before the black-box model learning starts.

Therefore, an important contribution the white-box inference process can bring is to provide both an abstract and a concrete alphabet of input and output events for black-box model learning. Extracting such an alphabet from the source code is significantly different from alphabet extraction via crawling, as the source code provides more information on an application's input/output events.

3.2.1 Extracting the Alphabet of Input Events

As a result of the white-box model inference, we have access to a superset of all feasible paths that link an input to an output event in the application under learning. These paths can be statically explored. When exploring the paths between input and output in the source code, we can gather path conditions and use them to extract relevant constraints on valid input parameters. This leads us to an abstract set of inputs, where each input in the alphabet has its parameters characterized by several logical constraints. We can see such an abstract input in Listing 3, where the domain of each input parameter is restricted by a set of constraints, obtained from the path conditions specifically referring to that parameter.

Listing 3: Abstract Input Example

```
<input address="http://localhost:8080/org/apache/jsp/mock/Login.jsp"
 method="POST" type="FORM">

  <parameters>

    <parameter name="FormName">NOT(RequestParameter("FormName") eq null)
     AND RequestParameter("FormName").equals("") eq 0</parameter>

        <parameter name="ret_page">NOT(RequestParameter("ret_page").equals("")
        eq 0) AND NOT(RequestParameter("ret_page") eq null)</parameter>

        <parameter name="querystring">RequestParameter("querystring").equals("")
        eq 0 AND NOT(RequestParameter("querystring") eq null)</parameter>

        <parameter name="Login">RequestParameter("Login") eq null</parameter>

        <parameter name="FormAction">RequestParameter("FormAction") eq null</parameter>

 </parameters>

</input>
```

Listing 4: Concrete Input Example

```
<input address="http://localhost:8080/org/apache/jsp/mock/Login.jsp"
 method="POST" type="FORM">

  <parameters>

    <parameter name="FormName">0</parameter>

        <parameter name="ret_page"></parameter>

        <parameter name="querystring">0</parameter>

        <parameter name="Login">null</parameter>

        <parameter name="FormAction">null</parameter>

 </parameters>

</input>
```

The abstract set of inputs cannot be used as such by the black-box inference process, but, by employing a SMT solver such as HAMPI [2], one can instantiate each abstract input. Thus, whenever the conditions associated to the parameters of an abstract input are satisfiable, we can solve them and create concrete values for each input parameter. In this way, we obtain an alphabet of concrete input events that can be used for black box inference.

Such an example of a concrete input, defined by a set of concrete parameter values, can be seen in Listing 4.

It is worth noting here that, while working with the abstract alphabet appears at first to complicate the alphabet extraction process, it actually brings significant value when compared to crawling. As the black-box model learning process aims to obtain an EFSM that would describe the behavior of the application under inference, it needs to determine relevant guard conditions for the model's transitions. The crawler only extracts, in a black-box manner, a concrete alphabet of the system under inference, where each parameter is characterized by a set of possible values. This set is not exhaustive and it does not give out any information on whether the system may react in a way on some of these values and differently on others. The actual guard conditions are extracted only during the black-box inference process and, as they are obtained empirically, using only runtime observations, they are imprecise. By contrast, when the concrete input is an instance of a known abstract input, the guard conditions can be derived from the conditions on the abstract input parameters, which are precise. So, the extraction of the abstract alphabet is not only a preliminary step in providing the concrete alphabet, but also a significant source of insight for the black-box inference process, which can use it to obtain precise guard conditions.

3.2.2 Extracting the Alphabet of Output Events

Obtaining a white-box model of the system also has advantages for determining the output alphabet of the system under inference. An output HTML page, generated by the application as a HTTP response, may have several variable fields. Such fields may depend on the actual input parameters provided in the HTTP Request, may originate from a database, etc. When the output alphabet is obtained by crawling, one has to request a page several times in order to be able to approximatively identify these variable fields. But, when white-box analysis is possible, we can determine statically, for each field of a HTTP response whether it is constant or variable, whether it was obtained from an access to a database or from a third party library function (which is left uninterpreted by jModex to avoid excessive complexity). Such fields can then be statically identified as variable fields, and reliably reported as such.

For uniformity in the way outputs are considered by the black-box inference process, we also strip the HTML pages returned as a HTTP response down to their relevant structure, and consider two pages as equivalent if their structure is similar. This is done by removing the actual data associated to the variable output parameters, as using it in the comparison between two

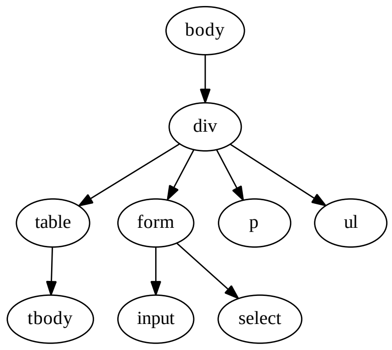

Figure 5: Output Example

HTML pages would result in a significant amount of false negative results. In Figure 5 we can see an example of tree corresponding to an output stripped of all data associated to the variable output parameters, as well as from irrelevant tags.

3.2.3 Alphabet Extraction Method

The steps involved in computing a concrete input and output alphabet that can be provided to the black-box inference tool are the following:

1. Use the iSummarize component of the white-box model inference tool jModex to analyze the source code of the application and extract a detailed model as an EFSM.

2. For each method, start from its initial state, the *MethodControlPoint* entry, and enumerate all possible paths through the method. While going through a path, we gather all *GuardExpression* conditions that involve entities of the type *ProgramRequestParameter*, which represent input parameters. Similarly, all the entities of type *ProgramOutput*, each representing an output fragment, are collected on the explored path, in the order in which they are met.

3. An abstract input (represented by a set of constraints on the values of concrete input parameters, which are, in this case, HTTP request parameters) is obtained whenever the end of a path (its exit point) is reached. An output for the explored path is also obtained at this point, by concatenating the ordered output fragments we have collected. Both the abstract input and the output are associated to the path and are memorized as such.

4. For each obtained path, we try to solve the constraints characterizing the abstract input parameters by using the Hampi [2] SMT string solver. If we succeed, i.e., the path is feasible, we obtain a concrete input, which is added to the concrete alphabet. Also, we add the corresponding abstract input to the abstract alphabet. If the path is found infeasible, the abstract input is dropped.

5. Feasible paths are grouped together based on their outputs. These outputs, originally HTML pages generated by the application, are stripped down of parameter data and common tags. For each group of equivalent paths, only one abstract and one concrete input event are kept – this allows us to reduce the size of the input alphabet. If an exhaustive alphabet is desired, the path grouping feature can be disabled.

6. As each group of equivalent feasible paths are characterized by one common output and one common, chosen representative input event (be it abstract or concrete), we pair together the input and output events belonging to the same path group and provide these pairs in a separate file. These input/output pairs are important, as they can be used to label the transitions on the Mealy machine learned by black-box inference. We can see an example of such an input/output pair in Listing 5 (where an abstract input is considered and the output is presented in its original, unstripped form).

Listing 5: Input/Output Pair Example

```
<IOPair>

<input address="http://localhost:8080/org/apache/jsp/Login.jsp"
method="POST" type="FORM">
  <parameters>
  <parameter name="FormName">NOT(RequestParameter("FormName") eq null) AND
        NOT(RequestParameter("FormName").equals("") eq 0)</parameter>
  <parameter name="ret_page">RequestParameter("ret_page").equals("") eq 0 AND
        NOT(RequestParameter("ret_page") eq null)</parameter>
  <parameter name="FormAction">NOT(RequestParameter("FormAction").equals("") eq 0)
        AND NOT(RequestParameter("FormAction") eq null)</parameter>
  <parameter name="querystring">RequestParameter("querystring").equals("") eq 0
        AND NOT(RequestParameter("querystring") eq null)</parameter>
  <parameter name="Login">RequestParameter("Login") eq null</parameter>
  </parameters>
</input>

<output>
  <html>
  <head>
  <title>Book Store</title>
  <meta name="GENERATOR" content=
      "YesSoftware CodeCharge v.1.2.0 / JSP.ccp build 05/21/2001"/>
  <meta http-equiv="pragma" content="no-cache"/>
  <meta http-equiv="expires" content="0"/>
  <meta http-equiv="cache-control" content="no-cache"/>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  </head>

  <body style="background-color:#FFFFFF; color:#000000; font-family:
Arial,Tahoma,Verdana,Helvetica background-color:#FFFFFF; color:#000000;
font-family:Arial,Tahoma,Verdana,Helvetica">
  <center>
  <table>
  <tr>
   <td valign="top">
   <table style="" border=1>
  <tr>
   <td style="background-color:#336699;text-align:Center;border-style:outset;
border-width:1" colspan="2">
   <font style="font-size:12pt; color:#FFFFFF; font-weight:bold">
   Enter login and password</font></td>
  </tr>
  <tr>
   <td colspan="2" style="background-color:#FFFFFF; border-width:1">
   <font style="font-size:10pt; color:#000000"></font></td>
  </tr>
  <form action="Login.jsp" method="POST">
  <input type="hidden" name="FormName" value="Login">
  <tr>
  <td style="background-color:#FFEAC5; border-style:inset; border-width:0">
  <font style="font-size:10pt; color:#000000">Login</font></td><td style=
  "background-color:#FFFFFF; border-width:1">
  <input type="text" name="Login" maxlength="50" value=""></td>
     </tr>
  <tr>
  <td style="background-color:#FFEAC5; border-style:inset; border-width:0">
  <font style="font-size:10pt; color:#000000">Password</font></td>
  <td style="background-color:#FFFFFF; border-width:1">
  <input type="password" name="Password" maxlength="50"></td>
  </tr>
  <tr>
  <td colspan="2"><input type="hidden" name="FormAction" value="login">
  <input type="submit" value="Login">
  <input type="hidden" name="ret_page" value="RequestParameter(ret_page)">
  <input type="hidden" name="querystring" value="RequestParameter(querystring)">
  </td>
  </form>
  </tr>
 </table>
    guest/guest<br>
        admin/admin
  </td>
  </tr>
 </table>
 <center><font face="Arial"><small>This dynamic site was generated with
 <a href="http://www.codecharge.com">CodeCharge</a></small></font></center>
 </body>
</html>
</output>

</IOPair>
```

3.2.4 Advantages and Limitations

Extracting the input and output alphabet from the white-box model, when source code is available, has the following benefits when compared to extracting the alphabet by crawling:

- an easier, straightforward selection of relevant input parameters
- obtaining a set of precise constraints on the domain of abstract input parameters, from which transition guards can be further derived for the EFSM learned by black box model inference
- automated generation of relevant concrete values for selected input parameters, based on the constraints obtained for the abstract parameters
- a static identification of variable elements in HTTP Responses
- the possibility of inferring relations between input and output events

As a limitation of the approach, one must remember that white-box inferred models are relatively low-level, and the control flow between an input and an output event can have many decision points, cycles, etc. This may lead, in some cases, to a large number of possible paths between an input and an output event, even though not all of them are always feasible. Enumerating all these paths is computationally expensive, even if some of them can be dropped as infeasible early on in the exploration process.

Further on, a large number of paths can translate in a large input and possibly also a large output alphabet. This last limitation was partially addressed by grouping together paths based on output equivalence (where a structure-based abstraction of actual outputs was considered), and by only considering one input and one output event for each group of equivalent paths. For the considered example of *Login.jsp*, this approach has led from 604 to only 27 input/output pairs, thus narrowing down the initial alphabet of events to a small relevant subset, much more suitable for use in the black-box model inference process.

3.3 Using the white-box model as a learning oracle

Let us assume that the black-box inference technique uses the alphabet of the statically extracted model. This model over-approximates the real system behavior as it may contain infeasible transitions and, also, by grouping together different states of the system. This leads to nondeterminism in the white-box model, therefore one input sequence can result in more than one output sequence, while the real system normally produces exactly one output sequence for each input sequence (abstracting away from on-demand ads, randomly selected database records, etc.).

However, there can be input sequences for which the white-box model is precise and answers with an unique output sequence. In such cases, the white box model can be used to answer membership queries for those input sequences, thus acting as a proxy for the real system. Depending on the degree of nondeterminism in the white-box model, using it as a proxy oracle can reduce the number of membership queries the oracle is actually asked (i.e., the number of tests run on the real system). As the number of runtime queries is the actual bottleneck for black box model inference, this enhancement can significantly reduce the cost of black-box model learning.[5]

Also, the white-box model (used as a proxy) can be refined in parallel with the black-box inference process: whenever the white-box model does not have a unique answer to an input sequence and the actual system is tested instead, the output sequence thus obtained can be used to prune the other, unfeasible paths from the over-approximated model. This can reflect on the proxy giving more precise answer to further queries.

[5]Another possibility is that, if the real system becomes unavailable at some point, the learning algorithm will drop the assumption of determinism for the learned system and continue the inference process by only querying the white-box proxy.

4 Model-driven detection of unintended executions

In addition to interfacing white-box and black-box inference, we are developing an approach that detects potentially unintended execution flows in a web application by comparing a model obtained by deep web crawling with the white-box model extracted from the server-side application code. The comparison aims to identify executions in the white-box model that can be confirmed by tests but cannot be triggered by following the presentation layer (i.e., following links and submitting forms in the application's web pages). These could be action sequences unintended by the application designer, in particular, instances of unauthorized access (e.g., to functionality that should be available only after authentication in a login page).

4.1 Page graph extracted by crawling

As a first step, we have developed a deep web crawler that builds a page graph of the application by following links on the application's web pages and submitting forms.

Abstracting HTML The crawler works with abstract HTML pages which contain the possible actions that a user can make on the given page. In order to create an abstract page, the crawler uses HtmlUnit, a headless browser, to send a HTTP request and to parse the response for links and forms. After getting the list of anchors and forms, the crawler generates a set of requests, which represent the actions of a user, associated with the abstract page. To generate this set, the crawler applies a request generation strategy for links and another one for forms. The requests generated by forms are the following:

1. a request with the form parameters having empty values
2. a request with parameters having their default values
3. a request with free-form parameters having random values, other parameters (like checkboxes, radiobuttons) having a randomly chosen value from their possible values
4. a user-specified request (for example login credentials)

Links are grouped based on their target and their parameters (ignoring their values); two pages with identical groups of links are deemed *similar*.

From each group the first N links are chosen and added to the request set. For example, if we have the following links:

```
<a href="Books.jsp?category_id=1&"></a>
<a href="Books.jsp?category_id=2&"></a>
<a href="Books.jsp?category_id=3&"></a>
<a href="BookDetail.jsp?item_id=1&"></a>
<a href="BookDetail.jsp?item_id=2&"></a>
<a href="BookDetail.jsp?item_id=3&"></a>
```

and we configure N=2, the following requests will be generated:

```
Request1[ target=Books.jsp method=GET Parameters={ category_id="1" } ]
Request2[ target=Books.jsp method=GET Parameters={ category_id="2" } ]
Request3[ target=BookDetail.jsp method=GET Parameters={ item_id="1" } ]
Request4[ target=BookDetail.jsp method=GET Parameters={ item_id="2" } ]
```

As another example, for the following form:

```
<form action="Login.jsp" method="POST">
 <input type="hidden" name="FormName" value="Login">
 <input type="text" name="Login" maxlength="50" value="">
 <input type="password" name="Password" maxlength="50">
 <input type="hidden" name="FormAction" value="login">
 <input type="submit" value="Login">
</form>
```

if the user specifies that the guest:guest values for the parameters should be tried too, then the generated requests will be the following:

```
Request1[ target=Login.jsp method=POST Parameters={ FormName="Login" , Login="" ,
     Password="" , FormAction="login" } ]
Request2[ target=Login.jsp method=POST Parameters={ FormName="Login" , Login="
    guest" , Password="guest" , FormAction="login" } ]
Request3[ target=Login.jsp method=POST Parameters={ FormName="Login" , Login="
    random123" , Password="random321" , FormAction="login" } ]
```

Constructing the page graph The graph's nodes are *states*, its edges are *requests*. A state is defined by the path (a sequence of requests) from the root node to it and by its associated abstract page. The crawler starts from the root node, which has a configured list of requests, the public pages of the application, which are known and can be directly accessed. The construction of the page graph is realized by expansion and reduction rounds.

In the expansion round, the crawler executes all requests of the states found in the previous round. With each expansion round, the current exploration depth (CD) is increased by one. At first only expansion rounds are made, until CD becomes greater than the comparison depth (K). When this happens then each expansion round is followed by a reduction round.

The reduction round compares the states at depth D = CD − K to all states at depth less than D. To determine if a state is new we check if it can be considered equal to a previously explored state. We consider two states equal if their abstract pages are similar and their child state's pages are similar too. The verification of the child states is done up to depth K. If they are found to be equal, the new state is removed from the page graph, along with its child states, reducing the number of future states to be explored.

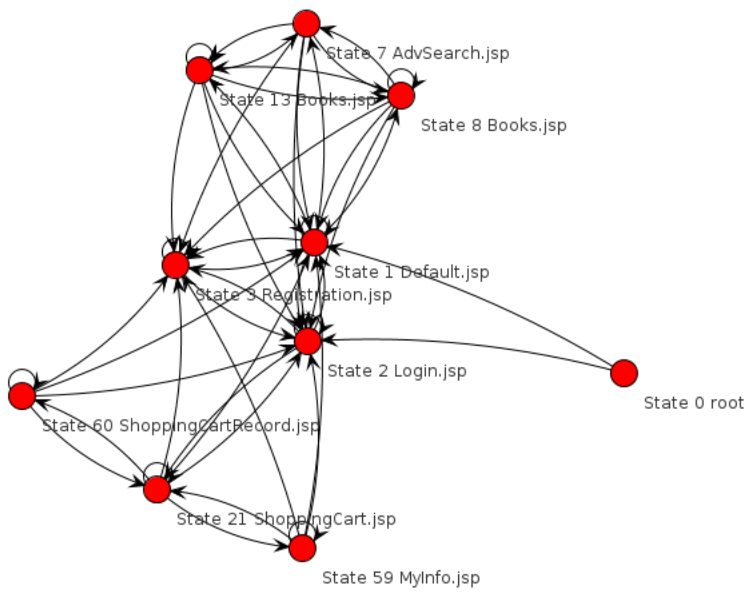

Figure 6: Crawled Bookstore

Figure 6 presents the page graph of the Bookstore application, with K=0, meaning two states are considered equal if their pages are similar. The login credentials for the guest user are specified as input to the crawler, as well as the two starting pages, Login.jsp and Default.jsp. A directed edge between node A and B means that there is at least one request made from state A that resulted in a transition to state B.

4.2 Correlating crawled and source-extracted models

Comparing the model extracted from source code with the one extracted through crawling raises several challenges, e.g., different levels of abstraction and different representations. To enable a smooth comparison between models extracted from source code and by deep crawling we need to abstract away information from the EFSM inferred from source code and create a model that is similar in nature to the one extracted through crawling.

First, we correlate the white-box model with the black-box one based on the output, i.e., the HTML code that the server sends as response. We do this by accumulating the output that each symbolic path in the EFSM would generate. The accumulated output is symbolic, containing abstract values, e.g., data coming from the database. We currently assume that the data coming from the repository is pure data, and not HTML code, links, etc. For the applications we address, this assumption holds most of the time.

In order to be able to create correlations between the black-box model and the white-box one we merge all paths that have equivalent outputs into a multipath. When grouping paths together, the guard for the new multipath becomes the disjunction of all the guards of the paths grouped together. The notion of output equivalence can be considered a parameter of this technique. We currently use the same technique described in Section 4.1 for abstracting the symbolic output. We consider two pages equivalent, and merge the paths, if two abstracted pages have the same possible actions, i.e., forms and links.

Second, by analysing each multipath, we are able to generate concrete requests that would execute that multipath. For multipaths that are control-dependent on request parameters, we generate concrete values for the requests parameters that would execute that multipath. Each multipath has a guard associated with it. We use the Hampi string solver [2] to solve the constraints on request parameters for each path. This gives us the option to validate faults detected through static analysis and also guide the crawler and improve on its incompleteness.

4.3 Finding executions undetected by crawling

In contrast to the crawled model, the model extracted by analyzing the application servlet code centers on the server state. It is an extended Mealy automaton, where the HTTP response generating the page content is viewed as transition output. The server processes any incoming HTTP request, not limited to those that can be produced by following the links and forms provided to the user.

In general, the white-box model will contain a superset of executions compared to those in the black-box page graph, considering that both models are sound. The first use of the white-box model will be to guide the crawler towards additional executions: checking for feasibility of transitions in the white-box model will lead to concrete test inputs that can be applied, leading the crawler towards portions of the state space that random exploration (with no knowledge of constraints to be satisfied) may have missed.

In a second step, relations between nodes in the black-box page graph can be identified (such as pages that can only be reached through a login page). We can identify these pages by looking for dominator nodes in the black-box model. Writing the corresponding specifications, the white-box model can be checked and violations reported and subsequently confirmed by testing.

The white-box model centers on capturing the server internal state. We define a state in the white-box model as a set containing the current values stored by the server, e.g., $Session[username] = Alice$. We express the constraints on a transition between two states in first-order logic without quantifiers. A relation R is a guard over the *Input*, i.e., the values of the request parameters of the current request, and the internal server state, e.g., the values in the current session. A relation $R(S_0, \mathit{Input}, S_1, \mathit{Output})$ expresses that an application currently in state S_0, on input *Input* reaches state S_1 and outputs the HTML in *Output*. The relation R is built from the white-box inferred EFSM and represents the disjunction of all guards over all paths in the white-box EFSM. We consider that cycles execute at most once for the sake of efficiency, but this can be improved.

Thus, we can express a user's initial interaction when logging in by $R(\{\}, \mathit{Login.jsp?username} = \mathit{Alice}\,\&\,\mathit{password} = {*}{*}{*}{*}, \mathit{Session}[\mathit{username}] = \mathit{Alice}, \mathit{Session}[\mathit{userRights}] = \mathit{1}, \langle \mathit{html} \rangle \ldots \langle /\mathit{html} \rangle)$, where $\{\}$ is the initially empty session state. We express chains of interactions as $R(\{\}, \mathit{Req}_1, S_1, \mathit{output}_1) \wedge R(S_1, \mathit{Req}_2, S_2, \mathit{output}_2) \wedge \ldots \wedge R(S_{n-1}, \mathit{Req}_n, S_n, \mathit{output}_n)$. We match these transitions with those in the black-box model based on the generated output. Although the output obtained through source code analysis might be symbolic and therefore a simple string comparison does not work, we use the technique described in Section 4.1 to successfully abstract away from these differences. This way we are able to match transitions in the white-box model with ones in the black-box model.

We use the black-box model to infer a specification for the application. Because the page-graph obtained through crawling represents the way an honest user would interact with the application, following only the displayed user interface, we consider this model as an intended specification. We can then check if the complement of the black-box interactions is SAT in the white-box model; if this is the case, we found an interaction that was not discovered following the user interface. This can be due to a vulnerability, an error, or to the incompleteness of black-box crawling in general. We need to validate the interaction found by executing the application and ensuring the found interaction cannot be executed following the user interface; if this is the case, we found an unintended execution, otherwise we refine the black-box model and restart the process. We can also check more specific patterns, e.g., if we find that one page is a dominator for another one through black-box model inference, we can check that this holds true also in the white-box model. If it does not, we have to validate the found counter-example and ensure that it cannot be executed just by following the user interface. If this is the case, we found a possible vulnerability. We use the Z3 SMT Solver to check these relationships and find unintended executions.

As the comparison of presentation layer and model extracted from source is a new approach in the project, it is currently under active development. Complete results will be provided in Deliverable 5.4 Final Tool Assessment.

5 Conclusions

We have presented jModex, a tool that enables the programmer to automatically extract an extended finite state machine model from a JSP/Servlet application and translate the EFSM to an ASLan++ model. In addition to the benefits of automation, the model produced in this way is more accurate than a handwritten model obtained by code inspection. A preliminary assessment shows that jModex is able to automatically infer complete models on the targeted case studies.

Next, we show two directions to combine the white-box inferred EFSM with the black-box model. We present means to use jModex in conjunction with black-box model inference. We show a method to guide the black-box model inference towards better exploration by using information obtained from source code by automatically extracting an alphabet for the application. We then show how to use the white-box model as an oracle for black-box exploration.

As a supplementary approach, we present a method to detect unintended executions by using the black-box model obtained through crawling as a specification for the white-box extracted model.

References

[1] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 25–32. IEEE, 2009.

[2] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In G. Rothermel and L. K. Dillon, editors, *18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116. ACM, 2009.

[3] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[4] SPaCIoS. Deliverable 5.2: Proof of Concept and Tool Assessment v.2, 2012.

[5] SPaCIoS. Deliverable 2.5.1: Framework for Concretisation of Abstract Tests, 2013.