EXTRACTING BEHAVIORAL MODELS FROM SERVICE IMPLEMENTATIONS

Ioana Şora[1,2] and Doru-Thom Popovici[1,2]
[1]*Department of Computer and Software Engineering, Politehnica University of Timisoara, Romania*
[2]*Institute e-Austria (IeAT) Timisoara, Romania*
ioana.sora@cs.upt.ro

Keywords: Reverse Engineering; Behavioral Model; EFSM; Distributed Computing; Service Computing

Abstract: Formal behavioral models of software services are used as input by analysis tools which check their properties on hand of the given models. However, there is a gap between the real systems which have to be validated and their abstract models. This work proposes to bridge this gap by tools which extract behavioral models from software services implementations. The method proposed here aims at ensuring a general solution, applicable to several service technologies. The core of this solution consists of transforming the control flow graph of a communicating system into its corresponding behavioral model represented as an EFSM (Extended Finite State Machine). The extracted EFSM model can be automatically translated into an entity description in a formal security specification language for distributed systems. This will enable the use of formal analysis tools for real service implementations.

1 INTRODUCTION

Important research efforts aim at improving security in the Internet of Services by developing a new generation of security analyzers for service deployment, provision and consumption (Vigano, 2012). The techniques used for discovering faults and vulnerabilities comprise model checking or model based testing. All these techniques take as input a model of the system under validation and the expected security goals, expressed in a specific description formalism. Usually the models are hand written by the security analyst, based on the service specifications. This approach has been successfully used in the discovery of protocol errors, of logical errors which are present in the known models of systems, or the discovery of errors due to the interaction of known systems.

However, relying on hand-written models is not a suitable approach in following situations:

At service consumption time, services are black-boxes that come without (trusted) models and their code is not available. A model can be inferred from I/O sequences. There is a large field of research of learning behavioral models by combining black-box testing and automata learning (Lorenzoli et al., 2008), and it begins to be used for inferring models of web applications (Bertolino et al., 2009), (Hossen et al., 2011), (Merten et al., 2012).

At service implementation and deployment time, service developers could benefit more from the large variety of tools for security analysis and validation, such as the SPaCIoS tool (Vigano, 2012), if they had model-extractor tools able to extract behavioral models from service implementations. Currently they have to manually write such models using the Aslan++ specification language (Oheimb and Modersheim, 2012). These model extraction tools are different from the ones used in the service consumption use case, since they should take advantage of having full access to the code of the implementation. Our current work addresses this issue of extracting behavioral models from service implementations, by applying specific white box techniques based on the analysis of their control flow graph.

The remainder of this article is organized as follows. Section 2 presents background information about representing behavioral models as extended finite state machines. Since services come in a variety of technologies, their implementation using different frameworks, we first define our method of model extraction in abstract, technology-independent terms in Section 3. Section 4 presents specific issues of particularizing our method for extracting models from services implemented using different technologies .

2 EXTENDED FINITE STATE MACHINES USED FOR BEHAVIORAL MODELING

In this work we use a form of Extended Finite State Machines (EFSM) for representing behavioral models. Our EFSMs are Mealy machine models which are specifically tailored for white-box modeling of I/O based systems.

We consider as I/O the messages exchanged by the system with its environment. Each message is characterized by a message type and a set of message parameters which may have different values. The input alphabet of the EFSM is the set RM of all message types rm, which may be received by the system. The output alphabet of the EFSM is the set SM of all message types sm, which may be sent by the system. For each message type m, $m \in RM$ or $m \in SM$, the set of parameter types $P(m)$ is known.

An EFSM model consists of S, the set of all states s, with only one state being the initial state s_0, a set T of all transitions t between states, and V the set of all state variables v.

A transition t is defined by six components: its origin state $s_i \in S$, its destination state $s_j \in S$, the received message $rm \in RM$, the guard predicate g, the action list al, the sent message $sm \in SM$.

A transition t between two states s_i and s_j occurs when a message rm is received and a guard condition predicate g is true. In this case, the list of actions associated with the transition al is executed and a message sm is sent. It is possible that some of the following components of a transition are missing: rm, g, al, sm.

State variables and parameters may be scalar variables or sets.

A guard condition predicate g is a boolean expression. The operands of the guard predicate g on a transition fired by a received message rm with a set of parameters $P(rm)$ can be both state variables $v \in V$ and parameters of the received message $p \in P(rm)$. The operators can be boolean operators (and, or, not), relational operators, or set operators (contains).

A list of actions al is a ordered sequence of actions a_i. An action a_i on a transition fired by a received message rm with a set of parameters $P(rm)$, which sends a message sm with a set of parameters $P(sm)$ is an assignment. The left value of the assignment is a state variable $v \in V$ or a parameter of the sent message $p \in P(sm)$. The right value of the assignment is an expression which can have as operands state variables $v \in V$, or parameters of the received message $p \in P(rm)$. Operators are boolean, relational and set operators (add to, remove from).

3 FROM (ABSTRACT) CONTROL FLOW GRAPH TO EXTENDED FINITE STATE MACHINE

3.1 Preliminary assumptions

We present the principles of our model inference algorithm starting from the following assumptions:

- The system is described by a complete, interprocedural Control Flow Graph (CFG).
- There are explicit statements, corresponding to a node in the CFG, for receiving and sending messages of a specified message type and having message parameters.

In our approach, we choose to determine the set of states in the EFSM model corresponding to a set of *essential* program counter values (a set of *essential* nodes in the CFG). A transition between two EFSM states corresponds to a path between CFG nodes which contains at least one *relevant* node. (We will detail the concepts of *relevant* and *essential* CFG nodes in Section 3.2).

This is different from the classical approach of defining the states as corresponding to predicates over the state variables, as done in the related approaches in the context of specification mining by static analysis for classes (Shoham et al., 2008), (Alur et al., 2005). We have chosen this approach because in real applications all the state variables can be complex data structures and it may be a complex task to determine predicate abstractions in this case.

3.2 Building the EFSM

3.2.1 Relevant nodes

An important preliminary step consists in identifying the *relevant* nodes of the CFG.

In principle, an aspect is considered to be relevant for our model if it influences the external observable behavior which consists of the messages received or sent by the system.

A variable is marked as *relevant* if one of the following occurs:

- it is on a downstream dataflow from a parameter of a received message
- it is on an upstream dataflow ending in a parameter of a sent message

A CFG node is marked as *relevant* if one of the following occurs:

- it corresponds to a message receive or message send instruction

- it handles a relevant variable

A CFG path is *relevant* if it contains at least one relevant node. Determining the relevant paths is actually a form of program slicing.

3.2.2 Essential nodes, EFSM states and transitions

It is not necessary that all relevant CFG nodes (which may be far too many) become states in the EFSM model. We call *essential* nodes only the CFG nodes which correspond to nodes of the EFSM.

We propose the following algorithm to identify the essential nodes and the transitions between them:

- The start node is an essential node, and it corresponds to the initial state of the EFSM.
- Any CFG node containing a ReceiveMesage statement is an essential node. It introduces a new EFSM state. The relevant outgoing paths will correspond to outgoing transitions enabled by the received message. Each of these transitions will end in the next state which will be identified as essential on the respective outgoing path. The relevant path conditions are collected as guard predicates for the corresponding transition, while assignments involving relevant variables are collected as list of actions for the corresponding transition.
- A conditional branching node in the CFG is an essential node only if it uses a relevant variable which has been defined in a node preceding it on an incoming path (this includes also the case of loops). It introduces a new EFSM state which has an incoming transition corresponding to the incoming path with the definition node and outgoing transitions corresponding to the outgoing conditional paths.

After determining the essential nodes and identifying the paths between them which correspond to transitions, for each transition we determine its received messages, guard predicates, actions, sent messages. The guard predicate of a transition is composed of all relevant conditions that are on the corresponding path between the two nodes. The action list of a transition contains all assignment or set operations executed on relevant variables on the corresponding path between the two nodes.

An EFSM is deterministic if from any state *s*, when any message *rm* is received, there is at most one transition possible. The EFSM built according to the method presented above is deterministic, since transitions outgoing from a state, in the case that they are labeled with the same received message, they have mutually exclusive guard predicates, since they resulted from different paths of the CFG .

3.3 Example

We consider as example the control flow graph of a typical server. For presentation purpose, we use pseudocode to describe the abstract control flow of a simple online Shop. It manages orders, their payment and their delivery. As we will discuss in Section 4, the actual code implementing this simple online Shop may look very different, depending on the particular technologies or APIs used.

```
1:   orders:={}
2:   payments:={}
3:   while(true)
4:     switch ReceiveMesssage():
5:      case:(orderType, name)
6:            add name to orders
7:      case:(payType, name)
8:            if (name in orders)
9:               add name to payments
10:     case:(deliveryType, name)
11:           if (name in payments)
12                remove name from payments
13:               remove name from orders
14:               SendMessage
                      deliveryResp, goods
15:          else  SendMessage
                       deliveryResp, error
16:  endwhile
```

We determine the nodes (pseudocode statements) 1 and 4 as being the essential nodes, according to the method outlined before. The five possible execution paths below this node correspond to five self-loop transitions.

Figure 1 presents the corresponding EFSM of the simple Shop server. In this figure we shortened for presentation purposes the names: the message types are denoted by `o`, `p`, `d`, and `dR` (for `orderType`, `payType`, `deliveryType`, and `deliveryResp`), the parameter `name` is denoted `n`, the state variables `orders` and `payments` are named `os` and `ps`.

4 MODELING SERVICES OF DIFFERENT TECHNOLOGIES

In section 3.2 we have outlined our model construction algorithm assuming as starting point an abstract CFG (i.e., a complete CFG of the whole system, having special explicit instructions for sending and receiving messages).

The basics of our method are set by building blocks for static code analysis such as call graph

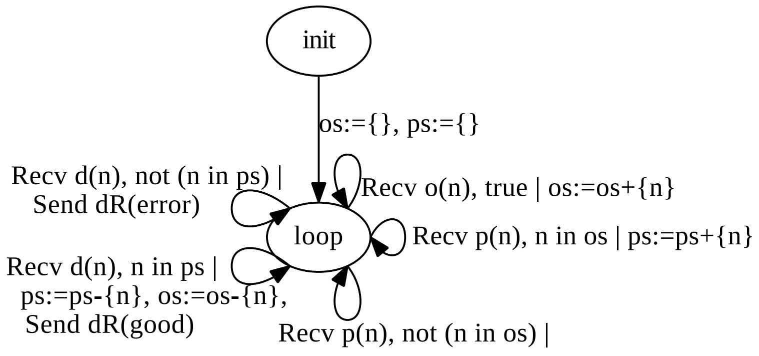
init
os:={}, ps:={}
Recv d(n), not (n in ps) |
Send dR(error)
Recv o(n), true | os:=os+{n}
loop
Recv p(n), n in os | ps:=ps+{n}
Recv d(n), n in ps |
ps:=ps-{n}, os:=os-{n},
Send dR(good)
Recv p(n), not (n in os) |

Figure 1: Example: EFSM model of simple Shop server

construction, inter-procedural control flow graph construction, and data flow analysis. For implementation we focused on systems implemented in the Java programming language and we used the Watson Libraries for Analysis (WALA) (IBM, 2010).

In practice, only analyzing the application code of real distributed or service-oriented systems will not directly produce a CFG such as the abstract one needed by the model construction algorithm. This is because real applications are usually developed with the help of special frameworks and APIs that help the application developer cope with the complexity of such systems. Two immediate consequences are:

- Instead of explicit *SendMessage* and *ReceiveMessage* instructions, frameworks offer complex APIs to describe the interactions of a server.

  The first step towards applying our model extraction method is to identify for each API the constructions which are equivalent with sending and receiving messages and define abstractions for them.

- Frameworks also provide infrastructure support for the execution of developed applications. Most often, by analyzing only the application code written by the application developer one cannot obtain the whole CFG of the real system. For example, in all frameworks the application developer does not explicitly provide the server loop, which is added by default through the framework.

  The particularities of each framework have to be known and the partial CFG or CFGs extracted from the application code must be completed or combined in order to obtain the complete CFG.

These issues (identifying and abstracting send/receive message operations, completing the partial CFG from application code) have to be solved by technology specific preprocessing frontends before the generic model construction method presented in 3.2 may proceed.

We consider modeling servers which are implemented in Java and according to a set of specific technologies. We categorize these technologies as being with or without explicit interfaces. Technologies such as WSDL Web Services, Java RMI, and CORBA, make the interfaces of the services explicit, either as language interfaces or as interfaces described in a special interface description language. Other technologies such as Servlets or JSP do not make the interfaces explicit.

4.1 Preprocessing frontend for interface-explicit technologies

In the case of Java RMI, but also in case of other interface-explicit technologies, a server is a special kind of object, implementing the methods described in an explicit interface. The interface description contains the list of possible operations, with their full signature (method name, number and types of parameters, return type). Clients can interact with a server invoking these methods. These are the entrypoints of the server application.

The entry points for a RMI application are those methods declared in an interface that extends the `rmi.Remote` interface. When analyzing an application that uses RMI, the preprocessing frontend looks for this kind of methods as entrypoints.

We can define the needed *SendMessage* and *ReceiveMessage* abstractions in RMI code in the following way: A RMI object receives a message when one of its remote methods is invoked. Thus the entrypoint of every remote method is modeled as an abstract *ReceiveMessage* operation. A RMI object sends a message when returning from a remote method invocation or when raising an exception.

Names for message types are derived automatically from method names. The type of the sent message differs from the type of the received message corresponding to the method invocation (it is a return-methodname type of message). The parameters of the received message correspond to the arguments of the method. The parameters of the sent message correspond to the returned values or exceptions raised.

For example, a method with following signature:

```
String deliver(String name) {
   ... // some statements
}
```

will be abstracted to:

```
ReceiveMessage deliverType, name
... // some statements
SendMessage deliverResp, aString
```

By analyzing the RMI application code, the CFGs of each entrypoint method can be built. In order to get the whole CFG of the RMI server, all these partial CFGs have to be framed by a server loop and preceded by the initialization code. After these preprocessing are done, the core model construction algorithm can be applied on the adjusted CFG.

4.2 Preprocessing frontend for servlets and JSP

Web applications are dynamic extensions of web or application servers, which may generate interactive web pages with dynamic content in response to requests. In the Java EE platform, the web components which provide these dynamic extension capabilities are either Java servlets or Java Server Pages (JSP).

A servlet is a Java class that conforms to the Java Servlet API, which establishes the protocol by which it responds to HTTP requests, and generates dynamic web content as response. The popular JSP technology, which embeds Java code into HTML, relies on Servlets, as these are automatically generated by the application server from JSP pages. When analyzing JSP pages, we first explicitly call the JSP compiler in order to obtain the source code of their corresponding servlet classes.

In the code analysis, we identify Java Servlets as the classes that extend the `javax.servlet.HttpServlet` class. Their entrypoints are the methods: `doGet`, `doDelete`, `doHead`, `doOptions`, `doPost`, `doPut`, `doTrace`, `service`. The servlets generated from JSP are classes which extend `org.apache.jasper.runtime.HttpJspBase` and their entrypoints are methods `jspInit()` and `jspService()`.

When analyzing an application that uses servlets, the preprocessing frontend looks for this kind of methods as entrypoints. Similarly to the RMI preprocessor, the CFGs of each entrypoint can be built and in order to get the whole CFG all these partial CFGs have to be framed by a server loop and preceded by the initialization code.

All entrypoint methods have as parameters `HttpServletRequest` and `HttpServletResponse`, which correspond to the types of the messages which are sent and received.

The `HttpServletRequest` allows access to all incoming data. The class has methods for retrieving form (query) data, HTTP request headers, and client information. The `HttpServletResponse` specifies all outgoing information, such as HTTP status codes, response headers, cookies, and also has a method of retrieving a `PrintWriter` used to create the HTML document which is sent to the client.

What is different and more difficult in this case is abstracting the parameters of the SendMessage and ReceiveMessage statements. Message parameters cannot be identified directly in this case, since incoming and outgoing data are handled through a large number of specific methods on the request and response objects.

As an example, we consider below an excerpt of the simple Shop, whose abstract control flow has been described in Section 3.3, this time in an implementation with servlets:

```
public class Shop extends HttpServlet {
  // ... omitted parts
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse resp)
    // ...  parts are omitted or simplified
   String op = req.getParameter("operation");
   if (op.equals("deliver"))   {
       String name=req.getParameter("name");
       if (payments.contains(name))
            delivResp=doService();
       else delivResp=error();
       Writer w=response.getWriter();
       w.write(delivResp);
       }
   else if (op.equals("pay"))
   // ...
```

Abstracting send and receive operations with parameters from the code of each entry point method is a complex task which must take into account every method that can be called on a `HttpServletRequest` or `HttpServletResponse` object.

The entrypoiny of the method corresponds to a *ReceiveMessage* statement, receiving a message of type *HttpRequest*. The parameters of this received message may contain: a set of name - value pairs, corresponding to the `ParameterMap`, and a set corresponding to the Session attributes. The parameters will be added to the ReceiveMessage statement only if they are used in the method body: the first parameter will be added only if the method body contains statements for retrieving the ParameterMap or specific parameters from the request object. The second parameter will be added only if there are statements retrieving a Session from the request object and getting values from there.

In our example, we have only calls of method `getParameter` on the `request` object, no `Session` object has been retrieved and used, thus the received abstract message is:

```
ReceiveMessage HttpRequest (
   ("operation", op), ("name", name))
```

Each path leading to an exit point of the method will end in a *SendMessage* statement, sending a message of type *HttpResponse*. The parameters of this sent message are: all the variables which are written by the output Writer along this path, and session attributes if they have been retrieved and handled in the method body.

In our example, following *SendMessage* statements are abstracted on the different paths:

```
SendMessage HttpResponse (delivResp)
SendMessage HttpResponse ("Order finished")
SendMessage HttpResponse ("Pay finished")
```

5 CONCLUSIONS

The goal of our work is to build a tool for the automatic extraction of behavioral models from service implementations. In order to cope with the diversity of technologies and APIs which can be used by service implementations, we propose an approach for model extraction in two steps: a technology-dependent preprocessing step, followed by the stable core step that implements a general method of transforming the abstracted control flow graph into an EFSM.

The kind of EFSM inferred by our approach is suitable to be automatically translated into an entity description in a formal security specification language for distributed systems such as Aslan++, the language used by the SPaCIoS tool. The security analyst will have to add manually only the security-related properties of the communication channels, which cannot be known from the implementation code, and to specify the desired properties to be checked.

Having tools which extract behavioral models from actual service implementations is an important step towards enabling formal security validation techniques to be applied on real systems at their implementation and deployment time.

ACKNOWLEDGEMENTS

This work has been supported by the FP7-ICT-2009-5 project no. 257876 SPaCIoS (”Secure Provision and Consumption in the Internet of Services”)

REFERENCES

Alur, R., Černý, P., Madhusudan, P., and Nam, W. (2005). Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’05, pages 98–109, New York, NY, USA. ACM.

Bertolino, A., Inverardi, P., Pelliccione, P., and Tivoli, M. (2009). Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE ’09, pages 141–150, New York, NY, USA. ACM.

Hossen, K., Groz, R., and Richier, J. (2011). Security vulnerabilities detection using model inference for applications and security protocols. In *IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*, pages 534–536.

IBM (2010). T.J.Watson Libraries for Analysis (WALA). Technical report, IBM T.J.Watson Research Centre.

Lorenzoli, D., Mariani, L., and Pezze, M. (2008). Automatic generation of software behavioral models. In *Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference on*, pages 501–510.

Merten, M., Howar, F., Steffen, B., Pellicione, P., and Tivoli, M. (2012). Automated inference of models for black box systems based on interface descriptions. In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 79–96. Springer Berlin Heidelberg.

Oheimb, D. and Modersheim, S. (2012). Aslan++ a formal security specification language for distributed systems. In Aichernig, B., Boer, F., and Bonsangue, M., editors, *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg.

Shoham, S., Yahav, E., Fink, S., and Pistoia, M. (2008). Static specification mining using automata-based abstractions. *Software Engineering, IEEE Transactions on*, 34(5):651–666.

Vigano, L. (2012). Towards the secure provision and consumption in the internet of services. In Fischer-Hobner, S., Katsikas, S., and Quirchmayr, G., editors, *Trust, Privacy and Security in Digital Business*, volume 7449 of *Lecture Notes in Computer Science*, pages 214–215. Springer Berlin Heidelberg.