SPaCIoS

**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 D2.2.1
# Method for Assessing and Retrieving Models


## Abstract

This deliverable defines algorithms and their integration in the framework confronting results of tests with the initial models of a system, so as to provide an assessment of the accuracy of the models and an automatic process to refine the models based on test observations.


## Deliverable details


Deliverable version: *v1.0*
Classification: *public*
Date of delivery: *31.03.2013*
Due on: *31.03.2013*
Editors: *All*
Total pages: *74*


## 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

# Contents

# List of Figures

# List of Tables

# 1 Introduction

The SPaCIoS approach with model-based property-driven test generation assumes that all the components of a service are modeled in ASLan++ [6, 41]. However, writing such models requires some expertise in ASLan++ as well as knowledge about the intended and actual behavior of each component. There are many cases where this expertise of knowledge might be scarce or unavailable, especially in the case of services that are built from components coming from different sources. It is also common experience that modelling can be misguided by expectations about the intended behavior of a subsystem, in which case the model may not match the implemented behavior.

The SPaCIoS tool includes two approaches to address this concern, by inferring models automatically from the actual software artifacts.

- This deliverable presents the method used to infer a model from external observations made by interacting with a software component (black box inference: BB).
- Deliverable D2.2.2 [37] presents methods to extract a model from its source code when available (white-box inference: WB); it also addresses the combination of both approaches, white-box and black-box.

In some cases, several models can be provided for a component: an initial model can be provided by a security analyst, and models can be extracted automatically in white or black box mode. This makes it possible to combine information from various sources and adjust models. For instance feasible paths are more easily found by dynamic black box testing, which can be combined with white box extraction to lessen the impact of overapproximation. And source code analysis can help in identifying the relevant inputs to be considered at each step in black box inference. This is presented in [37]. In another direction, the models derived automatically can also be refined by additional observations coming from further testing, as will be presented in the current deliverable. And of course, a security analyst can enrich the models derived automatically with semantic information, as well as compare them with his intuitions or an initial model.

Automatic black box inference only works from sampled behaviors and cannot capture the full semantic information that can be provided in an ASLan++ model. Therefore the models derived by automatic inference methods either in WB or BB will not cover the full features of ASLan++. However, both WB and BB in the SPaCIoS tool use the same format to represent the same subset of ASLan++.

The main inference method developed in the SPaCIoS project is described in Section 2. It is based on an extended finite state model described in Section 2.1, which takes into account key security features such as nonces, session IDs etc. This method of inference is called here "tabular model inference" because it is based on arranging observations into tables that record sequences of outputs observed in response to input sequences that label the rows and columns. These observation tables have long been used in grammatical inference, and the algorithm proposed here extends Angluin-style inference [4] to the complex EFSM model described in Section 2.1.

In the course of the project, we also came up, with the help of Alexandre Petrenko from the SPaCIoS expert group, with a new inference method, which no longer organizes observations in tables, but in a tree structure. As this second method is based on comparing the states reached during testing with a refinable equivalence relation from which the model is derived as a quotient, this second method is called "quotient model inference". Although this method is more recent and not yet as mature for SPaCIoS, because it currently handles only an FSM subset of the EFSM model of Section 2.1, we present it in this deliverable as a promising approach that could be more flexible than tabular inference.

In all cases, automatic inference algorithms and tools need to interact with a black box system which in the case of SPaCIoS is typically a web application. Inference algorithms must be provided with a description of the inputs accepted by the application. They work with abstract representations of the inputs and outputs of the system under inference (SUI) [1]. It is time consuming to write the drivers that map abstract symbols to concrete inputs and vice-versa for outputs. Therefore Section 4 presents a method that was developed in SPaCIoS to automatically derive such drivers from an initial crawling phase on the application.

Finally, Section 5 compares the models that are generated automatically by black and white box inference methods with models that were handwritten for two case studies of SPaCIoS.

# 2 Tabular Model Inference for EFSM

Under this name, we describe the method used for the inference of Extended Finite State models. This method is initially inspired by Angluin's L* algorithm [4]. L* learns a Deterministic Finite Automaton accepting a regular language by querying a "teacher" (which corresponds to our system under inference SUI) with sample words. Therefore the teacher answers "membership queries" replying whether a proposed word belongs to the language. L*

provides words that are built by concatenating prefix strings (access strings for candidate states) with suffix strings (to distinguish states). Prefixes label rows while suffixes label columns of tables, so that the cells of the table record results of the queries. This structure is called an observation table.

In our context, the system under inference can be queried by sending sequences of protocol inputs (e.g. HTTP requests for a web interface) and observing corresponding outputs (e.g. HTML pages). Since protocol data units have complex structures, they are best modeled as parameterized inputs, where the input type could correspond to a PDU (protocol data unit) type. Moreover, past values of parameters such as session IDs need to be recorded, and the response to a given input PDU may depend on the values of its parameters as well as the recorded values. This naturally leads to trying to infer EFSM models, Finite State machines extended with parameterized inputs and outputs, variables, and transitions that can depend on guards on these values, while updating variables and producing computed output parameters.

There must be some restrictions to the type of models that can be inferred, because, with general EFSM, it is impossible to ascribe a single model for any language on its inputs and outputs. The exact model that is used is presented in Section 2.1.

Whereas algorithms directly based on L* just infer the control part of an EFSM, we have to deal with the data values observed in the input and output parameters. Therefore our algorithm builds two observation tables:

- a control table, which can be seen as the structure of the machine when abstracting (partially) from data values,
- and a data table that records parameter values.

Of course, there are strong relations between the two tables, since they record two aspects of the same interactions with the SUI:

- they have the same dimensions and labeling of rows and columns;
- consequently, whenever data values distinguish between sequences of parameterized inputs that have the same type, this distinction may lead to splitting rows in both tables.

As in the case of L*, there are rules for extending a table with new rows or columns until it satisfies a certain number of properties that ensure that an EFSM model consistent with all observations can be built. L* has two such properties (closed and consistent). Our algorithm requires 3 properties (closed, balanced and dispute-free) which are detailed in Section 2.2.3. Other

aspects of the algorithm are presented in the rest of Section 2.2. The approach is incremental: once a first consistent model has been built, new observations from testing further with the SPaCIoS tool, or from tests suggested by the security analyst may lead to a refinement of the initial conjectured EFSM.

Section 2.3 illustrates how the algorithm works on simple case studies. Finally, Section 2.4 presents how the SPaCIoS approach compares with related work for state model inference.

## 2.1 Extended Finite State Machine (EFSM) Model

### 2.1.1 Basic Definitions

In the inference environment, the System Under Inference (SUI) communicates with a tester. In order to make the communication feasible, we need to configure the public keys and private keys on both sides. We can safely assume that the encryption and decryption functions are implemented correctly in both sides. Thus the information about encryption, decryption, and the keys is not necessary for inference and could be excluded from the model.

As the inference is performed in a controlled environment, some constants such as the identity of a communicating entity are known and preconfigured. Thus, the information about these constants could also be excluded from the model.

More generally, we rely on a mapping between concrete messages or events that can contain many parameters and complex encoding that are not relevant for the security testing task, and abstract events and parameters that are fed into the inference algorithm. This mapping approach has already been investigated by [1]. An abstract event will consist of an input or output symbol (typically a message type) and its relevant parameters.

As mentioned before, we introduce variables in the finite state model to store some context information. By analyzing typical software and protocols in the domain of software security, we make the following assumptions about the variables:

- The number of variables equals the number of all the input and output parameters, one corresponding to each parameter.
- In each transition, the latest value of a parameter is assigned to the corresponding variable. This is the only allowed assignment operation on the variables.

The first assumption is justified by the fact that we consider that only exchanged values will affect the behavior of the protocol. Although some internal bookkeeping on local resources may affect the behavior in some cases,

we consider only information directly related to the protocol.The second assumption prevents the reuse of past values: only the latest value can be stored and reused later in the computation of a guard or of an output parameter. We consider that this restriction would often be satisfied by security protocols: typically, a cookie is superseded by any new value, an old nonce is never significant, etc.

With these assumptions in mind, we can follow the traditional model of EFSM as defined in [30] and extend it with the concept of non-deterministic output parameter values.

In the following definitions, let $X$ and $Y$ be finite sets of input and output symbols, $P$ and $V$ be finite disjoint sets of $n$ parameters and variables such that $|P| = |V| = n$. Then $P = \{p_1, \ldots, p_n\}$ and $V = \{v_1, \ldots, v_n\}$, so we can associate a parameter with a variable having the same index. For $z \in X \cup Y$, we note $p(z) \subseteq P$ the set of its (associated) parameters and $D_z$ the set of valuations of parameters in the set $p(z)$. We lift these notations to sequences $\sigma$ of inputs and outputs. So $p(\sigma)$ is the sequence of sets of parameters, and $D_\sigma$ is a sequence of parameter valuations. Similarly, $D_V$ is a set of valuations of variables $V$. Note that multiple parameters could be associated with one input or output symbol. This helps to make the formulation of security protocols easy and straightforward.

An Extended Finite State Machine (EFSM) $M$ over $X$, $Y$, $P$, $V$, and the associated function $p$ and valuations is a pair $(S, T)$ of a finite set of states $S$, which contains an initial state $s_0$, and a finite set of transitions $T$ between states in $S$, such that each transition $t \in T$ is a tuple $(s, x, G, op, y, up, s')$, where

- $s, s' \in S$ are the initial and final states of the transition, respectively;
- $x \in X$ is the input symbol of the transition;
- $y \in Y$ is the output symbol of the transition;
- $G$, $op$, and $up$ are functions, defined over input parameters and variables $V$, namely,
  - $G : D_x \times D_V \to \{True, False\}$ is the guard of the transition;
  - $op : D_x \times D_V \to D_y$ is the output parameter function of the transition;
  - $up : D_x \times D_V \to D_V$ is the update function of the transition.

In a state $s \in S$, an input symbol $x \in X$ may not be accepted by the EFSM, e.g., in a certain page of a web application, there does not exist

an input field for some information. In this case, we use a transition with a special output symbol $\Omega \in Y$ to represent that the input symbol is not accepted by the EFSM. When an input or output symbol $z \in X \cup Y$ does not have a parameter, i.e., $p(z) = \{\}$, we use a parameter value $\omega$ to represent $d_z(p(z))$.

According to the assumptions about variables, suppose the valuation of parameters of $x$ is $d$, the valuation of variables $V$ is $v$, and the valuation of parameters of $y$ according to the $op$ function is $d' = op(d, v)$, the function $up$ changes variables corresponding to parameters of $x$ and $y$ to $d$ and $d'$ respectively, and keeps the current values of the other variables. If $x$ and $y$ share a common parameter, the corresponding variable is updated with the value of the output parameter. Formally, for $d \in D_x$, $v \in D_V$, then for any variable $v_i$:

- if $p_i \in p(y)$ then $up(d, v)(v_i) = op(d, v)(p_i)$,
- else if $p_i \in p(x)$ then $up(d, v)(v_i) = d(p_i)$,
- else $up(d, v)(v_i) = v(v_i)$ (in that case the value of $v_i$ is left unchanged).

Following the traditional EFSM, we have the following additional definitions.

Given input $x$ and an input parameter valuation $d$, a *parameterized input* is denoted as $x(d(p(x)))$. A sequence of parameterized inputs is called a *parameterized input sequence*. Similarly, we define parameterized outputs and their sequences.

An EFSM $M$ is said to be *deterministic* if any two transitions outgoing from the same state with the same input have mutually exclusive guards; *observable* if, for each state and each input, every outgoing transition with the same input has a distinct output. In this document, we assume the EFSM to be inferred is deterministic and observable.

### 2.1.2 Additional Considerations about Non-Deterministic Output Parameter Values

According to the definition of *op* in EFSM, in a transition, the output parameter value is decided by input parameter values and variable values. But in security protocols, there are some cases where the output parameter values are not deterministic with respect to input parameter values and variable values. Nonce is such a case.

Nonces are cryptographic inputs with the property that each value only occurs once within a given context [44]. Many modern cryptographic algo-

rithms require a key and a nonce as input, and as long as the key is unchanged, the nonce must not repeat. Examples for cryptographic solutions that require nonces are stream ciphers, certain block cipher modes of operation, some message authentication codes, and certain entity authentication solutions.

Session IDs and cookies are two other examples of such non-deterministic output parameter values.

When these output parameters are included in EFSM, since their values do not only depend on the values of input parameters and variables, *op* is no longer a function. In order to express them in the EFSM model, we introduce a special output parameter value *ndv* to represent a non-deterministic value. In this way, *op* is still a function with a unique valuation of output parameters for each valuation of input parameters and variables. Correspondingly, in the model inference procedure, we need to recognize that a certain output parameter value is *ndv*. In the formal model, the value recorded in the corresponding variable is *ndv*, but the mapping function to the concrete level will keep track of the corresponding concrete value.

## 2.2 Model Inference for EFSM

As usual, we assume that the SUI could be reset to its initial configuration after applying each input sequence, and the set of input symbols and the associated parameters of each input symbol are known.

In this section, we use the model depicted in Figure 1 as running example of SUI for it helps to present the inference algorithm step by step. In the model $X = \{a, b\}$, $Y = \{c, d, e, \Omega\}$, input parameter of $a$ is $p_1$, input parameter of $b$ is $p_2$, output parameter of $c$ is $p_3$, output parameter of $d$ is $p_4$, and $e$ does not have output parameter. Correspondingly, $V = \{v_1, v_2, v_3, v_4\}$. In this example, all parameters range over integers.

### 2.2.1 Observation Tables

Two observation tables are used in inferring EFSM models. A *Control Table* is used to record the relationship between input strings and output symbols, while a *Data Table* is used to record the relationship between input strings, variable values, and output parameter values. Both tables have the same structure.

Let $U$ be the set of possible parameterized input strings. The structure of the Control Table $(S, R, E, C)$ and Data Table $(S, R, E, D)$ are defined as follows.

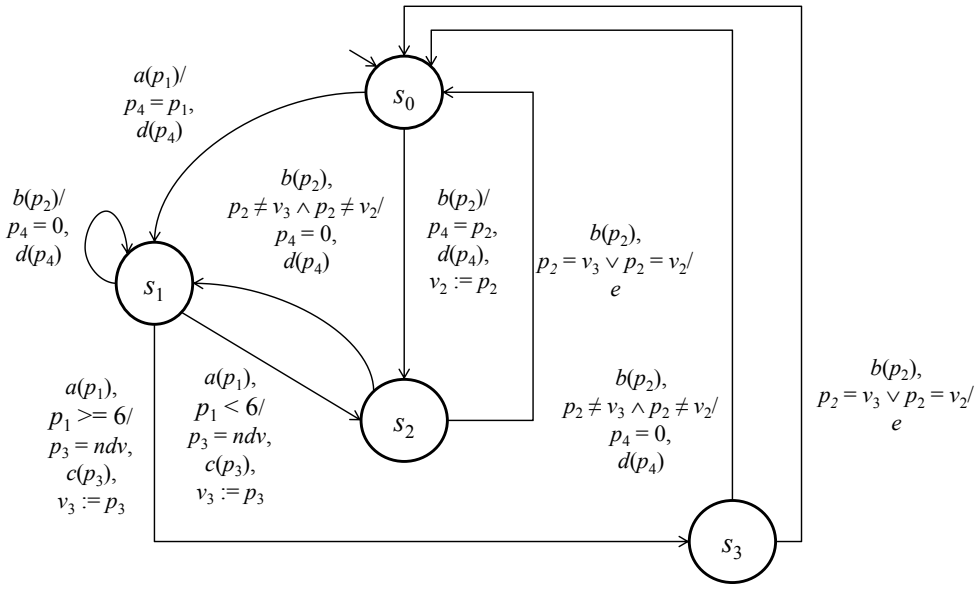


Figure 1: Running Example of SUI

$S \subseteq U$ and $R \subseteq U$ are nonempty finite sets of parameterized input strings that make the rows of the tables. $S$ is a prefix-closed set that is used to identify potential states in the conjecture and $R$ is used to explore one step further the behaviors of the SUI.

$E = X$ initially makes columns of the observation tables and is used to separate potential states of the conjecture. Note that in the procedure to obtain the first conjecture as described in this document, $E$ stays as $X$. Thus we write $x$ for an element of $E$. But generally an element of $E$ could be a sequence.

In the Control Table, each cell indexed by $s \in S \cup R$ and $x \in E$ is a set of elements in the form of $(d(p(x)), y)$, in which $d$ is a parameter valuation of $x$ and $y$ is an output symbol. In the Data Table, each cell indexed by $s \in S \cup R$ and $x \in E$ is a set of elements in the form of $(d(p(x)), v(V) \rightarrow d'(p(y)))$, in which $d$ is a parameter valuation of $x$, $v$ is a variable valuation, and $d'$ is a parameter valuation of $y$.

Formally, we define $C$ as a finite function mapping $((S \cup R) \times E)$ to the power set of $\{(d(p(x)), y) \mid x \in X, d \in D_x, y \in Y\}$. $D$ is a finite function mapping $((S \cup R) \times E)$ to the power set of $\{(d(p(x)), v(V) \rightarrow d'(p(y))) \mid x \in X, d \in D_x, v \in D_v, y \in Y, d' \in D_y\}$.

In the beginning of the inference procedure, $S$, $E$ are initialized as $S = \{\varepsilon\}$

and $E = X$. $R$ is initialized as follows: if an input symbol does not have parameter, it is included in $R$, otherwise, it is associated with only one parameter value and included in $R$. Suppose $X = \{x_i \mid 1 \leq i \leq k\}$, then $R = \{x_i \mid p(x_i) = \{\}, 1 \leq i \leq k\} \cup \{x_i(d_i(p(x_i))) \mid p(x_i) \neq \{\}, d_i \in D_{x_i}, 1 \leq i \leq k\}$. $S$ and $R$ will be extended during the inference procedure.

For the running example, initially, $S = \{\varepsilon\}$, $E = \{a, b\}$, $R$ could be $\{a(5), b(2)\}$.

After initialization, $S \cup R$ is prefix-closed. From the following description of how $S$ and $R$ are extended, we can see that in the whole inference procedure, $S \cup R$ is always prefix-closed.

### 2.2.2 Test Sequence Construction and Observation Recording

During the model inference procedure, test sequences are constructed as follows: for $s \in S \cup R$, $x \in E$, if $x$ has parameter, a parameter valuation $d \in D_x$ is selected to construct an input sequence $s \cdot x(d(p(x)))$, otherwise, $s \cdot x$ is the input sequence. For each combination of $s$ and $x$, multiple input sequences can be constructed.

Test data selection in this security context will be addressed in a later section. In this section, we just assume some test sequences are applied to the SUI, and corresponding output sequences are observed. The observations are recorded in observation tables as follows.

Suppose for input sequence

$$x_{i_1}(d_{i_1}(p(x_{i_1}))) \; x_{i_2}(d_{i_2}(p(x_{i_2}))) \; \ldots \; x_{i_m}(d_{i_m}(p(x_{i_m}))),$$

output sequence

$$y_{j_1}(d'_{j_1}(p(y_{j_1}))) \; y_{j_2} \; (d'_{j_2} \; (p(y_{j_2}))) \; \ldots \; y_{j_m}(d'_{j_m}(p(y_{j_m})))$$

is observed, with $x_{i_k} \in X$, $y_{j_k} \in Y$, for $1 \leq k \leq m$. During recording the observation, we keep a variable valuation $v$ with all the variable values initialized with 0 (regardless of their types, this simply denotes the initial value).

From the test sequence construction, we can see that

$$x_{i_1}(d_{i_1}(p(x_{i_1}))) \; x_{i_2}(d_{i_2}(p(x_{i_2}))) \; \cdots \; x_{i_{m-1}}(d_{i_{m-1}}(p(x_{i_{m-1}}))) \in S \cup R.$$

Because $S \cup R$ is prefix-closed, all its prefixes also belong to $S \cup R$.

Starting from $k = 1$, for each pair of input/output

$$x_{i_k}(d_{i_k}(p(x_{i_k})))/y_{j_k}(d'_{j_k}(p(y_{j_k}))),$$

| | | $E = X$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(\mathbf{5}, \mathbf{d}), (6, d)$ | $(2, d), (3, d)$ |
| $R$ | $a(5)$ | $(5, c)$ | $(\mathbf{2}, \mathbf{d})$ |
| | $b(2)$ | $(5, \Omega)$ | $(3, d)$ |

Table 1: Control Table

| | | $E = X$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(\mathbf{5}, (\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}) \to \mathbf{5})$, $(6, (0, 0, 0, 0) \to 6)$ | $(2, (0, 0, 0, 0) \to 2)$, $(3, (0, 0, 0, 0) \to 3)$ |
| $R$ | $a(5)$ | $(5, (5, 0, 0, 5) \to 100)$ | $(\mathbf{2}, (\mathbf{5}, \mathbf{0}, \mathbf{0}, \mathbf{5}) \to \mathbf{0})$ |
| | $b(2)$ | $(5, (0, 2, 0, 2) \to \omega)$ | $(3, (0, 2, 0, 2) \to 0)$ |

Table 2: Data Table

let $s$ be the input prefix of length $k-1$, in the cell indexed by $s$ and $x_{i_k}$ in the Control Table, we add element $(d_{i_k}(p(x_{i_k})), y_{i_k})$, in the cell indexed by $s$ and $x_{i_k}$ in the Data Table, we add element $(d_{i_k}(p(x_{i_k})), v(V) \to d'_{j_k}(p(y_{j_k})))$, and update $v$ as follows: for any variable $v_l$ :

- if $p_l \in p(y_{j_k})$ then $v(v_l) = d'_{j_k}(p_l)$;
- else if $p_l \in p(x_{i_k})$ then $v(v_l) = d_{i_k}(p_l)$;
- else $v(v_l)$ keeps unchanged.

In the running example, suppose with input sequence $a(5)\ b(2)$, we observe output sequence $d(5)d(0)$. We keep a variable valuation $v$, $v(V)$ is initialized as $(0, 0, 0, 0)$.

We start from the first input/output pair $a(5)/d(5)$, in the Control Table, we add element $(5, d)$ in the cell indexed by $\varepsilon$ and $a$, in the Data Table, we add element $(5, (0, 0, 0, 0) \to 5)$ in the cell indexed by $\varepsilon$ and $a$, and update $v$ to make $v(V) = (5, 0, 0, 5)$.

Then, for the second input/output pair $b(2)/d(0)$, in the Control Table, we add element $(2, d)$ in the cell indexed by $a(5)$ and $b$, in the Data Table, we add element $(2, (5, 0, 0, 5) \to 0)$ in the cell indexed by $a(5)$ and $b$, and update $v$ to make $v(V) = (5, 2, 0, 0)$.

Table 1 is the initial Control Table filled with observed traces for the running example. Table 2 is the initial Data Table. The input sequences used and the corresponding output sequences observed are as follows:

$$a(5)/d(5),\ b(2)/d(2),\ a(6)/d(6),\ b(3)/d(3),\ a(5)a(5)/d(5)c(100),$$
$$a(5)b(2)/d(5)d(0),\ b(2)a(5)/d(2)\Omega,\ b(2)b(3)/d(2)d(0).$$

| | | $E = X$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(5, d), (6, d)$ | $(2, d), (3, d)$ |
| $R$ | $a(5)$ | $(5, c), (\mathbf{6}, \mathbf{c})$ | $(2, d), (\mathbf{3}, \mathbf{d})$ |
| | $b(2)$ | $(5, \Omega), (\mathbf{6}, \mathbf{\Omega})$ | $(\mathbf{2}, \mathbf{e}), (3, d)$ |

Table 3: Balanced Control Table

| | | $E = X$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(5, (0,0,0,0) \to 5)$, $(6, (0,0,0,0) \to 6)$ | $(2, (0,0,0,0) \to 2)$, $(3, (0,0,0,0) \to 3)$ |
| $R$ | $a(5)$ | $(5, (5,0,0,5) \to 100)$ $(\mathbf{6}, (\mathbf{5},\mathbf{0},\mathbf{0},\mathbf{5}) \to \mathbf{200})$ | $(2, (5,0,0,5) \to 0)$ $(\mathbf{3}, (\mathbf{5},\mathbf{0},\mathbf{0},\mathbf{5}) \to \mathbf{0})$ |
| | $b(2)$ | $(5, (0,2,0,2) \to \omega)$ $(\mathbf{6}, (\mathbf{0},\mathbf{2},\mathbf{0},\mathbf{2}) \to \omega)$ | $(\mathbf{2}, (\mathbf{0},\mathbf{2},\mathbf{0},\mathbf{2}) \to \omega)$ $(3, (0,2,0,2) \to 0)$ |

Table 4: Corresponding Data Table

### 2.2.3 Properties of Observation Tables

A conjecture of the SUI can only be constructed when certain properties are satisfied by the observation tables. In this section, we discuss these properties and how to make the observation tables satisfy them.

**Balanced** Two rows in the Control Table cannot be compared directly, if the observations are recorded for different input parameter values. The rows $s_1$ and $s_2$ are called *balanced*, if $C(s_1, x)$ and $C(s_2, x)$ contain the same input parameter values for each $x \in E$. The Control Table is *balanced*, if for all $s, t \in S \cup R$, $s$ and $t$ are balanced.

When two rows $s_1$ and $s_2$ are not balanced, there exist an input symbol $x$, and an input parameter valuation $d \in D_x$, such that $s_1 \cdot x(d(p(x)))$ has been applied to the SUI but $s_2 \cdot x(d(p(x)))$ has not, or vice versa. In this case, we can apply the input sequence $s_2 \cdot x(d(p(x)))$ and record the observation to make the two rows balanced.

In Table 1, the rows $\varepsilon$ and $a(5)$ are not balanced, because input parameter value 6 is not in the cell $C(a(5), a)$, and input parameter value 3 is not in the cell $C(a(5), b)$. We apply input sequences $a(5)a(6)$ and $a(5)b(3)$ and record the observations to make these two rows balanced.

After making each pair of rows balanced, we obtain a balanced Control Table as Table 3 and the corresponding Data Table as Table 4. The elements in bold font are new ones added to make the Control Table balanced.

**Equivalence** The rows $s_1$ and $s_2$ are called *equivalent*, denoted by $s_1 \cong s_2$, if they are balanced and contain the same set of output symbols for all $x \in E$.

| | | $E = X$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(5,d),(6,d)$ | $(2,d),(3,d)$ |
| | $a(5)$ | $(5,c),(6,c)$ | $(2,d),(3,d)$ |
| | $b(2)$ | $(5,\Omega),(6,\Omega)$ | $(2,e),(3,d)$ |
| $R$ | $\mathbf{a(5)a(5)}$ | $\mathbf{(5,\Omega),(6,\Omega)}$ | $\mathbf{(2,d),(3,d)}$ |
| | $\mathbf{a(5)b(2)}$ | $\mathbf{(5,c),(6,c)}$ | $\mathbf{(2,d),(3,d)}$ |
| | $\mathbf{b(2)a(5)}$ | $\mathbf{(5,\Omega),(6,\Omega)}$ | $\mathbf{(2,e),(3,d)}$ |
| | $\mathbf{b(2)b(2)}$ | $\mathbf{(5,d),(6,d)}$ | $\mathbf{(2,d),(3,d)}$ |

Table 5: Making Control Table Towards Closed

| | | $E = X$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(5,(0,0,0,0) \to 5)$,<br>$(6,(0,0,0,0) \to 6)$ | $(2,(0,0,0,0) \to 2)$,<br>$(3,(0,0,0,0) \to 3)$ |
| | $a(5)$ | $(5,(5,0,0,5) \to 100)$<br>$\mathbf{(5,(5,0,0,5) \to 300)}$<br>$\mathbf{(5,(5,0,0,5) \to 500)}$<br>$\mathbf{(5,(5,0,0,5) \to 600)}$<br>$\mathbf{(5,(5,0,0,5) \to 700)}$<br>$(6,(5,0,0,5) \to 200)$ | $(2,(5,0,0,5) \to 0)$<br>$(3,(5,0,0,5) \to 0)$ |
| | $b(2)$ | $(5,(0,2,0,2) \to \omega)$<br>$(6,(0,2,0,2) \to \omega)$ | $(2,(0,2,0,2) \to \omega)$<br>$(3,(0,2,0,2) \to 0)$ |
| $R$ | $a(5)a(5)$ | $\mathbf{(5,(5,0,300,5) \to \omega)}$<br>$\mathbf{(6,(5,0,500,5) \to \omega)}$ | $\mathbf{(2,(5,0,600,5) \to 0)}$<br>$\mathbf{(3,(5,0,700,5) \to 0)}$ |
| | $a(5)b(2)$ | $\mathbf{(5,(5,2,0,0) \to 900)}$<br>$\mathbf{(6,(5,2,0,0) \to 110)}$ | $\mathbf{(2,(5,2,0,0) \to 0)}$<br>$\mathbf{(3,(5,2,0,0) \to 0)}$ |
| | $b(2)a(5)$ | $\mathbf{(5,(5,2,0,2) \to \omega)}$<br>$\mathbf{(6,(5,2,0,2) \to \omega)}$ | $\mathbf{(2,(5,2,0,2) \to \omega)}$<br>$\mathbf{(3,(5,2,0,2) \to 0)}$ |
| | $b(2)b(2)$ | $\mathbf{(5,(0,2,0,2) \to 5)}$<br>$\mathbf{(6,(0,2,0,2) \to 6)}$ | $\mathbf{(2,(0,2,0,2) \to 2)}$<br>$\mathbf{(3,(0,2,0,2) \to 3)}$ |

Table 6: Corresponding Data Table

For $s \in S \cup R$, we denote by $[s]$ the equivalence class of rows that includes $s$.

**Closed** The Control Table is *closed*, if for every $t \in R$, there exists $s \in S$ such that $s \cong t$. When a Control Table is not closed, we make it towards being closed by moving $t$ from $R$ to $S$, extending $R$ with $|X|$ strings $\{t \cdot x_i(d_i(p(x_i))) \mid d_i \in D_{x_i}, 1 \leq i \leq k\}$ assuming $X = \{x_i \mid 1 \leq i \leq k\}$, constructing and applying input sequences for all the empty cells, and recording the observations.

Since the row $a(5) \in R$ is not equivalent to the only row $\varepsilon \in S$, Table 3 is not closed. We move $a(5)$ from $R$ to $S$, and extend $R$ with strings $a(5)a(5)$ and $a(5)b(2)$. By performing the same operation on the row $b(2)$, and filling the empty cells, we obtain Table 5 and Table 6. The new rows and new elements added are in bold font.

**Identifying *ndv* while Recording Observations** While recording observations in observation tables, we may encounter such a situation where

in a cell $D(s, x)$ of Data Table, there are two elements with the same input parameter values $d(p(x))$ and the same variable values, but different output parameter values. In this case, we can decide that the output parameter value is $ndv$, since this value is not deterministically decided by the input parameter values and variable values.

When an $ndv$ is identified, the following operations need to be performed:

- In the cell of the Data Table, merge these elements to one by changing the output parameter value to $ndv$.
- In the following inference procedure, whenever we construct an input sequence having $s \cdot x(d(p(x)))$ as prefix, for all the input parameters after the prefix, we need to select the actual non-deterministic value generated as input parameter values, in addition to other values selected. When interacting with real protocol entities, the actual value of $ndv$ is only applied if an input parameter has a compatible *type*, which is a concept omitted in this document for the sake of simplicity. When multiple $ndv$s are identified in a prefix, and multiple parameters are associated with one input symbol, there are several possible strategies to apply the $ndv$s generated. One strategy is to guarantee that each $ndv$ appears at least once for each parameter. Another one is to guarantee that all the possible combinations of $ndv$s appear. The selection of a specific strategy depends on the tradeoff between the resource available and the accuracy of the conjecture to be obtained. While recording the observation in the Control Table, $ndv_i$, in which $i$ is the index of the output parameter $p_i$ of the corresponding output symbol $y$, rather than the concrete value itself, is recorded as input parameter value.

In the running example, this is the case for the Data Table cell $D(a(5), a)$. So, we can decide that the output parameter value is $ndv$, and perform the following operations:

- We merge those elements to make the cell $\{(5, (5, 0, 0, 5) \rightarrow ndv), (6, (5, 0, 0, 5) \rightarrow 200)\}$.
- For the Data Table cell $D(a(5)a(5), a)$, we need to construct an additional input sequence taking $ndv$ generated as input parameter value. This time, when $a(5)a(5)$ is applied, $d(5)c(400)$ is observed. Thus, we apply $a(400)$ after that, and observe $\Omega$. So, in the Data Table, we add $(400, (5, 0, 400, 5) \rightarrow \omega)$ into the cell, and in the Control Table, we add $(ndv_3, \Omega)$ into the cell.

- For the Data Table cell $D(a(5)a(5), b)$, we also need to construct an additional input sequence taking $ndv$ generated as input parameter value. This time, when $a(5)a(5)$ is applied, $d(5)c(800)$ is observed. Thus, we apply $b(800)$ after that, and observe $e$. So, in the Data Table, we add $(800, (5, 0, 800, 5) \to \omega)$ into the cell, and in the Control Table, we add $(ndv_3, e)$ into the cell.

**Making the Table Balanced with Respect to *ndv*** With the introduction of $ndv_i$ in the Control Table, in which $i$ is the index of output parameter $p_i$ of output symbol $y$, the way of making the Control Table balanced needs to be updated with respect to $ndv_i$. For a cell $C(s, x)$, when a sequence like $s \cdot x(ndv_i)$ needs to be constructed in order to make the table balanced, we take the current value of $v_i$ after $s$ is applied as the input parameter value of $x$.

For example, in Table 5, for the cell $C(\varepsilon, a)$, we could use $a(0)$ as input sequence, observe output $d(0)$, and record $(ndv_3, d)$ in the Control Table, and $(0, (0, 0, 0, 0) \to 0)$ in the Data Table.

| | | $E = X$ | | $\cong$ |
|---|---|---|---|---|
| | | $a$ | $b$ | |
| $S$ | $\varepsilon$ | $(5, d), (6, d)$ $(ndv_3, d)$ | $(2, d), (3, d)$ $(ndv_3, d)$ | |
| | $a(5)$ | $(5, c), (6, c)$ $(ndv_3, c)$ | $(2, d), (3, d)$ $(ndv_3, d)$ | |
| | $b(2)$ | $(5, \Omega), (6, \Omega)$ $(ndv_3, \Omega)$ | $(\mathbf{2}, \mathbf{e}), (\mathbf{3}, \mathbf{d})$ $(\mathbf{ndv_3}, \mathbf{e})$ | |
| $R$ | $a(5)a(5)$ | $(5, \Omega), (6, \Omega)$ $(ndv_3, \Omega)$ | $(2, d), (3, d)$ $(ndv_3, e)$ | $b(2)$ |
| | $a(5)b(2)$ | $(5, c), (6, c)$ $(ndv_3, c)$ | $(2, d), (3, d)$ $(ndv_3, d)$ | $a(5)$ |
| | $b(2)a(5)$ | $(5, \Omega), (6, \Omega)$ $(ndv_3, \Omega)$ | $(2, e), (3, d)$ $(ndv_3, e)$ | $b(2)$ |
| | $b(2)b(2)$ | $(5, d), (6, d)$ $(ndv_3, d)$ | $(2, d), (3, d)$ $(ndv_3, d)$ | $\varepsilon$ |
| | $\mathbf{b(2)b(3)}$ | $(\mathbf{5}, \mathbf{c}), (\mathbf{6}, \mathbf{c})$ $(\mathbf{ndv_3}, \mathbf{c})$ | $(\mathbf{2}, \mathbf{d}), (\mathbf{3}, \mathbf{d})$ $(\mathbf{ndv_3}, \mathbf{d})$ | $a(5)$ |

Table 7: Final Control Table

**Dispute-Free** In EFSM, starting from a parameterized input sequence, different parameterized output sequences can be observed by inputting the same input symbol sequence associated with different input parameter value sequences. Thus, in one single cell of the Control Table, multiple output symbols could be included. In the conjecture, multiple transitions will be built from this information. In the Control Table, a row $s \in S$ is a *Disputed*

*Row*, if there exists $x \in E$ such that more than one output symbols are included in the cell indexed by $s$ and $x$.

A disputed row is *resolved* if for each $y \in Y$ involved in the cell $C(s, x)$, there exist $t = s \cdot x(d(p(x))) \in R$, and $(d(p(x)), y) \in C(s, x)$. In order to make well-defined transitions in the conjecture, when a disputed row $s$ is not resolved because such a $t$ does not exist in $R$ for a certain $y$, we extend $R$ with such a $t$. A Control Table is *dispute-free* if all the disputed rows in $S$ are resolved.

If both $(d(p(x)), y)$ and $(d'(p(x)), y)$ belong to $C(s, x)$, because we assume that the EFSM being inferred is observable, the states reached by $s \cdot x(d(p(x)))$ and $s \cdot x(d'(p(x)))$ from the initial state must be the same. So, if $R$ needs to be extended to resolve disputed row $s$, only one from $s \cdot x(d(p(x)))$ and $s \cdot x(d'(p(x)))$ is included in $R$.

In Table 5, output symbols $d$ and $e$ are involved in the cell $C(b(2), b)$. Thus, the row $b(2)$ is disputed. Since $b(2)b(2) \in R$, in order to resolve this disputed row, we only need to introduce a row $b(2)b(3)$ in $R$. After constructing and applying input sequences to make the table balanced, we obtain Table 7 and Table 8. Now, Table 7 is balanced, dispute-free, and closed. For each row in $R$, the corresponding equivalent row in $S$ is indicated in the right most column of the table.

| | | $E = X$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(5,(0,0,0,0) \to 5)$, $(6,(0,0,0,0) \to 6)$, $(0,(0,0,0,0) \to 0)$ | $(2,(0,0,0,0) \to 2)$, $(3,(0,0,0,0) \to 3)$, $(0,(0,0,0,0) \to 0)$ |
| | $a(5)$ | $(5,(5,0,0,5) \to ndv)$, $(6,(5,0,0,5) \to 200)$, $(0,(5,0,0,5) \to 120)$ | $(2,(5,0,0,5) \to 0)$, $(3,(5,0,0,5) \to 0)$, $(0,(5,0,0,5) \to 0)$ |
| | $b(2)$ | $(5,(0,2,0,2) \to \omega)$, $(6,(0,2,0,2) \to \omega)$, $(0,(0,2,0,2) \to \omega)$ | $(2,(0,2,0,2) \to \omega)$, $(3,(0,2,0,2) \to 0)$, $(0,(0,2,0,2) \to \omega)$ |
| $R$ | $a(5)a(5)$ | $(5,(5,0,300,5) \to \omega)$, $(6,(5,0,500,5) \to \omega)$, $(400,(5,0,400,5) \to \omega)$ | $(2,(5,0,600,5) \to 0)$, $(3,(5,0,700,5) \to 0)$, $(800,(5,0,800,5) \to \omega)$ |
| | $a(5)b(2)$ | $(5,(5,2,0,0) \to 900)$, $(6,(5,2,0,0) \to 110)$, $(0,(5,2,0,0) \to 130)$ | $(2,(5,2,0,0) \to 0)$, $(3,(5,2,0,0) \to 0)$, $(0,(5,2,0,0) \to 0)$ |
| | $b(2)a(5)$ | $(5,(5,2,0,2) \to \omega)$, $(6,(5,2,0,2) \to \omega)$, $(0,(5,2,0,2) \to \omega)$ | $(2,(5,2,0,2) \to \omega)$, $(3,(5,2,0,2) \to 0)$, $(0,(5,2,0,2) \to \omega)$ |
| | $b(2)b(2)$ | $(5,(0,2,0,2) \to 5)$, $(6,(0,2,0,2) \to 6)$, $(0,(0,2,0,2) \to 0)$ | $(2,(0,2,0,2) \to 2)$, $(3,(0,2,0,2) \to 3)$, $(0,(0,2,0,2) \to 0)$ |
| | $\mathbf{b(2)b(3)}$ | $\mathbf{(5,(0,3,0,0) \to 140)}$, $\mathbf{(6,(0,3,0,0) \to 150)}$, $\mathbf{(0,(0,3,0,0) \to 160)}$ | $\mathbf{(2,(0,3,0,0) \to 0)}$, $\mathbf{(3,(0,3,0,0) \to 0)}$, $\mathbf{(0,(0,3,0,0) \to 0)}$ |

Table 8: Final Data Table

### 2.2.4 Test Data Selection Strategies

How to select test data is generally a difficult issue, especially in the context of security testing. Here, we face the problem of parameter value selection in model inference for security testing. We tackle this issue from the following aspects.

- In the beginning of the inference procedure, we assume that there is some information resource such as domain expertise to provide relevant parameter values needed. In this way, we can have some usable parameter values for each input symbol to start the inference. In the running example, the 5 and 6 for $a$ and 2 and 3 for $b$ are this case. Note that although the initial $R$ is obtained by associating each input symbol with only one parameter value if there are parameters associated with the input symbol, in the cell indexed by $\varepsilon$ and $x \in X$, we are free to use multiple parameter values of $x$ to explore the behavior of the SUI.

- As described in Section 2.2.3, whenever $ndv$ is identified for a prefix $s \cdot x(d(p(x)))$, for all the input parameters after the prefix, we need to select the $ndv$ generated as input parameter values, in addition to other values selected. In addition to that, we need to make the observation tables balanced with respect to the $ndv$ identified.

- In the definition of EFSM used in this document, transition guard is defined as $G : D_x \times D_V \rightarrow \{True, False\}$, which means informally that input parameter values are "compared" with variables to decide whether certain transition is applicable. In order to identify all the outgoing transitions from a state, we introduce an assumption of the SUI and a corresponding parameter value selection strategy. We assume all the transition guards of SUI are logical combinations of atoms of the form $p_i = v_j$ or $p_i \neq v_j$, where $p_i \in P$ are input parameters, $v_j \in V$, $1 \leq i, j \leq n$. We name this assumption as "Direct Comparison Assumption". During the inference procedure, we already keep record of the current valuation $v$ of variables $V$, in order to fill the data table. Thus, for $s \in S \cup R$, $x \in X$, suppose the current valuation of $V$ is $v$, for each input parameter in $p(x)$, we could use the set $\{v(v_i) \mid v_i \in V, 1 \leq i \leq n\} \cup \{v^*\}$ where $v^* \neq v(v_i)(1 \leq i \leq n)$ as the possible values, and try all the possible combinations of these values as $d(p(x))$. We name this strategy as "All Data Strategy". When the direct comparison assumption is held in SUI, by using the all data strategy, we are able to find all the outgoing transitions from a state. At the same time, applying the all data strategy is clearly costly. While

inferring a specific protocol entity or service, a tradeoff could be defined based on further information or heuristics about the SUI.

### 2.2.5 Conjecture Construction

When the Control Table is balanced, dispute-free and closed, an EFSM conjecture $Q = (S_Q, T_Q)$ can be constructed from observation tables $C$ and $D$. There are three steps in the conjecture construction.

**Construct raw conjecture** As the first step, a raw conjecture $Q = (S_Q, T_Q)$ is constructed. The equivalent classes of strings in $S$ are states in $Q$. Formally, $S_Q = \{[s] \mid s \in S\}$. Specifically, the initial state is $s_{0Q} = [\varepsilon]$.

The set of transitions $T_Q$ is defined as follows: for each $[s](s \in S)$, $x \in X$, one or more transitions are defined, according to the number of different output symbols in the cell $C(s, x)$. Suppose one of the output symbols is $y$, the transition is $([s], x, G, op, y, up, s')$ as defined below.

- When there are multiple output symbols in the cell $C(s, x)$, the guard $G$ is introduced as follows:
  - in each cell of Data Table $D(t, x)$ $(t \in [s])$, there is a set of elements in the form of $(d(p(x)), v(V) \rightarrow d'(p(y)))$ corresponding to output symbol $y$, we construct $(d(p(x)), v(V) \rightarrow True)$ correspondingly;
  - for all the other elements, we construct $(d(p(x)), v(V) \rightarrow False)$ correspondingly;
  - the guard $G$ is the set containing all these elements constructed.
- In each cell of Data Table $D(t, x)$ $(t \in [s])$, there is a set of elements corresponding to output symbol $y$. The output parameter function $op$ is the set containing all these elements.
- The variable update function $up$ is defined as in the definition of EFSM.
- Since observation tables are dispute-free, there exists $t = s \cdot x(d(p(x)))$, $t \in S \cup R$, such that $(d(p(x)), y) \in C(s, x)$. We set the target state $s' = [t]$.

If, in the Control Table cell $C(s, x)$, there is an element $(ndv_i, y)$, we consider the following additional transitions.

- If there is already a transition from $[s]$ with output symbol $y$, in the guard of this transition, we add "$\vee p(x) = v_i$"; otherwise, create a transition as a normal one except the guard is "$p(x) = v_i$".

- In the guards of all the other transitions from $[s]$, we add "$\wedge p(x) \neq v_i$".

The raw conjecture constructed from Table 7 and Table 8 is depicted in Figure 2.

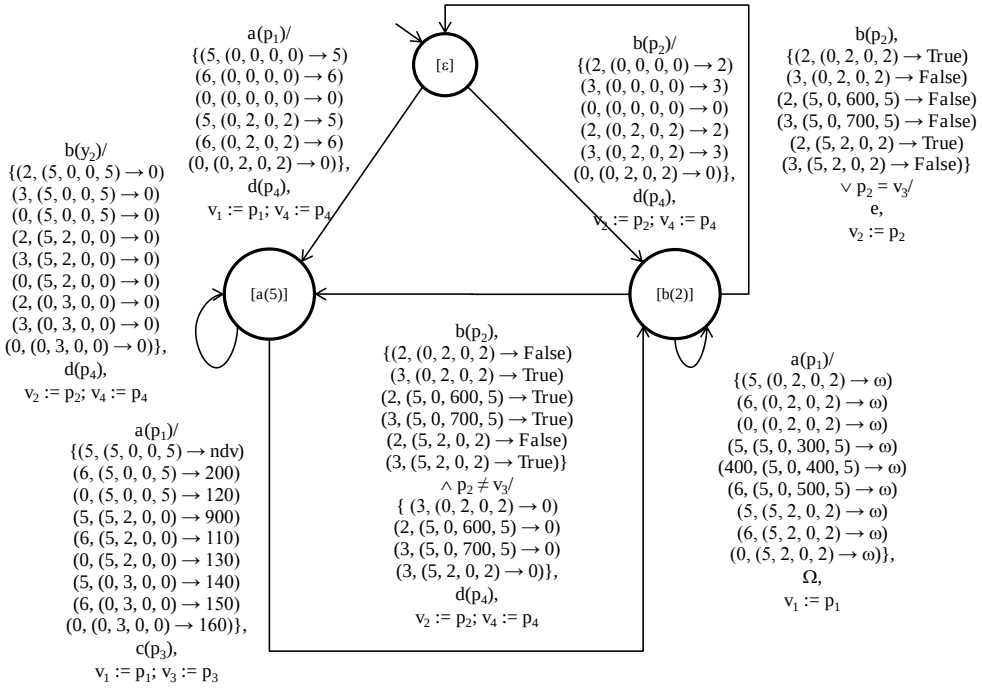


Figure 2: Raw Conjecture

**Infer guards and functions using data mining tools** In the raw conjecture, the guards and functions are in the form of sets, which are not convenient for a human developer or tester to understand. Providing these sets to tools such as Daikon [15] or the data mining library Weka [43], we could obtain a succinct form of these guards and functions. The algorithms and mechanisms of these data mining tools are out of the scope of this document, even out of the scope of testing techniques. Here, we only give the result obtained using two classifiers of Weka, `M5P` and `RandomTree`, for the running example in Figure 3.

**Clean unused variable assignments** From Figure 3 we can see that in the current form of conjecture, some variable assignments, e.g., all the assignments to variable $v_1$, are not useful, since $v_1$ is never used in any guard or function. We need to remove these assignments from the conjecture.

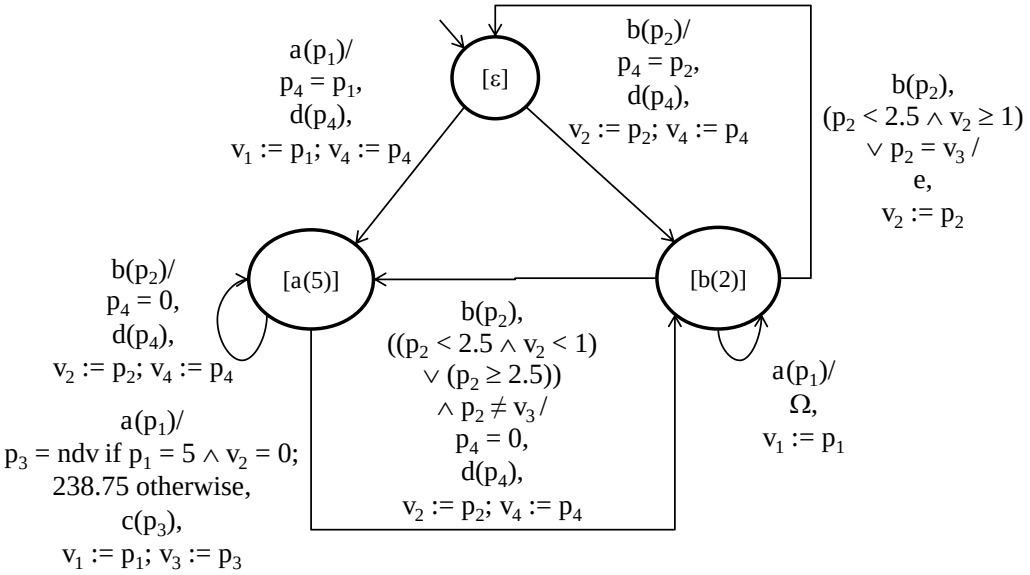


Figure 3: Succinct Form

This can be done as follows: For each variable assignment in the form of $v_i := d(p(z))(z \in X \cup Y)$, we find all the paths to the next assignment of $v_i$ in the form of $v_i := d'(p(z'))(z' \in X \cup Y)$. If $v_i$ is not used in all the guards and output parameter functions of all these paths, we remove the first assignment $v_i := d(p(z))$. After this cleaning, the final conjecture of the running example is depicted in Figure 4.

### 2.2.6 Outline of the Inference Algorithm

In summary, the initial inference algorithm (before new observation processing which corresponds to counterexample processing in Angluin's algorithm) is outlined in Listing 1.

### 2.2.7 Analysis of the Inference Algorithm

Informally, we say an EFSM is consistent with the observation tables if for each sequence whose execution is recorded in the observation tables, there exists a path in the EFSM corresponding to the execution.

In order to define the concept of *consistent* formally, we introduce some auxiliary notations. Assume $s$ is a sequence in $S \cup R$ of length $k$, i.e., $s \in S \cup R$, $s = x_{i_1}(d_{i_1}(p(x_{i_1})))\ x_{i_2}(d_{i_2}(p(x_{i_2})))\ \cdots\ x_{i_k}(d_{i_k}(p(x_{i_k})))$ $(x_{i_l} \in X, d_{i_l} \in D_{x_{i_l}}, 1 \leq l \leq k)$. We denote the prefix of $s$ of length $l$ as $s^l$, i.e., $s^l =$

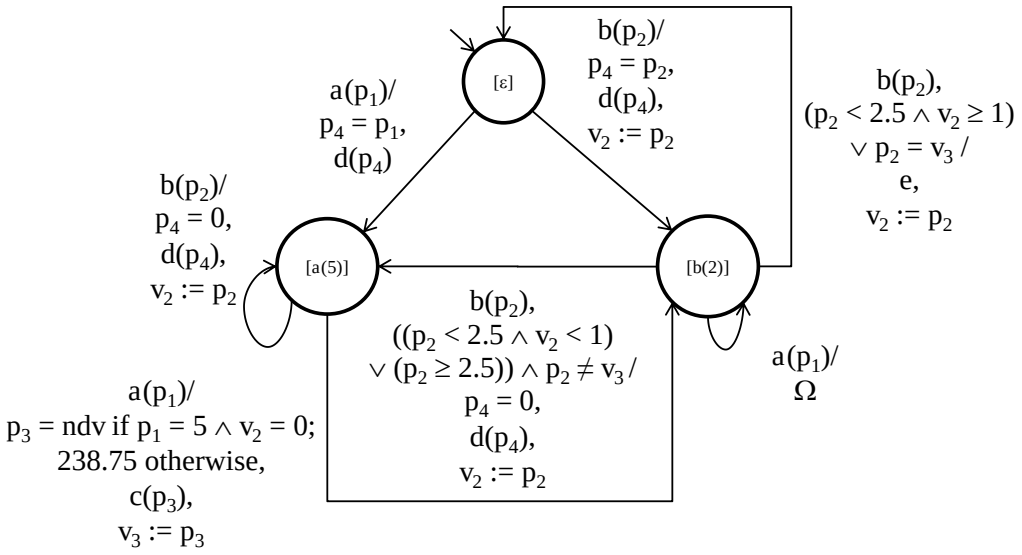


Figure 4: Final Conjecture

$x_{i_1}(d_{i_1}(p(x_{i_1})))x_{i_2}(d_{i_2}(p(x_{i_2})))\cdots x_{i_l}(d_{i_l}(p(x_{i_l})))$. Thus we have $s^0 = \varepsilon$, and $s^k = s$.

We assume the initial valuation of variables $V$ is $v^0$. We denote $v^k$ as the valuation of $V$ after applying input sequence $s$ of length $k$, which is defined recursively as $v^l = up(d_{i_l}(p(x_{i_l})), v^{l-1})$, $(1 \leq l \leq k)$.

Formally, an EFSM $M = (S_M, T_M)$ is *consistent* with the observation tables $(S, R, E, C)$ and $(S, R, E, D)$ if for each sequence $s$ in $S \cup R$ of length $k$, $x_{i_{k+1}} \in X$, and $y_{i_{k+1}} \in Y$ such that $(d_{i_{k+1}}(p(x_{i_{k+1}})), y_{i_{k+1}}) \in C(s, x_{i_{k+1}})$, there exists a sequence of transitions $(s_0, x_{i_1}, G_{i_1}, op_{i_1}, y_{i_1}, up, s_{i_1})$ $(s_{i_1}, x_{i_2}, G_{i_2}, op_{i_2}, y_{i_2}, up, s_{i_2}) \cdots (s_{i_{k-1}}, x_{i_k}, G_{i_k}, op_{i_k}, y_{i_k}, up, s_{i_k})$ $(s_{i_k}, x_{i_{k+1}}, G_{i_{k+1}}, op_{i_{k+1}}, y_{i_{k+1}}, up, s_{i_{k+1}})$ in $M$, such that for each $1 \leq l \leq k+1$:

- $G_{i_l}(d_{i_l}(p(x_{i_l})), v^{l-1}) = True$. Informally, this means the guard of each transition is satisfied.
- $(d_{i_l}(p(x_{i_l})), y_{i_l}) \in C(s^{l-1}, x_{i_l})$. Informally, this means that the output symbol of each transition is the same as what is recorded in the control table.
- $(d_{i_l}(p(x_{i_l})), v^{l-1} \rightarrow op_{i_l}(d_{i_l}(p(x_{i_l})), v^{l-1})) \in D(s^{l-1}, x_{i_l})$. Informally, this means the output parameter value of each transition is the same as what is recorded in the data table.

Listing 1: Outline of the initial inference algorithm

```
1  procedure INFERENCE
2      Initialize the Control Table and Data Table with the sets S = ε, E = X,
3          and R obtained by associating each input symbol with only one param-
4          eter value if there are parameters associated with the input symbol;
5      Construct parameterized input sequences and obtain corresponding param-
6          eterized output sequences by testing the SUI;
7      Record the observations in the Control Table and Data Table;
8      if ndv is identified then
9          Construct and apply additional input sequences accordingly;
10     end if
11     while the Control Table is not balanced, dispute-free or closed do
12         Make the table balanced, i.e., ∀s, t ∈ S ∪ R, s and t are balanced;
13         if ndv has been identified then
14             Make the table balanced considering the ndv;
15         end if
16         Make the table dispute-free, i.e., all disputed row s ∈ S are resolved;
17         Make the table closed, i.e., ∀t ∈ R, ∃s ∈ S such that s ≅ t;
18     end while
19     Construct the conjecture from the Control Table and Data Table.
20 end procedure
```

**Theorem 1.** *The conjecture EFSM $Q = (S_Q, T_Q)$ constructed from balanced, closed, dispute-free observation table $C$ and $D$ is consistent with the observation tables.*

*Proof.* Because $S \cup R$ is prefix closed, $s^{l-1} \in S \cup R$, $(1 \leq l \leq k)$. From the inference algorithm, we can see that all the strings in $R$ are obtained by extending a string in $S$ with one parameterized input symbol. Thus, $s^{l-1} \in S$, $(1 \leq l \leq k)$. We can also see that all the strings in $S$ are not equivalent to each other.

According to the way observations are recorded in observation tables, supposed when $s$ is applied to the EFSM being inferred, the parameterized output sequence is $y_{i_1}(d'_{i_1}(p(y_{i_1})))\ y_{i_2}(d'_{i_2}(p(y_{i_2})))\cdots\ y_{i_k}(d'_{i_k}(p(y_{i_k})))$, we have $(d_{i_l}(p(x_{i_l})), y_{i_l}) \in C(s^{l-1}, x_{i_l})$, and $(d_{i_l}(p(x_{i_l})),\ v^{l-1} \rightarrow d'_{i_l}(p(y_{i_l}))) \in D(s^{l-1}, x_{i_l})$ $(1 \leq l \leq k)$.

Thus, for each $s^{l-1}$ $(1 \leq l \leq k+1)$, there exists a transition in $Q$ from $[s^{l-1}]$ with input symbol $x_{i_l}$ and output symbol $y_{i_l}$ in the form of $([s^{l-1}], x_{i_l}, G_{i_l}, op_{i_l}, y_{i_l}, up, s'_{i_l})$.

- Since $(d_{i_l}(p(x_{i_l})), v^{l-1} \rightarrow d'_{i_l}(p(y_{i_l}))) \in D(s^{l-1}, x_{i_l})$ corresponding to $y_{i_l}$,

$(d_{i_l}(p(x_{i_l})), v^{l-1} \rightarrow True)$ is an element of $G_{i_l}$.
That means $G_{i_l}(d_{i_l}(p(x_{i_l})), v^{l-1}) = True$.

- $(d_{i_l}(p(x_{i_l})), y_{i_l}) \in C(s^{l-1}, x_{i_l})$.
- Since $(d_{i_l}(p(x_{i_l})), v^{l-1} \rightarrow d'_{i_l}(p(y_{i_l}))) \in D(s^{l-1}, x_{i_l})$ corresponding to $y_{i_l}$, $(d_{i_l}(p(x_{i_l})), v^{l-1} \rightarrow d'_{i_l}(p(y_{i_l})))$ is en element of $op_{i_l}$. Thus, $(d_{i_l}(p(x_{i_l})), v^{l-1} \rightarrow op(d_{i_l}\ (p(x_{i_l})), v^{l-1})) = (d_{i_l}(p(x_{i_l})), v^{l-1} \rightarrow d'_{i_l}(p(y_{i_l})))$ which belongs to $D(s^{l-1}, x_{i_l})$.
- The target state $s'_{i_l} = [s^{l-1} \cdot x_{i_l}(d_{i_l}(p(x_{i_l})))] = [s^l]$.

So, these transitions in $Q$ is a "chain" as described in definition of consistency and satisfy all the three conditions. This means $Q$ is consistent with the observation tables. □

Note that considering $ndv$ as a constant, the definition of consistency and the proof also apply to the case where $ndv$ is involved.

An EFSM $(S, T)$ is said to be *strongly enabled* if for each state $s \in S$, each variable valuation $v \in D_V$, and each transition $(s, x, G, op, y, up, s') \in T$, there exists an input parameter valuation $d \in D_x$ such as $G(d, v) = True$. For example, suppose there are two transitions starting from a state $s$, the two guards are $p_1 = v_1 \wedge p_2 = v_2$ and $p_1 \neq v_1 \vee p_1 \neq v_2$, and there is a possible valuation $v^*$ where $v^*(v_1) \neq v^*(v_2)$, such an EFSM is not strongly enabled, because there does not exist a valuation $d \in D_x$ satisfying $p_1 = v_1 \wedge p_2 = v_2$.

With the following theorem, we can see that the conjecture obtained is "minimal" under certain assumptions.

**Theorem 2.** *Suppose a strongly enabled SUI satisfies the direct comparison assumption, and the all data strategy is applied in the inference procedure, if there is another EFSM* $W = (S_W, T_W)$ *consistent with the observation tables,* $|S_W| \geq |S_Q|$.

*Proof.* Suppose there is an EFSM $W = (S_W, T_W)$ consistent with the observation tables, and $|S_W| < |S_Q|$.

Since $|S_Q| = |S|$, there exist strings $s_1, s_2 \in S$, such that in $W$ the same state $s^*$ is reached by applying $s_1$ or $s_2$ from the initial state.

Since in the observation tables, $s_1$ is not equivalent with $s_2$, there exist $x \in X$, $y \in Y$ such that there exists $(d_1(p(x)), y) \in C(s_1, x)$ but there is no such an element with output symbol $y$ in $C(s_2, x)$.

Since $W$ is consistent with the observation tables, in $W$ there is a transition $(s^*, x, G, op, y, up, s^{*\prime}) \in T_W$ from state $s^*$. Suppose the valuation of $V$ after applying $s_1$ to $W$ is $v_1$, $G(d_1(p(x)), v_1) = True$.

Suppose the valuation of $V$ after applying $s_2$ to $W$ is $v_2$, which is also the valuation after applying $s_2$ to SUI. In the state reached by $s_2$ in SUI, there must exist a valuation $d_2 \in D_x$ such that $G(d_2(p(x)), v_2) = True$. Otherwise we can transform the transitions starting from the state reached by $s_2$ in SUI, such that there is one transition having $G$ as the guard, and this transition is never enabled with variable valuation $v_2$. This means the SUI is not strong enabled.

Because the direct comparison assumption is satisfied in SUI, and the all data strategy is applied, during the inference procedure, we must have tried a valuation $d^* \in D_x$ after $s_2$ such that $G(d^*(p(x), v_2) = True$, and the output symbol is not $y$. This is a contradiction with the statement that in $W$ there is a transition $(s^*, x, G, op, y, up, s^{*\prime}) \in T_W$ from state $s^*$.

So, such an EFSM $W$ does not exist. This concludes the proof. □

Since the SUI itself is consistent with the observation tables, from Theorem 2, we can know that the number of states of the conjecture is bounded by the number of states of the SUI.

Suppose the number of states of SUI is $m$, the number of input symbols is $|X| = k$. From Theorem 2, we know that the number of rows in the set $S$ is bounded by $m$. According to the way $R$ is constructed, the number of rows in $R$ is bounded by $mk$, which means for each string in $S$, there are at most $k$ corresponding strings in $R$. Thus, the number of cells in the final control table is bounded by $(m + mk)k$.

The number of tests needed for each cell in the observation tables depends on the test data selection strategy. When the all data strategy is used, for each $s \in S \cup R$, $x \in X$, in the worst case, there could be $|V| = n$ different values in the current valuation of $V$, that means, for each parameter of $x$, there are $n + 1$ different values to be executed. If we want to execute all the combinations of different values of parameters of $x$, the number of tests needed will not be bounded by a polynomial expression. Thus, it is important to define an efficient test data selection strategy, or in another word, to find a tradeoff between the cost and the accuracy of the conjecture obtained.

### 2.2.8 New Observation Processing

In Angluin's algorithm, after a conjecture is obtained, it is given to a *teacher*, who will provide a counterexample if the conjecture is not equivalent to SUI. In our algorithm, we extend the concept of "counterexample" to that of "new observation", which is a pair of a parameterized input sequence and the corresponding output sequence of SUI. The input sequence has not been executed in the previous model inference procedure. In Angluin's algorithm,

counterexample processing always leads to a new conjecture with more states. In our case, since the guards and output parameter functions in the raw conjecture are always partial functions, new observation processing may only "enrich" these function without introducing new states in the next conjecture.

For the time being, we do not consider how a new observation is obtained.

Suppose the input symbol sequence of the new observation is $\alpha \in X^*$, the parameterized input sequence is $\alpha(d(p(\alpha)))$, the output symbol sequence is sequence is $\beta \in Y^*$, and the parameterized output sequence is $\beta(d(p(\beta)))$. We denote $\mathit{suffix}^j(\alpha)$ as the suffix of $\alpha$ of length $j$.

By adapting the "Suffix1by1" method described in [20], our new observation processing approach is outlined in Listing 2.

Listing 2: Processing new observation

```
1  procedure NEW_OBSERVATION_PROCESSING
2      Divide α as γ · σ, where γ is the longest prefix of α such that
3          there exists d'(p(γ)) ∈ D_γ and γ(d'(p(γ))) ∈ S ∪ R;
4      if d ≠ d' then
5          if γ(d'(p(γ))) ∈ S then
6              Add γ(d(p(γ))) into S;
7          else
8              Add γ(d(p(γ))) into R;
9          end if
10     end if
11     for j = 1 to |σ| do
12         if suffix^j(σ) ∉ E then
13             Add suffix^j(σ) into E;
14         end if
15         Fill the empty cells of the observation tables;
16         Make the Control Table balanced;
17         if the Control Table is not closed then
18             break the for loop;
19         end if
20     end for
21     Make the Control Table balanced, closed, and dispute-free;
22     Make a conjecture.
23 end procedure
```

In the running example, suppose after the conjecture depicted in Figure 2 is obtained, a new observation is obtained, where the input sequence is $a(5)a(6)b(2)$ and the output sequence is $d(5)c(1000)d(0)$.

Following the new observation processing algorithm, $\gamma = a \cdot a$, $\sigma = b$. We add $a(5)a(6)$ into $R$. Since $b \in E$, no column is added. After executing several tests, the balanced, closed, and dispute-free Control Table obtained is 9, while the corresponding Data Table is 10.

| | | $E$ | | $\cong$ |
|---|---|---|---|---|
| | | $a$ | $b$ | |
| $S$ | $\varepsilon$ | $(5,d),(6,d)$ $(ndv_3,d)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | |
| | $a(5)$ | $(5,c),(6,c)$ $(ndv_3,c)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | |
| | $b(2)$ | $(5,\Omega),(6,\Omega)$ $(ndv_3,\Omega)$ | $(2,e),(3,d)$ $(ndv_3,e)$ | |
| $R$ | $a(5)a(5)$ | $(5,\Omega),(6,\Omega)$ $(ndv_3,\Omega)$ | $(2,d),(3,d)$ $(ndv_3,e)$ | $b(2)$ |
| | $\mathbf{a(5)a(6)}$ | $\mathbf{(5,\Omega),(6,\Omega)}$ $\mathbf{(ndv_3,\Omega)}$ | $\mathbf{(2,d),(3,d)}$ $\mathbf{(ndv_3,e)}$ | $b(2)$ |
| | $a(5)b(2)$ | $(5,c),(6,c)$ $(ndv_3,c)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $a(5)$ |
| | $b(2)a(5)$ | $(5,\Omega),(6,\Omega)$ $(ndv_3,\Omega)$ | $(2,e),(3,d)$ $(ndv_3,e)$ | $b(2)$ |
| | $b(2)b(2)$ | $(5,d),(6,d)$ $(ndv_3,d)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $\varepsilon$ |
| | $b(2)b(3)$ | $(5,c),(6,c)$ $(ndv_3,c)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $a(5)$ |

Table 9: Control Table after Processing a New Observation

The raw conjecture constructed from Table 9 and Table 10 is depicted in Figure 5. The new data obtained from the processing is in bold font. We can see that by processing this new observation, the data of the conjecture is enriched, but no new state is introduced.

Now, consider we have another new observation, in which the input sequence is $a(5)a(6)b(2)a(5)$ and the output sequence is $d(5)c(222)d(0)d(5)$. Following the new observation processing algorithm, $\gamma = a \cdot a$, $\sigma = b \cdot a$. Since $a(5)a(6) \in R$, no row is added. In the **for** loop in the algorithm, when $j = 1$, no column is added. When $j = 2$, $b \cdot a$ is added to $E$. Then, we construct and execute tests to fill the empty cells in the observation table. Since $ndv$ is included in the inference procedure, policy described in Section 2.2.3 with respect to $ndv$ is followed. We also use more input data to follow the "All Data Strategy" described in Section 2.2.4. In the end of the **for** loop, we obtain the Control Table 11 and Data Table 12. Now, the Control Table is not closed, since the row $a(5)a(6)$ is not equivalent to any row in $S$. We move it to $S$, extend $R$ accordingly, construct and execute more tests, until we obtain balanced, closed, and dispute-free Control Table. Finally we are able to construct another conjecture, which has 4 states as the SUI depicted in Figure 1.

| | | $E$ | |
|---|---|---|---|
| | | $a$ | $b$ |
| $S$ | $\varepsilon$ | $(5,(0,0,0,0) \to 5)$,<br>$(6,(0,0,0,0) \to 6)$,<br>$(0,(0,0,0,0) \to 0)$ | $(2,(0,0,0,0) \to 2)$,<br>$(3,(0,0,0,0) \to 3)$,<br>$(0,(0,0,0,0) \to 0)$ |
| | $a(5)$ | $(5,(5,0,0,5) \to ndv)$,<br>$(\mathbf{6},(\mathbf{5},\mathbf{0},\mathbf{0},\mathbf{5}) \to \mathbf{ndv})$,<br>$(0,(5,0,0,5) \to 120)$ | $(2,(5,0,0,5) \to 0)$,<br>$(3,(5,0,0,5) \to 0)$,<br>$(0,(5,0,0,5) \to 0)$ |
| | $b(2)$ | $(5,(0,2,0,2) \to \omega)$,<br>$(6,(0,2,0,2) \to \omega)$,<br>$(0,(0,2,0,2) \to \omega)$ | $(2,(0,2,0,2) \to \omega)$,<br>$(3,(0,2,0,2) \to 0)$,<br>$(0,(0,2,0,2) \to \omega)$ |
| $R$ | $a(5)a(5)$ | $(5,(5,0,300,5) \to \omega)$,<br>$(6,(5,0,500,5) \to \omega)$,<br>$(400,(5,0,400,5) \to \omega)$ | $(2,(5,0,600,5) \to 0)$,<br>$(3,(5,0,700,5) \to 0)$,<br>$(800,(5,0,800,5) \to \omega)$ |
| | $\mathbf{a(5)a(6)}$ | $(\mathbf{5},(\mathbf{5},\mathbf{0},\mathbf{2000},\mathbf{5}) \to \omega)$,<br>$(\mathbf{6},(\mathbf{5},\mathbf{0},\mathbf{3000},\mathbf{5}) \to \omega)$,<br>$(\mathbf{4000},(\mathbf{5},\mathbf{0},\mathbf{4000},\mathbf{5}) \to \omega)$ | $(\mathbf{2},(\mathbf{5},\mathbf{0},\mathbf{1000},\mathbf{5}) \to \mathbf{0})$,<br>$(\mathbf{3},(\mathbf{5},\mathbf{0},\mathbf{5000},\mathbf{5}) \to \mathbf{0})$,<br>$(\mathbf{6000},(\mathbf{5},\mathbf{0},\mathbf{6000},\mathbf{5}) \to \omega)$ |
| | $a(5)b(2)$ | $(5,(5,2,0,0) \to 900)$,<br>$(6,(5,2,0,0) \to 110)$,<br>$(0,(5,2,0,0) \to 130)$ | $(2,(5,2,0,0) \to 0)$,<br>$(3,(5,2,0,0) \to 0)$,<br>$(0,(5,2,0,0) \to 0)$ |
| | $b(2)a(5)$ | $(5,(5,2,0,2) \to \omega)$,<br>$(6,(5,2,0,2) \to \omega)$,<br>$(0,(5,2,0,2) \to \omega)$ | $(2,(5,2,0,2) \to \omega)$,<br>$(3,(5,2,0,2) \to 0)$,<br>$(0,(5,2,0,2) \to \omega)$ |
| | $b(2)b(2)$ | $(5,(0,2,0,2) \to 5)$,<br>$(6,(0,2,0,2) \to 6)$,<br>$(0,(0,2,0,2) \to 0)$ | $(2,(0,2,0,2) \to 2)$,<br>$(3,(0,2,0,2) \to 3)$,<br>$(0,(0,2,0,2) \to 0)$ |
| | $b(2)b(3)$ | $(5,(0,3,0,0) \to 140)$,<br>$(6,(0,3,0,0) \to 150)$,<br>$(0,(0,3,0,0) \to 160)$ | $(2,(0,3,0,0) \to 0)$,<br>$(3,(0,3,0,0) \to 0)$,<br>$(0,(0,3,0,0) \to 0)$ |

Table 10: Data Table after Processing a New Observation

## 2.3 Implementation and Experimentation

The model inference algorithm proposed is implemented in Java. Experiments are performed taking different SUIs including: a corrected version of Needham Schroeder Public Key protocol (NSPK) [23], the running example depicted in Figure 1, a simulated service provider of SAML Single Sign On (SSO) [13], and a real service provider of SimpleSAMLphp, which can be found at the address `http://simplesamlphp.org/`.

### 2.3.1 Implementation

The implementation is composed of the following components:

- Inference algorithm.
- Test drivers. As in other testing platforms, the test driver is an important part of the tool to bridge the gap between abstract symbols, and messages and events at the implementation level. One of the test drivers we implemented is used for simulated EFSM SUI, which receives an input sequence, simulates the behaviors of the EFSM model,

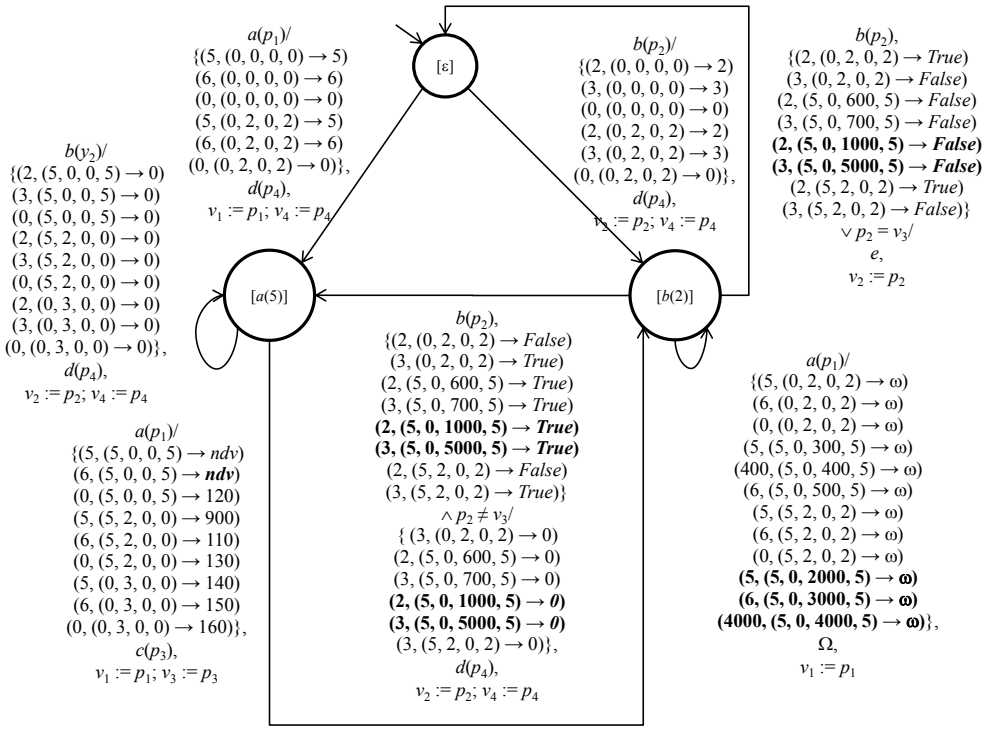


Figure 5: Raw Conjecture after Processing a New Observation

and gives back the output sequence. This is useful for comparing the conjecture with a real model. Another one is used to communicate with SimpleSAMLphp's implementation of service provider, which generates and receives HTTP requests and responses used in SAML SSO protocols.

- Connector to Weka. At the end of the inference, we use Weka to obtain a succinct form of guard and functions. The connector can generate input file in Weka format (ARFF), call filtering and classifying functions, and then the output are parsed and made available to the algorithm.

- Output (connector to GraphViz). For graphic representation of the conjecture, we use the DOT format accepted by GraphViz tool. Information about guard and functions is presented in the label of the transition.

- Logging. The logging component records the requests performed and the intermediate observation tables. Several statistics are also logged

| | | $E$ | | |
|---|---|---|---|---|
| | | $a$ | $b$ | **ba** |
| $S$ | $\varepsilon$ | $(5,d),(6,d)$ $(ndv_3,d)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $(\mathbf{2-5,d-\Omega}),(\mathbf{7-5,d-\Omega})$ $(\mathbf{ndv_3-5,d-\Omega})$ |
| | $a(5)$ | $(5,c),(6,c)$ $(ndv_3,c)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $(\mathbf{2-5,d-c}),(\mathbf{7-5,d-c})$ $(\mathbf{ndv_3-5,d-c})$ |
| | $b(2)$ | $(5,\Omega),(6,\Omega)$ $(ndv_3,\Omega)$ | $(2,e),(3,d)$ $(ndv_3,e)$ | $(\mathbf{2-5,e-d}),(\mathbf{7-5,d-c})$ $(\mathbf{ndv_3-5,e-d})$ |
| $R$ | $a(5)a(5)$ | $(5,\Omega),(6,\Omega)$ $(ndv_3,\Omega)$ | $(2,d),(3,d)$ $(ndv_3,e)$ | $(\mathbf{2-5,d-c}),(\mathbf{7-5,d-c})$ $(\mathbf{ndv_3-5,e-d})$ |
| | $a(5)a(6)$ | $(5,\Omega),(6,\Omega)$ $(ndv_3,\Omega)$ | $(2,d),(3,d)$ $(ndv_3,e)$ | $(\mathbf{2-5,d-d}),(\mathbf{7-5,d-d})$ $(\mathbf{ndv_3-5,e-d})$ |
| | $a(5)b(2)$ | $(5,c),(6,c)$ $(ndv_3,c)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $(\mathbf{2-5,d-c}),(\mathbf{7-5,d-c})$ $(\mathbf{ndv_3-5,d-c})$ |
| | $b(2)a(5)$ | $(5,\Omega),(6,\Omega)$ $(ndv_3,\Omega)$ | $(2,e),(3,d)$ $(ndv_3,e)$ | $(\mathbf{2-5,e-d}),(\mathbf{7-5,d-c})$ $(\mathbf{ndv_3-5,e-d})$ |
| | $b(2)b(2)$ | $(5,d),(6,d)$ $(ndv_3,d)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $(\mathbf{2-5,d-\Omega}),(\mathbf{7-5,d-\Omega})$ $(\mathbf{ndv_3-5,d-\Omega})$ |
| | $b(2)b(3)$ | $(5,c),(6,c)$ $(ndv_3,c)$ | $(2,d),(3,d)$ $(ndv_3,d)$ | $(\mathbf{2-5,d-c}),(\mathbf{7-5,d-c})$ $(\mathbf{ndv_3-5,d-c})$ |

Table 11: Control Table after Processing the Second New Observation

like the number of requests and duration of the learning process.

### 2.3.2 Experiments with SimpleSAMLphp

SAML SSO enables clients to access multiple services by signing-on only once. This is achieved using SAML (Security Assertion Markup Language) which is an XML-based standard for exchanging authentication and authorization data. By applying our algorithm to SAML SSO, we check that it can handle features of state of the art security protocols, so that the models could be used by model checkers and for model based test generation.

Three roles take part in the protocol: a client (C), an identity provider (IdP) and a service provider (SP). C, typically a web browser guided by a user, aims at getting access to a service or a resource provided by SP (i.e., SP-initiated SSO). IdP authenticates C and issues corresponding authentication assertions. Finally, SP uses the assertions generated by IdP to give C access to the requested service. Initially, C asks SP to provide the resource located at certain address. SP then initiates the SAML Authentication Protocol by sending C an Authentication Request redirected to IdP containing an ID which is a string uniquely identifying the request. IdP then challenges C to provide valid credentials and, if the authentication succeeds, IdP builds an Authentication Assertion. IdP places the Authentication Assertion into a response message and then makes C (usually through scripting) sending this message to SP. SP checks the Authentication Assertion and grants access to C. We can summarize the protocol flow in Figure 6.

From the model inference point of view, the ID generated by SP is an

| | | $E$ | | |
|---|---|---|---|---|
| | | $a$ | $b$ | $ba$ |
| $S$ | $\varepsilon$ | $(5,(0,0,0,0) \to 5)$,<br>$(6,(0,0,0,0) \to 6)$,<br>$(0,(0,0,0,0) \to 0)$ | $(2,(0,0,0,0) \to 2)$,<br>$(3,(0,0,0,0) \to 3)$,<br>$(0,(0,0,0,0) \to 0)$ | $(\mathbf{2-5},(\mathbf{0,0,0,0}) \to \mathbf{2}-\omega)$,<br>$(\mathbf{7-5},(\mathbf{0,0,0,0}) \to \mathbf{0}-\omega)$,<br>$(\mathbf{0-5},(\mathbf{0,0,0,0}) \to \mathbf{0}-\omega)$ |
| | $a(5)$ | $(5,(5,0,0,5) \to ndv)$,<br>$(6,(5,0,0,5) \to ndv)$,<br>$(0,(5,0,0,5) \to 120)$ | $(2,(5,0,0,5) \to 0)$,<br>$(3,(5,0,0,5) \to 0)$,<br>$(0,(5,0,0,5) \to 0)$ | $(\mathbf{2-5},(\mathbf{5,0,0,5}) \to \mathbf{0-900})$,<br>$(\mathbf{7-5},(\mathbf{5,0,0,5}) \to \mathbf{0-350})$,<br>$(\mathbf{0-5},(\mathbf{5,0,0,5}) \to \mathbf{0-888})$, |
| | $b(2)$ | $(5,(0,2,0,2) \to \omega)$,<br>$(6,(0,2,0,2) \to \omega)$,<br>$(0,(0,2,0,2) \to \omega)$ | $(2,(0,2,0,2) \to \omega)$,<br>$(3,(0,2,0,2) \to 0)$,<br>$(0,(0,2,0,2) \to \omega)$ | $(\mathbf{2-5},(\mathbf{0,2,0,2}) \to \omega-\mathbf{5})$,<br>$(\mathbf{7-5},(\mathbf{0,2,0,2}) \to \mathbf{0-330})$,<br>$(\mathbf{0-5},(\mathbf{0,2,0,2}) \to \omega-\mathbf{5})$ |
| $R$ | $a(5)a(5)$ | $(5,(5,0,300,5) \to \omega)$,<br>$(6,(5,0,500,5) \to \omega)$,<br>$(400,(5,0,400,5) \to \omega)$ | $(2,(5,0,600,5) \to 0)$,<br>$(3,(5,0,700,5) \to 0)$,<br>$(800,(5,0,800,5) \to \omega)$ | $(\mathbf{2-5},(\mathbf{5,0,333,5}) \to \mathbf{0-444})$,<br>$(\mathbf{7-5},(\mathbf{5,0,340,5}) \to \mathbf{0-340})$,<br>$(\mathbf{777-5},(\mathbf{5,0,777,5}) \to \omega-\mathbf{5})$ |
| | $a(5)a(6)$ | $(5,(5,0,2000,5) \to \omega)$,<br>$(6,(5,0,3000,5) \to \omega)$,<br>$(4000,(5,0,4000,5) \to \omega)$ | $(2,(5,0,1000,5) \to 0)$,<br>$(3,(5,0,5000,5) \to 0)$,<br>$(6000,(5,0,6000,5) \to \omega)$ | $(\mathbf{2-5},(\mathbf{5,0,222,5}) \to \mathbf{0-5})$,<br>$(\mathbf{7-5},(\mathbf{5,0,350,5}) \to \mathbf{0-5})$,<br>$(\mathbf{999-5},(\mathbf{5,0,999,5}) \to \omega-\mathbf{5})$ |
| | $a(5)b(2)$ | $(5,(5,2,0,0) \to 900)$,<br>$(6,(5,2,0,0) \to 110)$,<br>$(0,(5,2,0,0) \to 130)$ | $(2,(5,2,0,0) \to 0)$,<br>$(3,(5,2,0,0) \to 0)$,<br>$(0,(5,2,0,0) \to 0)$ | $(\mathbf{2-5},(\mathbf{5,2,0,0}) \to \mathbf{0-555})$,<br>$(\mathbf{7-5},(\mathbf{5,2,0,0}) \to \mathbf{0-360})$,<br>$(\mathbf{0-5},(\mathbf{5,2,0,0}) \to \mathbf{0-310})$ |
| | $b(2)a(5)$ | $(5,(5,2,0,2) \to \omega)$,<br>$(6,(5,2,0,2) \to \omega)$,<br>$(0,(5,2,0,2) \to \omega)$ | $(2,(5,2,0,2) \to \omega)$,<br>$(3,(5,2,0,2) \to 0)$,<br>$(0,(5,2,0,2) \to \omega)$ | $(\mathbf{2-5},(\mathbf{5,2,0,2}) \to \omega-\mathbf{5})$,<br>$(\mathbf{7-5},(\mathbf{5,2,0,2}) \to \mathbf{0-370})$,<br>$(\mathbf{0-5},(\mathbf{5,2,0,2}) \to \omega-\mathbf{5})$ |
| | $b(2)b(2)$ | $(5,(0,2,0,2) \to 5)$,<br>$(6,(0,2,0,2) \to 6)$,<br>$(0,(0,2,0,2) \to 0)$ | $(2,(0,2,0,2) \to 2)$,<br>$(3,(0,2,0,2) \to 3)$,<br>$(0,(0,2,0,2) \to 0)$ | $(\mathbf{2-5},(\mathbf{0,2,0,2}) \to \mathbf{2}-\omega)$,<br>$(\mathbf{7-5},(\mathbf{0,2,0,2}) \to \mathbf{7}-\omega)$,<br>$(\mathbf{0-5},(\mathbf{0,2,0,2}) \to \mathbf{0}-\omega)$ |
| | $b(2)b(3)$ | $(5,(0,3,0,0) \to 140)$,<br>$(6,(0,3,0,0) \to 150)$,<br>$(0,(0,3,0,0) \to 160)$ | $(2,(0,3,0,0) \to 0)$,<br>$(3,(0,3,0,0) \to 0)$,<br>$(0,(0,3,0,0) \to 0)$ | $(\mathbf{2-5},(\mathbf{0,3,0,0}) \to \mathbf{0-666})$,<br>$(\mathbf{7-5},(\mathbf{0,3,0,0}) \to \mathbf{0-380})$,<br>$(\mathbf{0-5},(\mathbf{0,3,0,0}) \to \mathbf{0-320})$ |

Table 12: Data Table after Processing the Second New Observation

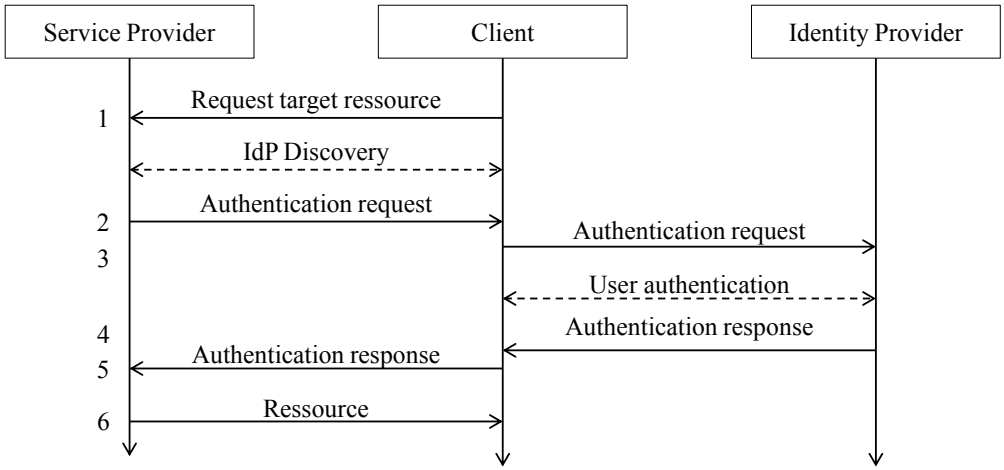


Figure 6: SAML SSO Protocol

*ndv*. In the protocol, this *ndv* is generated by SP and needs to be checked in incoming Authentication Assertions.

SimpleSAMLphp is an application written in native PHP that deals

with authentication. It contains both the service provider and the identity provider. This application is installed on a virtual machine and each service is on a different port. In the following, we illustrate the experimentation with SP.

First, we define the relevant input and output symbols. Depending on the level of abstraction, input symbols can be: *httprequest*, *httpresponse* or, at a higher level: *login*, *getInfo*, *logout*. In our case, we have chosen *In_GetResource*, *In_GetSAMLReq*, and *In_SAMLResp* as input symbols and *Out_IdPlist*, *Out_SAMLReq*, *Out_Resource*, and *Out_Error* as output symbols. In practice, we can follow the requests stream and give a name to each request used in the protocol.

For each request, we need to find parameters which make the request successful such as nonce, session ID, or user password. During this step, we may need to update the input and output symbols defined beforehand. For example, when the same type of HTTP messages are used in two cases with different number of relevant parameters, two input (or output) symbols need to be defined for these cases. Our test drivers implements those abstractions, and the model inference method is executed to infer a model.

### 2.3.3 Results

As an example, the conjecture of the SP in SimpleSAMLphp is depicted in Figure 7.

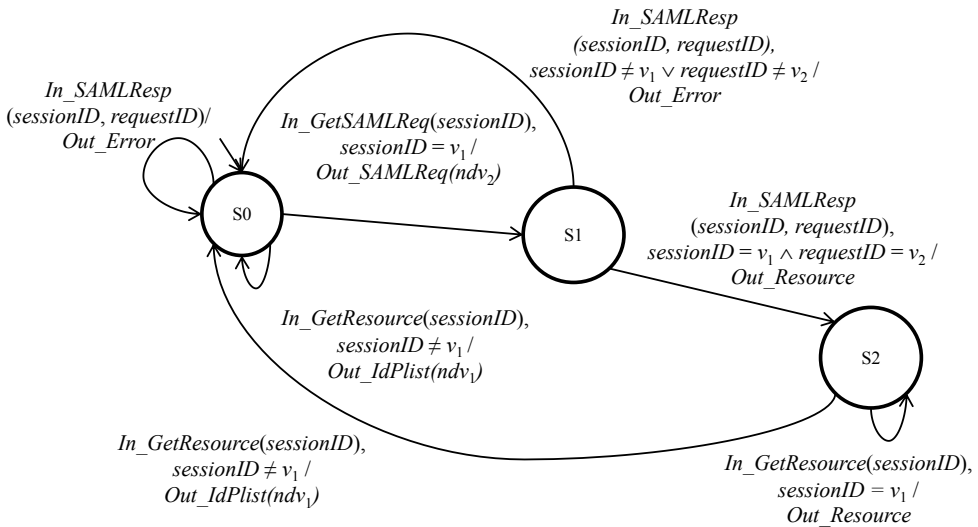


Figure 7: Conjecture of SP in SimpleSAMLphp

| Case | States | Requests | Average Length | Duration |
|---|---|---|---|---|
| NSPK | 2 | 92 | 2.91 | 0.31 |
| Example | 3 | 52 | 2.53 | 0.07 |
| Simulated SAML SP | 3 | 96 | 2.93 | 0.49 |
| SimpleSAMLphp SAML SP | 3 | 212 | 2.93 | 1192.84 |

Table 13: Statistics on Experiments

The experimentation runs quite fast (in less than one second) on locally simulated systems. On web applications, the time for each request can vary between 0 and 2 seconds, and reaches 5 seconds with virtual machine implementations.

In Table 13, the number of requests executed is reported for each experimentation performed. We also report the average length of the parameterized input sequences and the learning duration.

In Table 13, we can see that even for small sized automata the number of requests seems high. This is mainly due to the procedure to make the table balanced, especially when an *ndv* is identified, the value needs to be applied to the parameters of all the future requests. But this problem is reduced with types: an *ndv* is indeed reused only for input parameters of the same type. We thus need less requests to make the table balanced. SAML SSO SP is not really more complex than NSPK in terms of states and transitions but it needs more requests to be inferred. This is mainly because the SP has more input parameters than NSPK and an *ndv* is identified. The real SAML SSO SP needs more requests because in web applications, when we send something wrong, in most cases, we go back to the main page, and we get a session ID for each input symbols. For example, when we try to logout or get some information before login, we get a redirection to the main page and the server gives us a session ID. Thus each symbol produces an *ndv*, which leads to many requests. The time needed to infer the system is also very high compared to a simulated system because we use a virtual machine.

In summary, from the experimentations performed, we can see that the model inference method proposed can produce conjectures for state-of-the-art security protocols, in which multiple input/output parameters and non-deterministic values are involved, using a reasonable number of requests. Although the number of states may look small (for example with respect to the usual inference of deterministic finite automata), this actually corresponds to the type of models used for such protocols, where the complexity is hidden

in the transitions and parameter handling, which our inference method can successfully infer.

## 2.4 Related Work

The need of constructing specifications can occur in various context, and various techniques have been proposed. In [14] finite state models are extracted from Java source code. In [16], an automatic dynamic technique for simultaneously learning and enforcing general temporal properties over method call sequences is presented. In [7] and [22], specifications are inferred from source code for the purpose of reliability and security. In [28], learning is combined with assume-guarantee reasoning for the purpose of verification of autonomous software.

In previous work with other partners, we had shown that inference can be used to derive a restricted form of Parameterized Finite State Machine [21]. The inference algorithms have been applied successfully to several case studies provided by France Telecom [33]. However, such a restricted model (which is memory-less with respect to parameter values) is not suitable in contexts such as security testing where previously exchanged values must be recorded. Inference of variables while testing a black box component is not straightforward, because the internal structure of a component is unknown, the encoding of state information between set of states and variables might be arbitrary, and thus the inference process can hardly infer a meaningful state structure. In addition to that, we also extend previous work by various authors to accommodate non-deterministic values which are important in security protocols.

In [1], Angluin's algorithm [4] was adapted to include data parameters in messages and states for generating Symbolic Mealy Machine (SMM) models, which are similar to EFSM. The framework is inspired by predicate abstraction, which has been successful for extending finite-state model checking to large and infinite state spaces. A key assumption of [1] is that a set of guards and expressions, which includes the guards and expressions used in the symbolic transitions of the SMM, is also given a priori. This set can be seen as describing how state variables of SMM can be used, i.e., how they can influence control flow through tests, and how they can be manipulated to produce output symbols. In the present work, we do not rely on such a strong assumption.

In [8], variables and related operations are inferred for communication protocols. In this approach, first the behavior of the protocol is observed when the parameters of input messages are from a small domain. Then a finite state Mealy machine is generated, which describes the behavior of the

component on this small domain. Thereafter this finite state Mealy machine is folded into a smaller symbolic model. In addition to the constraints in the symbolic model mentioned before, this two-phase approach implies that the intermediate finite state Mealy machine may get rather large, in comparison with the final Symbolic Mealy Machine. Our approach infers EFSM directly and does not suffer from this problem. Furthermore, the model used by [8] only allows equality in guards and output functions. We extend this and allow the use of functions that can be inferred by traditional data inference techniques.

In [27], learning is integrated with verification and testing of cryptographic protocols. The learning ingredient of the approach contains two parts, adding examples while there exists a trace in the implementation but not in the model, and removing counterexamples while there exists a trace in the model but not in the implementation. In adding examples, transitions are added manually into the model with intuition that the control structure of the current specification should be kept as much as possible, while how the "triggering" trace is identified is not clearly defined. Removing a counterexample consists in undoing the effect of the last "adding examples" operation, and in proposing another way of adding the last example.

Among the existing work on security testing (e.g., [2, 42, 45, 31]), many of them suppose that a specification of the system under test (SUT) is available. As stated before, the limitation of these methods is that in reality, a formal specification of SUT is usually unavailable. Even in the case that such a specification is available, implementations are always more complicated than the specification with a lot of engineering details implemented, and these testing methods do not deal with the behaviors beyond what is specified. The model inference methods can help to obtain and enhance the model of SUT and thus improve the quality of testing. At the same time, input vector identification techniques used in penetration testing [18] may be integrated in model inference to identify "interesting" input data in the context of security.

In European project AVANTSSAR (www.avantssar.eu), model checking techniques are developed for security protocols and web applications. For example, [5] provided formal models of a variant of the SAML protocol implemented by Google, and revealed a severe security flaw that allows a dishonest service provider to impersonate a user at another service provider. In SPaCIoS project, these techniques are extended and applied to generate security tests. For example, [11] considers models of protocols and describe an approach to generate tests with mutation operators to introduce potential security-specific leaks into the model. Then, if the leak is confirmed by a model analyzer, a test case for the security property is generated. The EFSM model obtained using the model inference algorithm proposed here has the

same semantics as the models used in these papers. Thus, the algorithm could be applied together with techniques described in [5] and [11] to solve practical security validation problems.

# 3 Quotient Model Inference for FSM

Quotient model inference is a new method to infer finite state behavioral models of black box components by testing them. Typically, such components could be accessed over a network, so that we do not even assume that the executable can be scrutinized: the system can only be observed at its interfaces [1]. This corresponds to a typical black box testing situation where the tester would send inputs to a system and observe its outputs.

We assume that the System Under Test (SUT) can be modeled, at some level of abstraction, on its inputs and outputs, as a Finite State Machine (FSM). FSM-based testing theory has shown that an FSM can be identified, i.e., the SUT can be tested to be proven equivalent to it, with the help of state identifying (distinguishing) sequences, constituting, e.g., a characterization set, W-set, of input sequences.

Central to this approach is the notion of initial quotient of an FSM associated with a "partial" characterization set $Z \in W$. In essence, the quotient represents an approximate model where some states might not be distinguished. Previous methods for automata inference have mostly been derived from grammatical inference techniques. And this method follows partly the same paradigm framework, in particular the Minimally Adequate Teacher [4]. It assumes that the SUT can be used to answer queries in two forms: output queries, where an input sequence can be submitted to the SUT (after resetting it to its initial state) to get the corresponding output sequence; and a restricted form of equivalence queries: we assume that further testing of the SUT can provide counterexamples, i.e., input sequences for which the output sequences differ from the SUT in the inferred FSM. This method will thus infer an SUT by building increasingly precise quotients of it using counterexamples.

This method departs from previous methods in that it is directly inspired by testing theory. It is also designed to be more adapted and efficient in a software testing context: although it is an active testing and learning method, it can start from existing test records as in passive testing and inference; an initial Z-set of distinguishing sequences can be provided, e.g., by the test expert. The experiments confirm that it requires fewer queries (tests) than existing recent algorithms to infer a given FSM.

## 3.1 Basic Definitions

A Finite State Machine (FSM) A is a 5-tuple $(S, s_0, I, O, h_A)$, where:

- $S$ is a finite set of states with the initial state $s_0$;
- $I$ and $O$ are finite non-empty disjoint sets of inputs and outputs, respectively;
- $h_A$ is a behavior function $h_A : S \times I \to 2^{S \times O}$, where $2^{S \times O}$ is the powerset of $S \times O$.

Depending on the properties of the behaviur function, a number of various types of FSM can be defined as follows. FSM $A = (S, s_0, I, O, h_A)$ is:

- *trivial* if $h_A(s_0, a) = \varnothing, \forall (s_0, a) \in S \times I$;
- *complete* if $h_A(s, a) \neq \varnothing, \forall (s, a) \in S \times I$;
- *partially specified* (a partial FSM) if $h_A(s, a) = \varnothing$, for some $(s, a) \in S \times I$;
- *deterministic* if $|h_A(s, a)| \leq 1, \forall (s, a) \in S \times I$;
- *nondeterministic* if $|h_A(s, a)| > 1$, for some $(s, a) \in S \times I$;
- *observable* if the automaton $A_\times = (S, s_0, I \times O, \delta)$, where $\delta(s, ab) \ni s'$ iff $(s', b) \in h_A(s, a)$, is deterministic

We consider only observable machines; one could employ a standard procedure for automata determinization to transform a given FSM into an observable one. Moreover, all the machines are assumed to be initially connected, i.e., each state is reachable from the initial state. We use $a$, $b$, $c$ for input and output symbols, $\alpha$, $\beta$, $\gamma$ for input and output sequences, $s$, $t$, $p$, $q$ for states, and $u$, $v$, $w$ for traces.

Given FSM $A = (S, s_0, I, O, h_A)$, $(s_1, ab, s_2)$ is a transition if $s_1, s_2 \in S$ and $(s_2, b) \in h_A(s_1, a)$. A path from state $s_1$ to $s_{n+1}$ is a sequence of transitions $(s_1, a_1b_1, s_2)(s_2, a_2b_2, s_3) \dots (s_n, a_nb_n, s_{n+1})$ such that $(s_{i+1}, b_i) \in h_A(s_i, a_i)$, where $1 \leq i \leq n$ and $n$ is the length of the path. A sequence $u \in (I \times O)^*$ is called a *trace* of FSM $A$ in state $s_1 \in S$, if there exists a path $(s_1, a_1b_1, s_2)(s_2, a_2b_2, s_3) \dots (s_n, a_nb_n, s_{n+1})$ such that $u = a_1b_1a_2b_2 \dots a_nb_n$. Note that a trace of $A$ in state $s_0$ is a word of the automaton $A_\times$. Let $inp(u) \subseteq I$ denote the set of inputs appearing in a trace $u$. We use $Tr(s)$ to denote the set of all traces of $A$ in state $s$ and $Tr(A)$ to denote the set of traces of $A$ in the initial state.

The *projection operator* $\downarrow_B$, which projects sequences in $(I \times O)^*$ onto the set $B \subseteq I \cup O$, is recursively defined as $\varepsilon \downarrow_B = \varepsilon$, $(ua) \downarrow_B = u \downarrow_B a$ if $a \in B$, and $(ua) \downarrow_B = u \downarrow_B$ otherwise, where $u \in (I \times O)^*$ and $a \in I \cup O$. Given a sequence $u \in (I \times O)^*$, the sequence $u \downarrow_I$ is the input projection of $u$. Input sequence $\alpha \in I^*$ is a defined input sequence in state $s$ of $A$ if there exists $u \in Tr(s)$ such that $\alpha = u \downarrow_I$. We use $\Omega(s)$ to denote the set of all defined input sequences for state $s$.

Given two states $s, t \in S$ of FSM $A$ and a set of input sequences $Z \subseteq \Omega(s) \cap \Omega(t)$, $s$ and $t$ are *Z-equivalent*, if for all $a \in Z$ it holds that $\{u \in Tr(s) \mid u \downarrow_I = a\} = \{u \in Tr(t) \mid u \downarrow_I = a\}$. $Z$-equivalent states are *k-equivalent*, if $Z$ includes all input sequences of length $k$. States $s$ and $t$ are equivalent if they are $Z$-equivalent and $Z = \Omega(s) = \Omega(t)$, i.e., $Tr(s) = Tr(t)$. If $Tr(s) \subseteq Tr(t)$ then $s$ is trace-included in $t$. States $s$ and $t$ that are not $Z$-equivalent are *Z-distinguishable*. An input sequence $a \in \Omega(s) \cap \Omega(t)$ such that $\{u \in Tr(s) \mid u \downarrow_I = a\} \neq \{u \in Tr(t) \mid u \downarrow_I = a\}$ is called a sequence *distinguishing* $s$ and $t$. States $s$ and $t$ are *(k-)distinguishable*, if there exists a sequence distinguishing them (of length $k$). A set of input sequences $Z$ such that each pair of distinguishable states is $Z$-distinguishable is called a *characterization set* of FSM $A$. A complete FSM which has no equivalent states is called *minimal*.

The introduced equivalence and distinguishability relations over states are extended to states of different machines.

## 3.2 Inferring a State Model of a System

In this section, we introduce the concept of an initial *Z-quotient* of a given FSM and give the algorithm for its inference by testing.

### 3.2.1 Initial $Z$-Quotient

Assume we are given an SUT which behaves as an FSM and on which we can perform experiments by applying inputs and observing outputs to infer an FSM model. To this end, we need to know at least a subset of its input alphabet. The inference can easily be performed assuming that the black box behaves as a deterministic machine and the number of its distinct states $n$ as well as a characterization set are known. Recall that characterization set distinguishes all non-equivalent states. To infer an FSM model, it is sufficient to use a slightly modified $W$-method [40]. Namely, instead of using a state cover, as it is unknown, we use the set of all possible input sequences of up to $n-1$ length. However, when the actual number of states and characterization set are unknown, we need to find a way of inferring an approximated model

with a controllable precision. This could be done by using, instead of a characterization set, a predefined set $Z$ of input sequences. The idea comes from the fact that it defines the $Z$-equivalence relation between states of the FSM, so states of this machine could be identified modulo $Z$-equivalence. The idea leads to the following definition and eventually to an inference algorithm.

Given a complete FSM $A = (S, s_0, I, O, h_A)$ and a finite non-empty set of input sequences $Z \in I^*$, let $\pi_Z$ be the partition on the set of states $S$ induced by the $Z$-equivalence relation. For state $s$, the states that are $Z$-equivalent to state $s$ constitute the equivalence class $\pi_Z(s)$. It is known that a state equivalence relation induces a quotient model of the original machine, see, e.g., quotient model of Kripke structure [12]. The idea we use, which can be traced back to work in [10] and [25] is to collapse all $Z$-equivalent states ($k$-equivalent in [10]) while keeping all transitions. The obtained model preserves all the traces of the original machine, but contains additional traces.

Quotient models are widely used in model checking, however, in the inference process we are constrained to a single representative of each equivalence class $\pi_Z$. The reason is that once a distinct $Z$-distinguishable state is identified, it should be included into the inferred model. This constraint leads to the following definition.

**Definition 1.** *Given a complete FSM $A = (S, s_0, X, O, h_A)$ and a set of input sequences $Z \subseteq I^*$, $I \subseteq X$, an FSM $K = (Q, q_0, I, O, h_K)$ is an* initial $(Z, I)$-quotient *of $A$, if there exists an injection $f$ from $Q$ to $S$ such that*

- *$f(q_0) = s_0$;*
- *for any two distinct states $q_1, q_2 \in Q$, $f(q_1)$ and $f(q_2)$ are $Z$-distinguishable;*
- *for any $q \in Q$, there exists a path $(s_0, a_1b_1, s_1) \dots (s_{n-1}, a_nb_n, s_n)$, such that $s_i = f(q_i)$, $q_i \in Q$, $1 \leq i \leq n$ and $s_n = f(q)$;*
- *for any $q \in Q$ and $a \in I$, $b \in O$, $(p, b) \in h_K(q, a)$ iff there exists $s \in S$, such that $(s, b) \in h_A(f(q), a)$ and $s$ and $f(p)$ are $Z$-equivalent.*

We use *initial* in the introduced term to emphasize the fact that the latter represents a part reachable from the initial state of a given FSM modulo $Z$-equivalence. We do not require that a quotient use each and every input of the FSM to be inferred. This choice is motivated by the observation that in practical situations, the number of inputs can be just too big to explore all of them, moreover, some inputs can be *equivalent*, in the sense that they cause

transitions that differ only in input symbols. If $I = X$ then instead of initial $(Z, X)$-quotient, we will refer to *initial Z-quotient* or simply to *quotient* when the set $Z$ is clear from the context. As an example, consider the FSM $A$ in Figure 8 and its initial $Z$-quotient in Figure 9, where $Z = \{a, b\}$, $f(\varepsilon) = 0$, $f(a_1) = 1$ and $f(a_1a_2) = 2$. Note that the initial $Z$-quotient is deterministic.

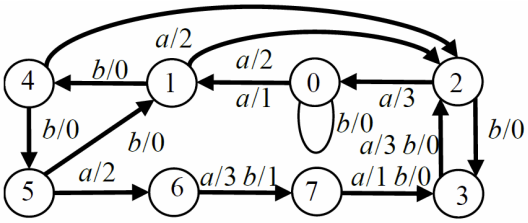


Figure 8: FSM $A$

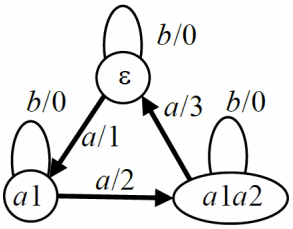


Figure 9: Initial $\{a, b\}$-quotient of FSM $A$

There is a special case of initial $Z$-quotient, when the set $Z$ is a characterization set of a given FSM $A$. In this case, since any two states of $A$ are distinguished by a sequence in $Z$, each state of the given machine is represented by a distinct state in the initial quotient, which thus becomes equivalent to $A$. Recall that in the case of Figure 8, where $Z = \{a, b\}$ is not a characterization set of $A$, the initial $Z$-quotient and $A$ are not equivalent. This observation leads to the following theorem about properties of initial $Z$-quotient.

**Theorem 3.** *Given an initial Z-quotient $K$ of a complete FSM $A$, if $Z$ is a characterization set of $A$, then FSM $A$ and $K$ are equivalent; otherwise, if $A$ has distinguishable but* Z-equivalent *states, then $A$ and $K$ are distinguishable.*

The more sequences of a characterization set are included into the set $Z$ the more precise is the approximation. The precision of initial $Z$-quotient of

an FSM can be controlled by the parameter $Z$, henceforth called the inference parameter.

Given a natural $k$, an initial $Z$-quotient is called a *$k$-quotient* if $Z = I^k$ [17]. It is known that the set $I^k$ contains distinguishing sequences for any pair of states in any FSM over the input alphabet $I$ with at most $n = k + 1$ states, hence it is a characterization set of such machines.

The case of $k = n-1$ actually corresponds to worst case situations, which occur in special pathological machines, such as Moore locks. For other types of machines, $k$-equivalence of states becomes state equivalence for a much lower value of $k$ and thus it may be more appropriate to consider instead of upper bounds asymptotic characterization of FSM parameters for "almost all FSMs". The monograph [39] indicates that for complete FSM with $n$ states, $m$ inputs, and $l$ outputs, the length of input sequences reaching all $n$ states is asymptotically equal to $\log_m(n)$ and distinguishing states just $\log_m \log_l(n)$. These results suggest that even when the value of the parameter $k$ is well below the actual number of states of a given FSM, the approximation of the FSM in the form of a $k$-quotient might be sufficiently precise and thus acceptable for practical applications. This also means that choosing a set $Z$, one is not obliged to focus on long input sequences, but rather on those which may discriminate various operational modes of an SUT and thus its internal states.

### 3.2.2 Inferring $Z$-Quotient of the SUT

We assume that a given SUT treated as a black box can be modeled by a complete and deterministic FSM over the input $X$ and output $O$ sets. Moreover, a reset operation can be performed on the SUT, when different test sequences are to be applied to its initial state.

We further assume that we are given a set of inputs $I \subseteq X$ and set of input sequences $Z \subseteq I^*$ to infer a $(Z, I)$-quotient of $A$ by testing its implementation, the SUT. A basic idea of our inference method directly following from the clauses of the definition of $(Z, I)$-quotient is as follows. We start building an initial $(Z, I)$-quotient by including an initial state which could be injected to the initial state of A. We explore the states of $A$ from the initial state by applying inputs from the given set $I$. For each visited state, we decide whether to include a corresponding state in the initial $(Z, I)$-quotient. If the current state is $Z$-equivalent to a state already visited, then we do not explore states of $A$ further from this state. For each state, the transitions are defined respecting the $Z$-equivalence.

This basic idea is described in more detail in the following algorithm. To represent the observed traces, we use a tree FSM.

**Definition 2.** *Given a (prefix closed*[1]*) set $U$ of observed traces of an FSM over input set $I$ and output set $O$, the* observation tree *is FSM $(U, \varepsilon, I, O, h_U)$, where the state set is $U$, and $h_U(u, a) = \{(uab, b) \mid \exists b \in O\ (uab \in U)\}$.*

We use $U$ to refer to both, a prefix-closed set of FSM traces, i.e., states and the FSM $(U, \varepsilon, I, O, h_U)$.

The quotient inference method includes two phases: first constructing an observation tree while identifying all the quotient's states and then determining transitions between the states. In the state identification phase, we apply inputs to an SUT, observe traces and add them to the observation tree $U$, initialized with $\{\varepsilon\}$. We perform Breadth First Search (BFS) on the tree and if the current node, i.e., state $u$, is $Z$-distinguishable from each already traversed state in the observation tree, we add the state $u$ into the set of states of the quotient. Otherwise, if there exists a traversed state $w$ which is $Z$-equivalent to $u$, we label the state $u$ with $w$, i.e., $label(u) = w$; $u$ is not included into the states of the quotient, and the behavior of the FSM $A$ will no longer be explored from the state $u$. Once the tree stops growing, all the states of a $(Z, I)$-quotient are identified. Transitions between the states of the quotient are determined from the transitions of the observation tree. Namely, a transition is considered if neither the source state nor any of its predecessors is labelled in the tree. If the target state is not labelled either, the same transition exists in the quotient. Otherwise the transition is redirected to the state which is used to label the target state.

The inference algorithm uses two procedures defined as follows.

Given an observation tree $U$, and an unknown FSM $A$ with input alphabet $X$, procedure *Extend_Node(A, u, Σ)* explores the behavior of FSM $A$ from a state reached by input sequence $u$ by applying the input sequences of a given set $\Sigma$ of *queries* and returns the augmented observation tree $U$. This procedure is shown in Listing 1.

Given an observation tree $U$, and an unknown FSM $A$ over the input set $X$ and output set $O$, procedure *Build_Quotient(A, I, Z, U)*, where $I \in X$, $Z \in I^*$, constructs the FSM $K = (Q, q_0, I, O, h_K)$ which is a $(Z, I)$-quotient of $A$, and returns an augmented observation tree $U$. This procedure is shown in Listing 2. While procedure Build_Quotient treats any given observation tree $U$, it can also be initialized with the trivial tree $U = \{\varepsilon\}$.

We illustrate procedure Build_Quotient by inferring a $(Z, I)$-quotient of FSM $A$ in Figure 8. The input set $X$ of $A$ is $\{a, b\}$, and we build a $(Z, I)$-quotient with the input set $Z = I = \{a, b\}$. Initially, $U = \{\varepsilon\}$, $Z = \{a, b\}$.

[1]Recall that a symbol of an FSM trace is a pair of an input from $I$ and an output from $O$, so every prefix takes the FSM from its initial state into some state.

Listing 1: procedure Extend_Node

```
1  procedure EXTEND_NODE (A, u, Σ), WHERE u ∈ U, Σ ⊆ X*
2      while there exists a1a2...ak ∈ Σ such that a1a2...ak ≠ v ↓I,
3              for any v ∈ Tr(u) do
4          reset A to its initial state
5          apply u ↓I to A
6          apply a1a2...ak to A, let the corresponding observed output sequence
7              be b1b2...bk
8          add the trace ua1b1a2b2..akbk and all its prefixes ua1b1,
9              ua1b1a2b2, ..., ua1b1a2b2..akbk to U
10     end while
11 end procedure
```

In the first execution of the first *for* loop, the only node $\varepsilon$ is traversed. After calling Extend_Node$(A, \varepsilon, Z)$, nodes $a1$ and $b0$ are added to $U$, and $\varepsilon$ is added to $Q$. In the second execution, node $a1$ is traversed. This time, nodes $a1a2$ and $a1b0$ are added to $U$. Since node $a1$ is not $Z$-equivalent to the node $\varepsilon$, $a1$ is also added to $Q$. In the next execution, Extend_Node$(A, b0, Z)$ is called, nodes $b0a1$ and $b0b0$ are added to $U$. Since node $b0$ is $Z$-equivalent to the node $\varepsilon$, it is labelled with $\varepsilon$. The procedure continues, nodes $a1a2$, $a1b0$, $a1a2a3$, and $a1a2b0$ are traversed, node $a1a2$ are added to $Q$, $a1b0$ is labelled with $a1$, $a1a2a3$ is labelled with $\varepsilon$, $a1a2b0$ is labelled with $a1a2$. In the end of the execution of the first *for* loop, $Q = \{\varepsilon, a1, a1a2\}$.

In the execution of the second *for* loop, the following transitions are added to $K$: $(\varepsilon, a1, a1)$, $(\varepsilon, b0, \varepsilon)$, $(a1, a2, a1a2)$, $(a1, b0, a1)$, $(a1a2, a3, \varepsilon)$, and $(a1a2, b0, a1a2)$. With these transitions, the quotient FSM in Figure 9 is obtained.

Finally, the third *for* loop is executed. For example, node $b0$ is labelled with $\varepsilon$, in $U$ there is a transition $(b0, a1, b0a1)$, in $K$ there is a transition $(\varepsilon, a1, a1)$, so, the node $b0a1$ is labelled with $a1$. The resulting tree $U$ is depicted in Figure 10.

The procedure Build_Quotient$(A, I, Z, U)$ partitions the explored nodes of the observation tree $U$ according to their output reactions to $Z$ so that each block becomes a state of the quotient, however, it does not guarantee that the resulting quotient preserves traces that distinguish them. Thus, the quotient returned by the procedure still needs to be checked for consistency with the observation tree $U$.

Let $a_1a_2...a_k$ be an input sequence from $X^*$ and $K$ be an FSM ($Q$, $q_0$, $I$, $O$, $h_K$) where $I \subseteq X$. A trace $a_1b_1a_2b_2...a_kb_k$ is *incompatible* with state $q$ of $K$ iff there is no trace $c_1b_1c_2b_2...c_kb_k$ of state $q$ such that for all $i$, if $a_i \in I$ then

Listing 2: procedure Build_Quotient

```
1  procedure BUILD_QUOTIENT(A, I, Z, U)
2      for each state u of U being traversed during Breadth First Search
3              such that u has no labelled predecessor do
4          Extend_Node(A, u, Z)
5          if u is Z-equivalent to a traversed state w of U then
6              label u with w, i.e., label(u) = w
7          else
8              add u into Q
9              Extend_Node(A, u, I)
10         end if
11     end for
12     for each transition (u, ab, v), such that neither state u nor any of its
13             predecessors is labelled do
14         if v is not labelled then
15             add transition (u, ab, v) to K
16         else
17             add transition (u, ab, w) to K, where w = label(v)
18         end if
19     end for
20     for each node u' labelled with u do
21         label the successors node of u' such that
22         for each transition (u', ab, v) do
23             if there is a transition (u, ab, w) in K then
24                 label v with w
25             end if
26         end for
27     end for
28     return the labelled tree U and the resulting FSM
29         K = (Q, ε, I, O, h_k) as a (Z, I)-quotient of the FSM A
30 end procedure
```

$c_i = a_i$. Notice that for a $(Z, X)$-quotient, i.e., $I = X$, the above definition requires only that a given trace should not be in $Tr(q)$. We present a more general version for dealing with counterexamples, used in the next section.

Given an observation tree $U$ and a $(Z, I)$-quotient $K = (Q, q_0, I, O, h_K)$ obtained from $U$, we say that a node of the tree $U$ labelled with $q \in Q$ has an *inconsistent* label, if it has a trace that is incompatible with state $q$ of the $(Z, I)$-quotient $K$; the trace is called an inconsistency witness for $q$.

The idea of resolving this inconsistency is to extend the set $Z$ with the input projection of the witness and repeat the procedure Build_Quotient($A$,

$I$, $Z$, $U$) until the resulting tree has no inconsistent label for the obtained quotient. This idea is implemented in the fix-point procedure *Fix_Point_Consistency(A, I, Z, U, K)* shown in Listing 3.

Listing 3: procedure Fix_Point_Consistency

```
1  procedure FIX_POINT_CONSISTENCY(A, I, Z, U, K)
2      while there exists an unprocessed witness w for state q
3              such that its input projection is not in Z do
4          if there does not exist a trace v ∈ Tr_U(q) such that v ↓_I = w ↓_I then
5              apply (qw) ↓_I to A in the initial state to obtain trace v ∈ Tr_U(q)
6          end if
7          if w ≠ v then
8              Z' = Z ∪ {w ↓_I} and I' = I ∪ inp(w)
9              Build_Quotient(A, I', Z', U), returning an updated observation
10                 tree and quotient
11             Z = Z' and I = I'
12         end if
13         mark w as processed
14     end while
15     return the last labelled observation tree and quotient
16 end procedure
```

In our running example, the observation tree $U$ in Figure 10 does not have any inconsistent label.

The following theorem claims that the above method can be used to infer an initial $Z$-quotient of the FSM from which traces are collected during testing.

**Theorem 4.** *The procedures Build_Quotient and Fix_Point_Consistency applied to a deterministic FSM A terminate. The resulting FSM K is an initial $(Z, I)$-quotient of FSM A, moreover, it is a minimal machine.*

### 3.2.3 Dealing with Counterexamples

A *counterexample CE* for a $(Z, I)$-quotient is a trace of the FSM to be inferred that is incompatible with the initial state of the $(Z, I)$-quotient.

The inference method including counterexample processing is described in the following procedure.

Given an unknown FSM $A$ with input alphabet $X$, output alphabet $O$, $I \subseteq X$, $Z \subseteq I^*$, procedure *Infer(A, I, Z)* returns updated $I'$ and $Z'$ and

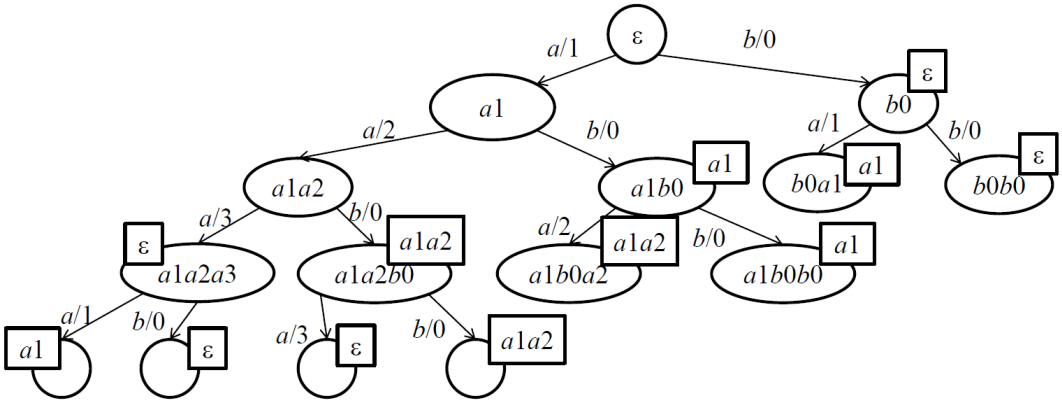


Figure 10: Observation tree $U$

constructs an FSM which is an initial $(Z', I')$-quotient of $A$. This procedure is shown in Listing 4.

Listing 4: procedure Infer

```
1  procedure INFER(A, I, Z)
2      Build_Quotient(A, I, Z, {ε}) returning U and K = (Q, q0, I, O, hK)
3      Fix_Point_Consistency(A, I, Z, U, K)
4      ask for counterexamples for K
5      while there exists an unprocessed counterexample CE do
6          U = U ∪ CE
7          Fix_Point_Consistency(A, I, Z, U, K)
8      end while
9      return I, Z and the last quotient
10 end procedure
```

In our example, we have performed steps 2 and 3 to obtain the $(Z, I)$-quotient depicted in Figure 10. In step 4, suppose we obtain a counterexample $a1a2b0b0b0a3b0b0a3$. In step 6 of the *while* loop, we update the tree $U$ with the counterexample and obtain the tree in Figure 11.

Now, the procedure Fix_Point_Consistency is called. In the first execution of the *while* loop, we determine that in $U$, state $a1a2b0$ labelled with $a1a2$ has an inconsistent label, since it has a trace $b0b0a3b0b0a3$ incompatible with state $a1a2$ in the $(Z, I)$-quotient, as this state has no such trace. Thus, the witness $w$ is $b0b0a3b0b0a3$. We first apply $a1a2$ followed by $bbabba$ to $A$ in the initial state and obtain the trace $b0b0a3b0b0a1$, which is different from $w = b0b0a3b0b0a3$. Thus, we extend $Z$ to $\{a, b, bbabba\}$ and execute

the procedure Build_Quotient. The new $(Z, I)$-quotient is represented in Figure 12.

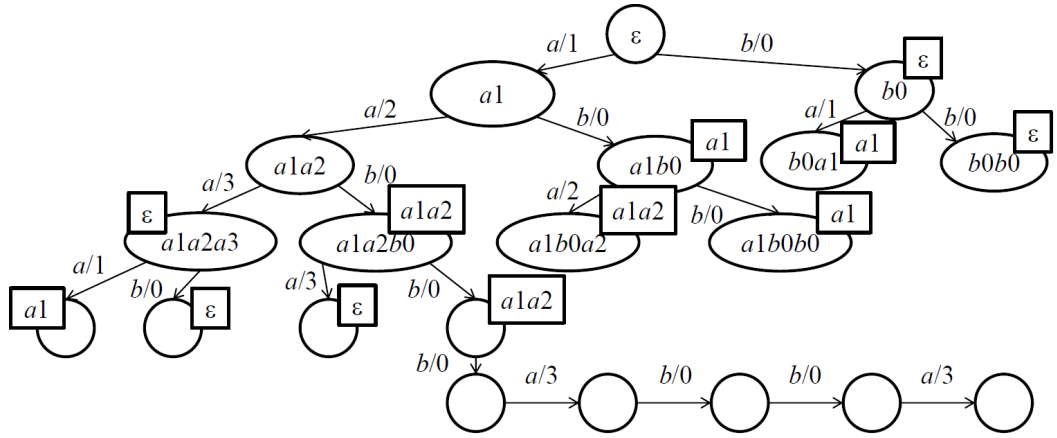


Figure 11: The observation tree $U$ updated with the counterexample

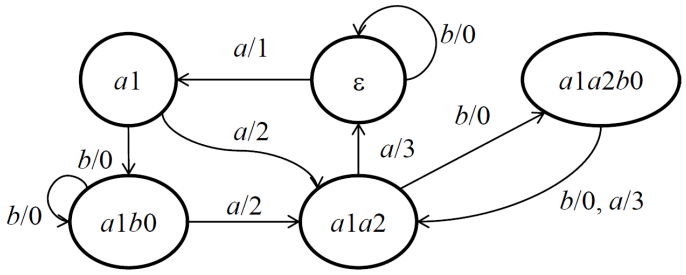


Figure 12: FSM $K_1$, an initial $\{a, b, bbabba\}$-quotient of FSM $A$

In the second execution of the *while* loop, we identify that in $U$, state $a1b0b0$ labelled with $a1b0$ has an inconsistent label, since it has the trace $a2b1$ incompatible with state $a1b0$ in the $(Z, I)$-quotient in Figure 12. In this case, the witness $w$ is $a2b1$. State $a1b0$ has a trace $a2b0$, which has the input projection $ab$ and is different from $w$. Thus, we extend $Z$ to $\{a, b, bbabba, ab\}$ and execute Build_Quotient again. The obtained $(Z, I)$-quotient is equivalent to the FSM to be inferred. With this result, the execution of Fix_Point_Consistency and Infer procedures terminates.

### 3.2.4 Strategies in Counterexample Processing

The Fix_Point_Consistency procedure used in counterexample processing can be implemented using an arbitrary strategy for searching witnesses. For example, suffixes of the counterexample of increasing length can be checked. Indeed, a bottom-up strategy, as in the Suffix1by1 method [20], identifies a minimal-length witness suffix of the counterexample. Contrary to methods based on observation tables such as [20, 38], where the length of the witness is a concern, in our method we can just as well perform a top-down search which in many cases (including Moore locks and counters) is more efficient. Following Rivest and Schapire [32], it is also possible to identify the shortest witness by binary search on the counterexample.

At the same time, a distinctive feature of our method is that once the counterexample initial processing has yielded a new quotient, our fixpoint procedure is able to use witnesses that are in different branches of the observation tree. In the experiments reported in this deliverable, we have simply reused a BFS strategy to look for all witnesses.

Finally, let us remark that checking whether a trace is a witness is straightforward despite the apparent generality of the definition of compatibility (which uses negation of an existential quantifier): we process it from left to right, recording at each input the potential states in which the quotient can be, based on the output from the trace provided by the black box and the corresponding transitions that have this output in the quotient to find the next states.

## 3.3 Experiments

### 3.3.1 Random Machines

To evaluate the proposed method, we use it to infer randomly generated FSM. Although we know each FSM to be inferred and could compute a minimal counterexample, we simulate a practical CE search by first randomly walking through a generated FSM and then comparing the output sequence with that of the inferred quotient. Random walks are restarted up to a limit (defined by the number of states). Once this limit is reached, the resulting machine is considered equivalent to the FSM and the inference terminates. In the experiments, we determine the average number of generated queries (tests) needed to terminate the inference of a random machine with a varying number of states from 50 to 1000, 10 inputs and 10 outputs.

Figure 13 presents the numbers of queries generated by the proposed method and the table-based $LM^*$ algorithm [34] combined with the *Suffix1by1* method for processing counterexamples [20]. With our method, the

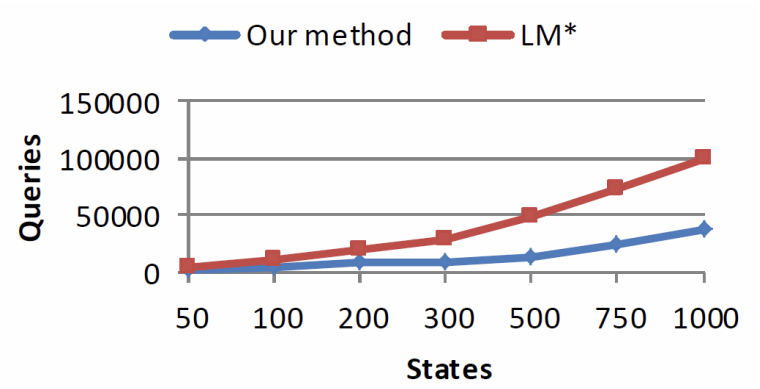


Figure 13: Comparing the proposed method with the $LM^*$ algorithm

number of needed queries grows slowly whereas with the $LM^*$ method, it increases quickly. We also compared our method with the table-based method $L_m^*$ using the example provided by its authors in [38]. The FSM to be inferred has 6 states, 4 inputs and 3 outputs. For both methods we consider only two counterexamples, using which $L_m^*$ infers the machine with 155 queries. When the initial set of inputs $I$ is empty, our method also generates as many queries; however, when all the inputs are considered (which is exactly what $L_m^*$ does), the number of queries is cut to 124.

### 3.3.2 SIP Protocol

The *SIP* protocol is widely used in telecommunications and especially for controlling voice and video calls over *IP*. This protocol has already been used by other authors to assess inference methods, although this has usually been done on simulated implementations (in NS-2). In our experiment, the model inference tool implementing our method directly interacts with a real-world implementation of *SIP*. We have created a new account on *iptel.org* which provides free SIP services. According to their website, this service is also used for software/hardware interoperability testing. Our tool plays the role of client and infers a model of the SIP server implemented by iptel.org. The inputs are parameterized to make a call to the Echo test service. This is a special account to check if the client is correctly configured to use the service.

*SIP* contains 14 different types of requests, in this experiment we focused on the main four types commonly used to make a call: *Register*, *Invite*, *Ack* and *Bye*. Outputs are the responses code and Timeout when no response is received. As we have no information about the distinguishing sequences of

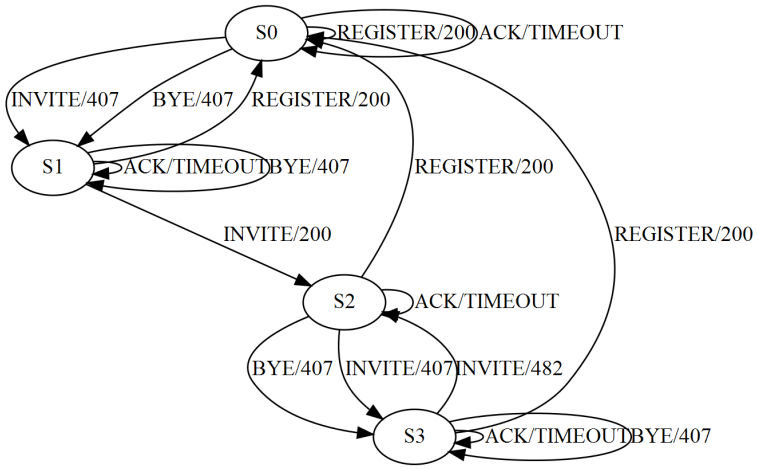


Figure 14: Inferred SIP protocol

*SIP*, the set $Z$ is initially empty. Counterexamples are provided manually.

Figure 14 shows the model of *SIP* protocol implemented by *iptel.org* and obtained with 26 requests in 10 seconds. This model is consistent with the specification of the protocol. The only counter-example needed is *Invite.Invite*: In order to call someone on *iptel.org*, we have to be authenticated before. Here, the first *Invite* sent by the client is not enough to make a call. A "Proxy authentication required" response (code 407) is received with the nonce needed for the authentication. Then, the authenticated Invite is built and sent and the "OK" response (code 200) is obtained.

Our tool could easily infer any other SIP server implementation available on the Internet just by changing the service provider address.

## 3.4 Related Work

D. Peled's paper [29] triggered the interest in using inference algorithms (in this case $L^*$ [4]) for software validation. The proposed method actually relies on the $W$-method [40] as an oracle for finding counterexamples. It was followed by a number of approaches to use $L^*$ or modified versions of it in a testing context [1, 34, 35, 38]. Most of these approaches have used tables (as in $L^*$) as the basic structure for recording observations and building models.

Our approach departs from those table-based approaches inspired by machine learning algorithms that record positive and negative samples. First, it records behaviors (traces) of the SUT in a tree structure. Saving on common prefixes is not really an argument there, as similar space-saving methods are

also used in table-based methods (with filters [26] and dictionaries to cache queries). The key point is that the tree structure makes it possible to compare nodes of the tree with states of the model even when such nodes are not associated with a row label in a table. Our witness identification procedure can find incompatibilities with observations even recorded earlier. Also, it avoids requiring suffix-closure properties as in many table-based approaches, yet it is sound and produces models consistent with observations contrary to Rivest [32].

The $L_m^*$ algorithm presented in [38] is a table-based approach that capitalizes on previous improvements of such methods. It uses binary search as [32] to reduce the worst case complexity factor from linear in counterexample length to logarithmic. It also attempts avoiding suffix-closure by a new notion of "semantic suffix-closedness". However, this notion and the algorithm proposed to implement it are less powerful than our fixpoint witness search (cf. Section 3.3). The main reason is that like all table-based approaches, it just looks at suffixes of a counterexample, whereas our approach will make the most of a counterexample by comparing nodes throughout the tree, not just in the path provided by the counterexample. In our running example, it would typically fail to identify the inconsistent label of the state $a1b0b0$ that we find in the second round of the *while* loop.

The work by Meinke [24] is another attempt to define a method of inference for Mealy automata, CGE. It also uses notions of congruences of outputs (similar to the equivalence we assume with the abstraction on concrete outputs) and states; therefore it has some similarities with our quotient-based approach. It is also designed to make the most of every observation requiring new queries only to get information that cannot be deduced from the existing set, and it uses a similar fixpoint iteration over the rules that are used as a compact way of representing Mealy automata. However, in the form presented in [24], CGE only does passive inference: it does not define how new queries would be derived. Also, it does not consider equivalence on inputs, as we do with our definition of compatibility.

Our quotient approach generalizes the $k$-quotient method presented in [17]. First, instead of considering all sequences up to a given length, it reduces this exponential complexity with the $Z$ set, which in our case only contains sequences of inputs that at some point separate states in a quotient. Importantly, $Z$ does not even have to contain all inputs. Our new approach is incremental in $Z$ and $I$. Another major improvement is that [17] only built an initial $k$-quotient for an SUT, but further refinements derived from counterexamples would no longer be quotients. Our new approach makes it possible to derive a converging sequence of quotients that are well-defined approximations of the SUT.

A related area as defined in [3] is specification mining, which is a machine learning approach to discover formal specifications of the protocols that code must obey when interacting with an application program interface or abstract data type. From the model inference point of view, this belongs to passive inference, where model is built based on a given set of traces. On the contrary, our work belongs to active inference, where the next input to the unknown FSM is decided according to the observation so far. Among the work in this area, [20] uses $k$-equivalence as the criterion to merge states. At the same time, it assumes variable values are accessible, while our work follows a black-box approach.

In [9], static web service signature information described in WSDL is used to construct behavior protocol model of web service, and testing is used to prune false data dependence between input/output parameters of operations defined in the WSDL. In our work, a finite state model is built based on run time observations of software execution, and testing is used to obtain system behavior corresponding to various inputs.

# 4 Driver Generation for Web Applications

The inference approaches presented before need to communicate actively with the application but they work at an abstract level whereas applications mainly work at the concrete *HTTP* level. To fill this gap, crawling techniques can be used to extract all inputs and outputs from the application and get the corresponding test driver which will act as a proxy between these two levels. More details about the test driver generation method can be found in [19]

## 4.1 Web application abstraction

Modern web applications can be developed in multiple languages dedicated for web applications, e.g., *PHP*, *ASP*, and not designed for them, e.g., *Python*, *Java*, but all applications generate a common language: *HTML*. Inputs and outputs will be extracted from the *HTML* source code of a page.

An input is an action provided by the application and executed using a link or a form element in *HTML*. We define an input $I$ as a tuple ($M$, $A$, $P$) where $M$ is the *HTTP* method used, $A$ is the URL address of a web page to handle the request and $P$ is a finite set of couples (*name, values*) where *name* is the name of the parameter and *values* is a set of strings corresponding to the possible values. In the case of a link, $P$ could be the empty set and $M$ is

always the *GET* method. Parameters can have more than one value, e.g. in Listing 5 and even for a link, e.g. in Listing 6.

Listing 5: select element

```
<select name="id">
  <option value="1">1</option>
  <option value="2">2</option>
  <option value="3">3</option>
</select>
```

Listing 6: link with multiple values for one parameter

```
http://test/index.php?id=1&id=2&id=3
```

An output is a page, independently of the content. For example, two profile pages are considered as the same page: only the content changes but not the structure. To extract the different outputs of the application, we only have to consider the structure of the page. Then, for the structure-based differentiation, page content is removed and a page tree is constructed from the tags which are related to the structure of the page. Two pages with different page trees are different.

Some kind of contents like ads or dynamic contents can slightly modify the structure of the page by adding some tags; this can lead to false negatives during the differentiation. This is why we use a threshold. Below this threshold, two *HTML* pages are considered as the same output.

Like inputs, outputs may contain parameters. We define an output parameter as a part of the source code which directly depends on an input parameter, in this case it is deterministic, otherwise it is a nondeterministic parameter, typically ads, date time and dynamic content like *RSS*, *Chat*. To detect them, each output page is requested several times, then similar parts are matched together using a pairwise alignment algorithm and remaining parts are considered as parameters.

For example, with a first request we can obtain Listing 7, and with a second one Listing 8. The *div* and *span* elements match and only the content of the span tags are different. We can see that each of these two *span* contains one parameter.

Listing 7: A page obtained for the first time

```
<div id="#profile">
  <span id="#username">John</span>
  <span id="#money">100</span>
```

```
4 </select>
```

Listing 8: The same page obtained for the second time

```
1 <div id="#profile">
2   <span id="#username">Henri</span>
3   <span id="#money">300</span>
4 </select>
```

We have defined a method to extract all inputs in a page, differentiate two pages and extract the output parameters. Combining these methods with a crawler allows us to build the abstraction and the corresponding test driver automatically.

## 4.2 Crawling strategy

The crawling process has to browse the different pages as much as possible in order to find the maximum of different outputs and inputs. Even if the goal is to generate a test driver automatically, some data still need to be provided by the tester, like credentials.

A naive crawling method would only enumerate the reachable pages. Depth-first-search or breath-first-search strategies can also be used, starting from a given page and stopping when all possible pages are explored. However, these methods do not work anymore with modern web applications where the order of visit is important.

Consider a simple web application with the start page *index.php*, then *login.php* and, after being authenticated, *view.php*, *search.php* and *logout.php*. A classical crawler will start from *index.php*, then the only page accessible page without being authenticated is *login.php*. After being authenticated, the crawler can access *view.php*. This page contains a link to *logout.php*. Since the crawler visits *logout.php*, all other pages found, e.g., *search.php*, will not be accessible anymore because *logout.php* changes the internal state of the application. This is why we need another strategy.

To avoid this problem, we do not use a single request to retrieve a page but a sequence of requests combined with depth-first-search and page differentiation to avoid loops and crawling the same page too many times. The sequences send for the last example are described in Listing 9.

Listing 9: Sequence of inputs

```
1 index.php
2 index.php;login.php
3 index.php;login.php;view.php
```

```
4 index.php;login.php;view.php;logout.php
5 index.php;login.php;search.php
```

By sending sequences of inputs, we are able to explore the entire application automatically even if the internal state changes.

### 4.2.1 Optimizations & heuristics

Applications can use a parameter to select the content of the page. Typically, the page *index.php* has one parameter id to select the content or in gallery-style application, each element of the gallery has one id associated. Each of these links differs only by one parameter value and lead to the same page. To avoid visiting this entire set of links we use this heuristic: if at least two pages are the same and parameters differs only by one value, other similar pages will not be visited.

A form element can contain multiple actions. In WebGoat Stored XSS lesson, the main page has a list of employees and four buttons, each associated with a different action. During the crawling, actions are extracted and parameters are associated with them. Therefore, some parameters may be unnecessary. This is why we filter the parameters if we find the same action somewhere else with a subset of parameters.

# 5 Model assessment

In this section, we demonstrate why it is important to have some automatic procedures to generate the ASLAN++ models by comparing the generated models with the handwritten models.

## 5.1 Inferred vs. handwritten models

To assess the need for a black-box inference component, we compare the handwritten model with the inferred one for the *WebGoat* application. Listing 10 was constructed by hand and described in detail in Deliverable D5.2 [36]. Listing 11 represents the same part of the application but in the inferred model by SIMPA, the inference component of the SPaCIoS tool.

Listing 10: A part of the manual model of the WebGoat Stored XSS lesson.

```
1 specification example
2 channel_model CCM
3
4 entity Environment {
5   types
```

```
6       profile < text;
7       content < text;
8       cookie < message;  % cookie < text;
9
10    symbols
11      login( agent, symmetric_key ) :  message;
12      logout:    message;
13      viewProfile ( profile ):  message;
14      editProfile ( profile ):  message;
15      listProfiles:      message;
16      updateProfile( profile, content ):  message;
17      findProfile( content ):    message;
18      ...
19    symbols % local variables for the lesson example
20      webServer:  agent;
21      larryProfile, peterProfile: profile;
22      larry, peter:  agent;
23      larryContent, peterContent:  content;
24
25    entity Session (U, S: agent) {
26      entity User( Actor, S : agent ) {
27        symbols
28          P:  profile;
29          ProfileContent:  content;
30          Cookie: cookie;
31          SentProfiles: profile set;
32
33        body { % of User
34          % we define a trace through the state machine so that every message
                is sent at least once
35          % login - viewProfile - editProfile - updateProfile - listStaff -
                searchStaff - logout
36
37          % login
38          [Actor] *->* S : login( Actor, shared_secret:( password( Actor, S )
                ) );  % Actor itself logs in with shared secret
39          S *->* [Actor] : secret_cookie:( ?Cookie) ;  % user gets his cookie
40
41          % viewProfile
42          if ( Actor->canView( ?P ) ) {
43            [Actor] *->* S : Cookie.viewProfile( P );
44            S *-> [Actor] : shared_profile:( ?ProfileContent );
45          }
46
47          % editProfile
48          if( Actor->canEdit( ?P ) ) {
49            [Actor] *->* S : Cookie.editProfile( P );
50            S *->* [Actor] : shared_profile:( ?ProfileContent );  % User
                  learns name of the profile
51          }
52
53          ...
54
55        }
56      }
57
58      entity Server( U, Actor : agent ) {
59        symbols
60          UP:      public_key;  % user's pseudonym
61          Profile:      profile;
62          Cookie: cookie;
63          Content: content; % User can search for a profile by name
```

```
64          Parent : nat;  % identifies the parent entity
65          SentProfiles: profile set; % used by Server to send profiles.
66        body {
67          while(true) {
68            select {
69
70              % Login action
71              on( [?U]_[?UP] *->* Actor: login( ?U, shared_secret:( password(?
                    U, Actor) ) ) ): {
72                secret_cookie:(Cookie) := fresh();
73                cookies(Actor)->contains( (U, Cookie) );
74                Actor *->* [U]_[UP] : Cookie;
75
76                select{ on(child( ?Parent, IID ) ): {} }
77                while( canView( U, ? ) ) {
78                  shared_profile_set( Parent )->add( U );
79                }
80              }
81
82              % ViewProfile action
83              on( [?U]_[?UP] *->* Actor: ?Cookie.( viewProfile(?Profile) ) &
                    cookies(Actor)->contains( (U, Cookie) ) & U->canView(
                    Profile ) ): {
84                  %select{ on(child( ?Parent, IID ) ): {} }
85                  select{ on(contentOf( Profile, ?Content ) ):{} }
86                  %  this parent entity is stored in ?Parent
87                  Actor *->* [U]_[UP] : shared_profile:( user_auth:(Content) )
                        ;
88              }
89
90              % Logout action
91              on( [?U]_[?UP] *->* Actor: ?Cookie.(logout) & cookies(Actor)->
                    contains( (U, Cookie) ) ): {
92                  cookies(Actor)->remove( (U, Cookie) );
93                }
94                shared_profile_set( Parent )->remove( U );
95              }
96
97              ...
98            }
99          }
100       }
101     }
102     ...
103   }
104     ...
105 }
```

Listing 11: A part of the model inferred by SIMPA for the WebGoat Stored XSS lesson.

```
1  specification WEBGOAT_STORED_XSS
2  channel_model CCM
3
4  entity Environment {
5    symbols
6      a : agent;
7      editionpage(text) : message;
8      editprofile(text) : message;
9      home(text) : message;
```

```
10      listing(text) : message;
11      login(text, text) : message;
12      logout(text) : message;
13      profilepage(text) : message;
14      ...
15
16    entity WEBGOAT_STORED_XSS(Actor : agent, Other : agent){
17      symbols
18        State : nat;
19        Xsspayload : text;
20        Codeeditionpage : text;
21        Codehome : text;
22        Codelisting : text;
23        Codeprofilepage : text;
24        ...
25
26      body {
27        State := 0;
28        while (true){
29          select {
30            on(State = 0): {
31              select {
32                on(Other -> Actor: login(?Profileidlogin, ?Passwordlogin)): {
33                  if ((Profileidlogin = s101) & (Passwordlogin = slarry) | (
                        Profileidlogin = s111) & (Passwordlogin = sjohn)){
34                    Codelisting := sOK;
35                    Actor -> Other: listing(Codelisting);
36                    State := 1;
37                  }
38                  if ((Profileidlogin = s101) & (Passwordlogin = sfoo) | (
                        Profileidlogin = s101) & (Passwordlogin = sjohn) | (
                        Profileidlogin = s111) & (Passwordlogin = sfoo) | (
                        Profileidlogin = s111) & (Passwordlogin = slarry) | (
                        Profileidlogin = s666)){
39                    Codehome := sOK;
40                    Actor -> Other: home(Codehome);
41                    State := 0;
42                  }
43                }
44                on(Other -> Actor: logout(?Profileidlogout)): {
45                  Codehome := sOK;
46                  Actor -> Other: home(Codehome);
47                  State := 0;
48                }
49                ...
50              }
51            }
52            on(State = 1): {
53              select {
54                on(Other -> Actor: login(?Profileidlogin, ?Passwordlogin)): {
55                  if ((Profileidlogin = s101) & (Passwordlogin = slarry) | (
                        Profileidlogin = s111) & (Passwordlogin = sjohn)){
56                    Codelisting := sOK;
57                    Actor -> Other: listing(Codelisting);
58                    State := 1;
59                  }
60                  if ((Profileidlogin = s101) & (Passwordlogin = sfoo) | (
                        Profileidlogin = s101) & (Passwordlogin = sjohn) | (
                        Profileidlogin = s111) & (Passwordlogin = sfoo) | (
                        Profileidlogin = s111) & (Passwordlogin = slarry) | (
                        Profileidlogin = s666)){
61                    Codehome := sOK;
```

```
62                    Actor -> Other: home(Codehome);
63                    State := 1;
64                  }
65                }
66                on(Other -> Actor: logout(?Profileidlogout)): {
67                  Codehome := sOK;
68                  Actor -> Other: home(Codehome);
69                  State := 0;
70                }
71                on(Other -> Actor: viewprofile(?Profileidprofile)): {
72                  Codeprofilepage := sOK;
73                  Actor -> Other: profilepage(Codeprofilepage);
74                  State := 1;
75                }
76                ...
77              }
78            }
79          }
80        }
81      }
82    }
83    body {
84      new WEBGOAT_STORED_XSS(system, a);
85    }
86  }
```

**Pros.** The first advantage of the inferred model (Listing 11) is that it has been produced in less than one minute and considering the system as a black-box only, while the manual one required a lot of time and detailed knowledge of the application. As a second advantage, since the inputs are based on the actions of the application, the model is easy to read and understand. For instance, the inferred model generates only one entity to represent the system. Each state can be identified clearly in the switch structure.

**Cons.** The inferred model (Listing 11) is an abstraction of the real system, and the level of detail may not show all the internal behaviors. For example, the handwritten model also specifies the security properties of the individual channels, whereas the inferred model uses a basic, unsecured channel.

The handwritten model is obviously better than the inferred one. But it needs a lot of time to be written and the user must have some experience with the specification language to be able to develop one from scratch. On the other hand, the inferred model can be obtained automatically and quickly. It is the perfect base for the user who can start to work on this model to improve it. This shows the **need** of the inferred model and reveals the importance of having such **automatic** procedures for generating models.

## 5.2 Code-based vs. handwritten models

### 5.2.1 Model Extraction in a Nutshell

Complementary to the black-box model inference, IEAT has developed white-box model extraction techniques. In order to compare the resulting models with manually written ones, we briefly present the JMODEX model extraction tool. A detailed description can be found in Deliverable D2.2.2 [37].

The JMODEX tool analyzes the code of a JSP/Servlet-based web application and produces an ASLAN++ model. Figure 15 shows a high-level view of the model extraction process, with its two main analysis steps, each implemented by a component which is briefly described in the following.

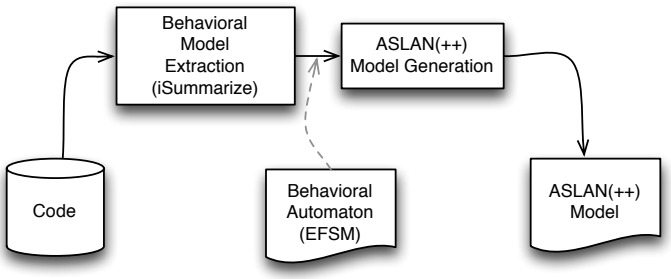


Figure 15: jModex Processing Steps

- ISUMMARIZE – this component analyzes the code of the application and builds an EFSM in which each transition represents an execution path through the program, while a state represents the entry/exit point of the system under analysis or a loop header. For each transition, the constraints that enable the execution of the transition are captured (e.g., branch conditions that must be satisfied to follow a particular execution path), together with updates to state variables relevant to the application and to the analysis goal (e.g., session attributes of the servlet, or the output of the program). The guard conditions and the updates capture the usage of every relevant value, including the inputs of the application (e.g., values coming from the user via request parameters, or database values). More details about the EFSM extraction algorithm can be found in Deliverable D2.2.2 [37, Section 2.2.2].

- ASLAN++ CONVERTER – receives an EFSM built by the previous component, and transforms it into an ASLAN++ model. For this purpose, a control tree is built, capturing the required ASLAN++ statements

and how they will be chained/nested in order to capture the semantics of the translated EFSM. After that, the guards and the updates from the automaton are translated to ASLAN++ statements/expressions according to translation schemata which are further used as conditions or added to the bodies of corresponding control statements from the control tree. As a result, the ASLAN++ model is built. More details about the conversion process including the mapping of code constructs to ASLAN++ are presented in Deliverable D2.2.2 [37, Section 2.2.3].

### 5.2.2 Sample Models and Discussion

As part of the assessment, we present a portion of the ASLAN++ model for the *BookStore* application (Listing 12) constructed by hand and described in detail in Deliverable D5.2 [36], with some updates. Listing 13 shows the same part of the model inferred by JMODEX in its present development stage[2], with some manual simplifications (currently being automated). We compare the two models and discuss pros and cons regarding the automatic extraction of ASLAN++ models.

Listing 12: A part of the manual model of BookStore.

```
specification Bookstore_Login_Manual
channel_model CCM

entity Environment {
  entity Session(S: agent) {
    symbols
    ...
    entity Server(Actor, U: agent, Sess: text.message set) {
      symbols
        getUID(text): text;
        getURights(text): text;
        request(text, text.message set): text;
        ...
        validU(text.message set, text, text): fact;
        paramsFromText(text) text.text set;
        ...
        Params: text.message set;
        ...
        sFormName: text;
        sLogin: text;
        sEmpty: text;
                ...
      body {
                ...
        %Start server actions
        while(true) {
          select {
            on (?U *->* Actor: request(sLogin,?Params)
                & ?Params->contains((sFormName, sLogin))
                & ?Params->contains((sFormAction, slogin))): {
```

[2]Some information, e.g., exception handling, is not captured yet in the model.

```
31                if (validU(Params, ?Uname, ?Password)) {
32                  if (Sess->contains((sUID, ?Remove)))
33                    Sess->remove((sUID, Remove));
34                  if (Sess->contains((sURights, ?Remove)))
35                    Sess->remove((sURights, Remove));
36                  Sess->contains((sUID, getUID(Uname)));
37                  Sess->contains((sURights, getURights(Uname)));
38                  if (Params->contains((sRetPage,?RetPage)
39                      & RetPage != sLogin & RetPage != sEmpty)
40                    if (Params->contains((sQuerystring,?Querystring))
41                      U *->* Actor: request(RetPage,paramsFromText(Querystring
                            ));
42                    else
43                      U *->* Actor: request(RetPage,{});
44                  else
45                    U *->* Actor: request(shoppingCart,{});
46                 } else
47                     U *->* Actor: request(login,{});
48              }
49              on (?U *->* Actor: request(sLogin,?Params)
50                  & ?Params->contains((sFormName, sLogin))
51                  & ?Params->contains((sFormAction, slogout))): {
52                if (Sess->contains((sUID, ?Remove)))
53                  Sess->remove((sUID, Remove));
54                if (Sess->contains((sURights, ?Remove)))
55                  Sess->remove((sURights, Remove));
56                Sess->contains((sUID, sEmpty));
57                Sess->contains((sURights, sEmpty));
58                Actor*->*U: viewLogin;
59              }
60              ...
61            }
62          }
63        }
64        ...
65      } % end Server
66    ...
67  } % end Environment
```

Listing 13: A part of the model inferred by JMODEX for BookStore.

```
1  specification Bookstore_Login_Simple
2  channel_model CCM
3
4  entity Environment {
5   symbols
6   ...
7   entity Session(S:agent) {
8     entity Server(Actor:agent, U:agent, Sess:message.message set) {
9       symbols
10      ...
11      body {
12        while(true) {
13          select {
14            on (?U*->*Actor:login_jsp(?Params)): {
15              select {
16                %PASSWORD IS OK BUT USER NAME IS MISSING
17                on (Params->contains((sPassword,?Password))
18                    & Params->contains((sFormAction,?FormAction))
19                    & Params->contains((sFormName,?FormName))
20                    & (!(Params->contains((sLogin,?)))
```

```
21                      | Params->contains((sLogin,sEmpty)))
22                    & !(?Password = sEmpty) & Params->contains((sPassword,?))
23                    & !(Login_jsp_LoginAction_v53=0)
24                    & ?FormAction = slogin & ?FormName = sLogin): {
25                  if (Params->contains((sPassword,?Password)) { }
26                  else Password := oNull;
27                  if (Sess->contains((sUserID,?UserID))) {
28                    Sess->remove((sUserID,UserID));
29                  }
30                  if (DBAccessFunction(
                      select_member_id__member_level_from_members_where_member_login_EQ_
                      .sEmpty._and_member_password_EQ_.Password,1)!=oNull) {
31                    Sess->contains((sUserID,DBAccessFunction(
                        select_member_id__member_level_from_members_where_member_login_EQ_
                        .sEmpty._and_member_password_EQ_.Password,1)));
32                  }
33                  if (Params->contains((sPassword,?Password))) { }
34                  else Password := oNull;
35                  if (Sess->contains((sUserRights,?UserRights))) {
36                    Sess->remove((sUserRights,UserRights));
37                  }
38                  if (DBAccessFunction(
                      select_member_id__member_level_from_members_where_member_login_EQ_
                      .sEmpty._and_member_password_EQ_.Password,2)!=oNull) {
39                    Sess->contains((sUserRights,DBAccessFunction(
                        select_member_id__member_level_from_members_where_member_login_EQ_
                        .sEmpty._and_member_password_EQ_.Password,2)));
40                  }
41                }
42                %USER NAME IS OK BUT PASSWORD IS MISSING
43                on (... %long condition% ...): {
44                   ...
45                }
46                %LOGOUT
47                on (Params->contains((sFormAction,?FormAction)) & Params->
                    contains((sFormName,?FormName)) & ?FormAction = slogout &
                    ... %long condition% ...): {
48                  if (Sess->contains((sUserID,?UserID))) {
49                    Sess->remove((sUserID,UserID));
50                  }
51                  Sess->contains((sUserID,sEmpty));
52                  if (Sess->contains((sUserRights,?UserRights))) {
53                    Sess->remove((sUserRights,UserRights));
54                  }
55                  Sess->contains((sUserRights,sEmpty));
56                }
57                %USER NAME AND PASSWORD ARE OK
58                on (Params->contains((sLogin,?Login))
59                    & Params->contains((sPassword,?Password))
60                    & Params->contains((sFormAction,?FormAction))
61                    & Params->contains((sFormName,?FormName))
62                    & !(?Login = sEmpty) & Params->contains((sLogin,?))
63                    & !(?Password = sEmpty) & Params->contains((sPassword,?))
64                    & !(Login_jsp_LoginAction_v53=0)
65                    & ?FormAction = slogin & ?FormName = sLogin): {
66                  if (Params->contains((sLogin,?Login))) { }
67                  else Login := oNull;
68                  if (Params->contains((sPassword,?Password))) { }
69                  else Password := oNull;
70                  if (Sess->contains((sUserID,?UserID))) {
71                    Sess->remove((sUserID,UserID));
72                  }
```

```
73                  if (DBAccessFunction(
                      select_member_id__member_level_from_members_where_member_login_EQ_
                      .Login._and_member_password_EQ_.Password,1)!=oNull) {
74                    Sess->contains((sUserID,DBAccessFunction(
                        select_member_id__member_level_from_members_where_member_login_EQ_
                        .Login._and_member_password_EQ_.Password,1)));
75                  }
76                  if (Params->contains((sLogin,?Login))) { }
77                  else Login := oNull;
78                  if (Params->contains((sPassword,?Password))) { }
79                  else Password := oNull;
80                  if (Sess->contains((sUserRights,?UserRights))) {
81                    Sess->remove((sUserRights,UserRights));
82                  }
83                  if (sDBAccessFunction(
                      select_member_id__member_level_from_members_where_member_login_EQ_
                      .Login._and_member_password_.Password,2)!=oNull) {
84                    Sess->contains((sUserRights,DBAccessFunction(
                        select_member_id__member_level_from_members_where_member_login_EQ_
                        .Login._and_member_password_.Password,2)));
85                  }
86                }
87                %PASSWORD AND USER NAME ARE MISSING
88                on (... %long condition% ...): {
89                   ...
90                }
91              }
92            }
93          }
94        }
95      }
96    ...
97  }
```

*Condition structure.* The automatically extracted model has complex conjunctions of conditions for each `select on (...)` branch. In comparison, the handwritten model initially tests only the JSP module and the form name, and does the remaining tests later, in `if` statements. This is because the automatically generated model successively accumulates conditions as the execution paths are traversed backwards, and individually identifies and treats each cycle-free execution path. Consequently, the handwritten model is more easily understandable as it closely reflects the step-by-step decision structure of the original application code. Moreover, the automatically generated code may exhibit duplication, e.g., of assignments appearing in a joint prefix of several execution paths.

To reduce complexity and code duplication, paths with identical assignments are grouped together, and their path conditions combined using disjunction (this optimization is partially implemented). It would be possible to extract common assignments between different groups of paths and move them up in the control flow, thus more closely reconstructing the original decision structure. This would be beneficial for human readability. However, this is only a matter of presentation and has no effect on the complexity

of the model (for analysis / model checking), since the translator to ASLan re-creates the same set of individual transitions as in the EFSM built in the first step (by iSummarize).

*Handling request parameters.* Both models represent the HTTP request (and the session state) as set of pairs of strings (parameter name and value). A frequent operation in the code is to extract and use the value of a given parameter (typically given as a constant string). However, the parameter may be absent and this case must always be treated. Therefore, in the automatically extracted model, each use of an extracted parameter value is guarded by a test of the form `Params->contains((paramName, ?ParamValue))` as seen in lines 25, 33, 66, etc. Currently, this test is inserted independently of context, without taking into account the path condition, which may make the test superfluous (or, on the contrary, unsatisfiable). Correlating these tests with the path condition and performing the resulting simplification does not raise conceptual difficulties and will be implemented, resulting in cleaner generated models.

*Database modeling.* In the handwritten model, the check for username and password is represented by a dedicated predicate, `validU()`, and likewise extracting the user ID and access rights from the corresponding database record (functions `getUID(Uname)` and `getURights(Uname)`. The representation in the automatically extracted model is closer to the JSP code, using a generic function `DBAccessFunction(sqlQuery, columnNo)` to model performing the query and selecting a given field from the result. Representing queries will become more expressive with the ongoing work on database modeling and interpreting a subset of SQL.

*Temporary variable names.* Automatically generated models may contain temporary variables (such as `_v53` in this model) that appear either due to the static single assignment form employed by the intermediate representation, as state needed during a loop, or to represent certain function results. Information linking them to the original code will need to be generated as an explanation for the user.

**Pros.** Two advantages, speed and accuracy, are direct consequences of automation. The model is produced in a few seconds, and is not prone to human error in understanding and capturing the behavior of the analyzed system. In particular, human errors may appear because a person is biased to infer and capture the intent of a piece of code, rather than the actual implementation of that intention in the code, as has been our experience with an initial version of the handwritten bookstore model. The benefit of accuracy should become more apparent once a complete analysis of exceptions

is implemented, as they are generally known both to complicate the control flow and as potential sources of flaws.

The automatically generated model can also be at times simpler than a handwritten one-to-one translation from the application source code, since the satisfiability checks on transitions can both identify infeasible paths and simplify superfluous tests.

**Cons.** The extracted model (Listing 13) is arguably more difficult to read by a human analyst, mostly due to the very complex condition formulas. This could be alleviated by extracting common parts of path conditions and common assignments in order to reconstruct the original code structure. Another possibility is to use comments that link parts of the model (e.g., conditions, sub-conditions, etc.) with the actual source code of the system.

Overall, it is clear that *automatic* procedures for extraction of ASLan++ models from code are indispensable for making automatic analysis extensively applicable. Combined with appropriate simplification procedures and support for traceability from source code to model, they can also aid a human in comprehending the behavior of the analyzed application.

# 6 Conclusions

We have presented the black box model inference method that makes it possible to derive a model of any component or subsystem of an application. This can be done with almost no knowledge about the component: its behavior will be inferred, and the set of its inputs and outputs can even be extracted automatically with the crawling approach presented in Section 4 when it has a web interface. The method is implemented in the SIMPA component of the SPaCIoS tool.

Although inferring such parametric models with security features is at the leading edge of research, it has been shown to work on case studies from the project, and the inferred model is comparable (modulo the restrictions on the semantic parts that are not accessible to inference) with the models that had been handwritten in ASLan++. The same applies to models extracted from source code with the jMODEX component of the SPaCIoS tool. Of course, the models produced automatically are less readable and lack the semantics and meaningful naming that are directly produced with handwritten models. But in both cases, the automated inference/extraction can be run very fast, and provides accurate models that can be refined if needed by a security analyst. Including such approaches in the SPaCIoS project and tool was a challenge and a bet, and the results so far are more than encouraging. At

the same time, the project also triggers incentives for further research, with new directions such as suggested in Section 3 of this deliverable.

# References

[1] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, A. da Silva Simão, and J. C. Maldonado, editors, *ICTSS*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[2] W. H. Allen, C. Dou, and G. A. Marin. A model-based approach to the security testing of network protocol implementations. In *LCN*, pages 1008–1015. IEEE Computer Society, 2006.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In J. Launchbury and J. C. Mitchell, editors, *POPL*, pages 4–16. ACM, 2002.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In V. Shmatikov, editor, *FMSE*, pages 1–10. ACM, 2008.

[6] AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial, 2011. Available at http://www.avantssar.eu.

[7] N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 211–221. ACM, 2011.

[8] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In J. L. Fiadeiro and P. Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008.

[9] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 141–150. ACM, 2009.

[10] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.

[11] M. Büchler, J. Oudinet, and A. Pretschner. Security mutants for property-based testing. In M. Gogolla and B. Wolff, editors, *TAP*, volume 6706 of *Lecture Notes in Computer Science*, pages 69–77. Springer, 2011.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Cheking*. MIT Press, 1999.

[13] O. Consortium. Security Assertion Markup Language V2.0 Technical Overview. http://wiki.oasis-open.org/security/Saml2TechOverview, Mar. 2008.

[14] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE*, pages 439–448, 2000.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[16] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *ICSE (1)*, pages 15–24. ACM, 2010.

[17] R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, editors, *TestCom/FATES*, volume 5047 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 2008.

[18] W. G. J. Halfond, S. R. Choudhary, and A. Orso. Penetration testing with improved input vector identification. In *ICST*, pages 346–355. IEEE Computer Society, 2009.

[19] K. Hossen, R. Groz, C. Oriat, and J.-L. Richier. Automatic generation of test drivers for model inference of web applications. In *Fourth International Workshop on Security Testing (SECTEST 2013), Workshop of the IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST2013)*, Luxembourg, March 2013. IEEE CS Press.

[20] M. N. Irfan, C. Oriat, and R. Groz. Angluin style finite state machine inference with non-optimal counterexamples. In *Proceedings of the First International Workshop on Model Inference In Testing*, MIIT '10, pages 11–19, New York, NY, USA, 2010. ACM.

[21] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2006.

[22] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In M. Hind and A. Diwan, editors, *PLDI*, pages 75–86. ACM, 2009.

[23] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[24] K. Meinke. CGE: A sequential learning algorithm for mealy automata. In J. M. Sempere and P. García, editors, *ICGI*, volume 6339 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010.

[25] A. Nerode. Linear automata transformation. In *American Mathematical Society*, volume 9, pages 541–544, 1958.

[26] O. Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, 2003.

[27] M. Oostdijk, V. Rusu, J. Tretmans, R. G. de Vries, and T. A. C. Willemse. Integrating verification, testing, and learning for cryptographic protocols. In J. Davies and J. Gibbons, editors, *IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 538–557. Springer, 2007.

[28] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[29] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999.

[30] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. Software Eng.*, 30(1):29–42, 2004.

[31] F. Ricca and P. Tonella. Analysis and testing of web applications. In H. A. Müller, M. J. Harrold, and W. Schäfer, editors, *ICSE*, pages 25–34. IEEE Computer Society, 2001.

[32] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In S. J. Hanson, W. Remmele, and R. L. Rivest, editors, *Machine Learning: From Theory to Applications*, volume 661 of *Lecture Notes in Computer Science*, pages 51–73. Springer, 1993.

[33] M. Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing*. PhD thesis, Institut Polytechnique de Grenoble, 2008.

[34] M. Shahbaz and R. Groz. Inferring mealy machines. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2009.

[35] G. Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *ICDCS*, page 25. IEEE Computer Society, 2007.

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

[37] SPaCIoS. Deliverable 2.2.2: Combined black-box and white-box model inference, 2013.

[38] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In M. Bernardo and V. Issarny, editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.

[39] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite automata, behavior and synthesis*. North-Holland Pub, 1973.

[40] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9:653–665, 1973.

[41] D. von Oheimb and S. Mödersheim. ASLan++ — a formal security specification language for distributed systems. In B. Aichernig, F. de Boer,

and M. Bonsangue, editors, *Formal Methods for Components and Objects, FMCO 2010, Graz, Austria*, volume 6957 of *LNCS*, pages 1–22. Springer, Dec. 2010. http://ddvo.net/papers/FMCO2010.html.

[42] G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. In *ICFEM*, pages 471–482, 2002.

[43] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)*. Morgan Kaufmann, 2011.

[44] E. Zenner. Nonce generators and the nonce reset problem. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *ISC*, volume 5735 of *Lecture Notes in Computer Science*, pages 411–426. Springer, 2009.

[45] M. Zulkernine, M. F. Raihan, and M. G. Uddin. Towards model-based automatic testing of attack scenarios. In B. Buth, G. Rabe, and T. Seyfarth, editors, *SAFECOMP*, volume 5775 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 2009.