Inferring models with rule-based expert systems.

William Durand, Sébastien Salva

Inferring models with rule-based expert systems.[*]

William Durand
LIMOS - UMR CNRS 6158
Blaise Pascal University, France
william.durand@isima.fr

Sébastien Salva
LIMOS - UMR CNRS 6158
Auvergne University, France
sebastien.salva@udamail.fr

ABSTRACT

Many works related to software engineering rely upon formal models, e.g., to perform model-checking or automatic test case generation. Nonetheless, producing such models is usually tedious and error-prone. Model inference is a research field helping in producing models by generating partial models from documentation or execution traces (observed action sequences). This paper presents a new model generation method combining model inference and expert systems. It appears that an engineer is able to recognise the functional behaviours of an application from its traces by applying deduction rules. We propose a framework, applied to Web applications, simulating this reasoning mechanism, with inference rules organised into layers. Each yields partial IOSTSs (Input Output Symbolic Transition Systems), which become more and more abstract and understandable.

Keywords

Model inference, automatic testing, IOSTS, expert system

1. INTRODUCTION AND CONTRIBUTION

Software engineering is a disciple helping to design, implement and validate applications by means of a lot of dedicated methods and tools. Many of them require either some documentation or models to automate or, at least, ease some steps. For instance, model-based testing approaches rely upon formal models to define test relation and to automatically construct test cases. Nonetheless, writing complete documentation or formal models is often a tedious and error-prone task. That is why lightweight models are usually found in the Industry. This leads to several issues, e.g., the toughness of validating an application with a good test coverage or the difficulty to diagnose its failures, and to maintain it since this one is poorly documented. The ultimate alternative, usually left to developers, is to learn how the application behaves before introducing changes on it.

Model inference is a research field which brings interesting concepts to bypass these issues. It aims at retrieving models, expressing functional behaviours of applications which already exist or which are under development. These models, that help understand how an application behaves, are generated from execution traces (observed action sequences) or from documentation. These can be exploited to automatically generate test cases, but could also be considered as drafts to write complete specifications. Although this area sounds promising, it still exposes several open problems, which require further investigation. Among them, the model generation may lead to a state space explosion problem. Some works construct lightweight models to avoid this issue, others yield extrapolated models by merging application's states which, of course, express more behaviours than those observed [10, 5]. Furthermore, most of these methods only works with event-driven applications, i.e. applications offering a Graphical User Interface (GUI) to interact with, and which respond to a sequence defined by the user. The other kind of applications are barely targeted.

Our proposal takes another direction to infer models. First, we do not suppose that the application being analysed is event-driven, only that it yields traces. Intuitively, our proposal emerges from the following idea: a human expert, who is able to conceive specifications, is also able to diagnose the behaviours of the corresponding implementation by reading and interpreting its execution traces. His knowledge could then be formalised and exploited to automatically infer models. Our approach is based upon this notion of knowledge implemented with an expert system which includes business rules. Such rules aim at analysing the behaviours of the application and incrementally produce models capturing the application's behaviours at a higher level of abstraction. These models are tangentially reduced in term of size without improper extrapolation. In this paper, we focus on models called Input/Output Symbolic Transition Systems (IOSTS) [8].

Paper organisation: below, we briefly present some related works and describe the architecture of our model generation framework. Then, we recall some definitions on the IOSTS formalism used throughout the paper. We concretely describe and define this framework in the context of Web applications in Section 3. We give some experimentation results in Section 4. Conclusions are drawn in Section 5 together with directions for further research and improvements.

[*]Research conducted in collaboration with industrial partner Michelin.

1.1 Related work

Model inference is a relatively recent research field which originates from works of different nature. Below, we present some of them related to our work.

Zong et al. [16] proposed to infer specifications from API documentation in order to check whether implementations match them. Such specifications do not reflect the implementation behaviour though. Furthermore, this method can only be applied if the API documentation is written in a readable format.

Most of the other methods aims at observing the application's behaviours at runtime. Some of them are proposed in the context of white-box testing. In [13], specifications, which are extremely detailed, show the method calls observed from a related set of objects. The methods, presented in [2, 4], exercise Mobile and Web applications written in PHP. They rely upon concolic testing to explore symbolic execution paths of the application and to detect bugs. These white-box approaches could theoretically offer a better code coverage than black-box automatic testing. However, the number of paths being explored concretely limits to short paths only. Furthermore, the constraints have not to be too complex in order to be solved. Last but not least, the models are too detailed for reading.

On the other hand, other methods [11, 12, 1, 6, 9, 15, 5], which originate from automatic black-box testing, retrieve specifications of event-driven applications (Desktop, Web or Mobile) by exploring them (a.k.a. crawling). In practice, the obtained models should encompass all the observed actions performed by the implementation. But, to avoid a state explosion problem and to ease the understanding of the application's behaviours, only the three main leads have been explored to reduce the model size. Some works [11, 1, 9] proposed to generate simplified trees, depicting the observed GUI of the application. Mesbah et al. [12] proposed the tool Crawljax specialised in Ajax applications. It produces a state machine model to capture the changes of the DOM structures of HTML documents. But here, state abstractions have to be manually given. Also, some works [5] rely upon standard learning algorithms such as L^* [3], but this technique leads to extrapolated models, potentially describing incorrect behaviours.

1.2 Insight

Our proposal takes another direction by inferring several models (and not only one), expressing the behaviours of the same application at different abstraction levels by leveraging an expert system.

We focus on Web applications in this paper, altough this approach could be applied on any application producing traces. Thanks to its design, our framework supports any kind of Web applications, including Single Page Applications.

The approach is divided into several modules as depicted in Figure 1. The *Models generator* is the centrepiece of this framework. It takes traces as inputs, which can be sent by a *Monitor* collecting them on the fly. But it is worth mentioning that the traces can also be sent by any tool or even any user, as far as they comply to a chosen standard format. The Models generator is based upon an expert system, which is an artificial intelligence engine emulating acts of a human expert by inferring a set of rules representing his knowledge. Such knowledge are organised into a hierarchy of several layers. Each gathers a set of inference rules written with a first order predicate logic. Typically, each layer creates an IOSTS, and the higher the layer is, the more abstract the IOSTS becomes. Models are then stored and can be later analysed by experts, verification tools, etc. The number of layers is not strictly bounded even though it is manifest that it has to be finite.

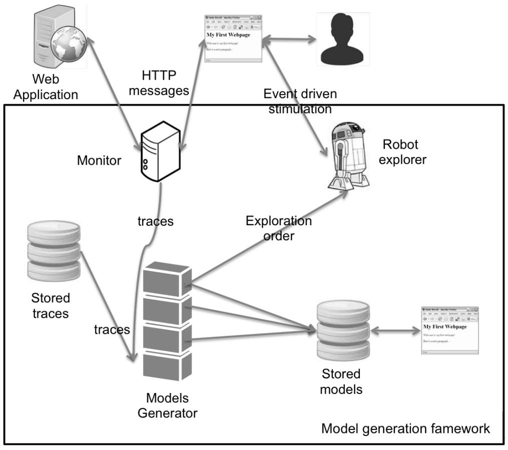

Figure 1: Model generation framework

The Models generator relies upon traces to construct IOSTSs, but the given trace set may not be substantial enough to generate relevant IOSTSs. More traces could be yet collected as far as the application being analysed is an event-driven application. Such traces can be produced by stimulating and exploring the application with automatic testing. In our approach, this exploration is achieved by the *Robot explorer*. In contrast with most of the existing crawling techniques [11, 2, 12, 1, 15], our robot does not cover the application in blind mode or with a static traversal strategy. Instead, it is cleverly guided by the Models generator which applies an exploration strategy carried out by rules. This involves the capture of new traces by the Monitor or by the Robot explorer which returns them to the Models generator, and so on, as described in [14]. The advantages of this approach are manifold:

- it takes a predefined set of traces collected from any kind of applications producing traces. In the context of Web applications, traces can be produced using automatic testing,
- the application exploration is guided with a strategy which can be modified according to the type of application being analysed. This strategy offers the advantage of directly targeting some states of the application when its state number is too large for being traversed in a reasonable processing time,
- the knowledge encapsulated in the expert system can

be used to cover trace sets of several applications belonging to the same category with generic rules,

- but, the rules can also be specialised and refined for one application to yield more precise models. This is interesting for application comprehension,
- our approach is both flexible and scalable. It does not produce one model but several ones, depending on the number of layers of the Models generator, which is not limited and may evolve in accordance to the application's type. Each model, expressing the application's behaviours at a different level of abstraction, can be used to ease the writing of complete formal models, to apply verification techniques, to check the satisfiability of properties, to automatically generate functional test cases, etc.

In the following, we focus on the Models generator in the context of Web applications, this part being the centrepiece of our framework. It is worth mentioning that the Monitor is here a classical HTTP proxy, hence the support of any kind of Web applications, and generally speaking, any system producing traces.

2. MODEL DEFINITION AND NOTATIONS

We shall consider the input/output Symbolic Transition System (IOSTS) formalism [8] for describing the functional behaviour of systems or applications. An IOSTS is a kind of automata model which is extended with two sets of variables, internal variable to store data, and parameters to enrich the actions. Transitions carry actions, guards, and assignments over variables. The action set is separated with inputs beginning by ? to express actions expected by the system, and with outputs beginning by ! to express actions produced by the system. An IOSTS does not have states but locations.

Definition 1 (IOSTS) *An IOSTS $\mathcal{S}$ is a tuple $< L, l0, V, V0, I, \Lambda, \rightarrow>$, where:*

- *L is the finite set of locations, $l0$ the initial location,*
- *V is the finite set of internal variables, I is the finite set of parameters. We denote D_v the domain in which a variable v takes values. The assignment of values of a set of variables $Y \subseteq V \cup I$ is denoted by valuations where a valuation is a function $v : Y \rightarrow D$. $v_\emptyset$ denotes the empty valuation. D_Y stands for the valuation set over the variable set Y. The internal variables are initialised with the assignment $V0$ on V, which is assumed to be unique,*
- *Λ is the finite set of symbolic actions $a(p)$, with $p = (p_1, ..., p_k)$ a finite list of parameters in $I^k (k \in \mathbb{N})$. p is assumed unique. $\Lambda = \Lambda^I \cup \Lambda^O \cup \{!\delta\}$: Λ^I represents the set of input actions, (Λ^O) the set of output actions,*
- *$\rightarrow$ is the finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \in L$ to $l_j \in L$, denoted $l_i \xrightarrow{a(p),G,A} l_j$ is labelled by: an action $a(p) \in \Lambda$, a guard G over $(p \cup V \cup T(p \cup V))$ which restricts the firing of the transition. $T(p \cup V)$ is a set of functions that return boolean values only (a.k.a. predicates) over $p \cup V$, an assignment function A which updates internal variables. A is on of the form $(x := A_x)_{x \in V}$, where A_x is an expression over $V \cup p \cup T(p \cup V)$.*

An IOSTS is also associated with an IOLTS (Input/Output Labelled Transition System) to formulate its semantics. Intuitively, IOLTS semantics correspond to valued automata without symbolic variable, which are often infinite: IOLTS states are labelled by internal variable valuations while transitions are labelled by actions and parameter valuations. The semantics of an IOSTS $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow>$ is the IOLTS $[\![\mathcal{S}]\!] =< Q, q_0, \Sigma, \rightarrow>$ composed of valued states in $Q = L \times D_V$, $q_0 = (l_0, V_0)$ is the initial one, Σ is the set of valued symbols and $\rightarrow$ is the transition relation. The IOLTS semantics definition of can be found in [8]. In short, for an IOSTS transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain an IOLTS transition $(l_1, v) \xrightarrow{a(p),\theta} (l_2, v')$ with v a set of valuations over the internal variable set, if there exists a parameter valuation set θ such that the guard G evaluates to true with $v \cup \theta$. Once the transition is executed, the internal variables are assigned with v' derived from the assignment $A(v \cup \theta)$. Runs and traces of an IOSTS can now be defined from its semantics:

Definition 2 (Runs and traces) *For an IOSTS $\mathcal{S} = < L, l0, V, V0, I, \Lambda, \rightarrow>$, interpreted by its IOLTS semantics $[\![\mathcal{S}]\!] =< Q, q_0, \Sigma, \rightarrow>$, a run of $\mathcal{S}$, $q_0\alpha_0 q_1 ... q_{n-1}\alpha_{n-1}q_n$ is a sequence of terms $q_i\alpha_i q_{i+1}$ with $\alpha_i \in \Sigma$ a valued action and q_i, q_{i+1} two states of Q. $Run(\mathcal{S}) = Run([\![\mathcal{S}]\!])$ is the set of runs found in $[\![\mathcal{S}]\!]$.*

It follows that a trace of a run r is defined as the projection $proj_\Sigma(r)$ on actions. $Traces_F(\mathcal{S}) = Traces_F([\![\mathcal{S}]\!])$ is the set of traces of all runs finished by states in $F \times D_V$.

3. MODEL INFERENCE

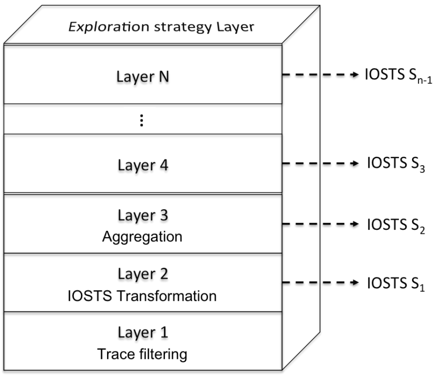

Figure 2: Models generator

The Models generator is mainly composed of a rule-based system, adopting a forward chaining. Such a system separates the knowledge base from the reasoning: the former is expressed with data a.k.a. facts and the latter is realised with inference rules that are applied on the facts. Our Models generator initially takes traces as an initial knowledge base and owns inference rules organised into layers for trying to match the human expert behaviour. These layers are depicted in Figure 2.

Usually, when a human expert has to read traces of an application, he often filters them out to only keep those which make sense against the current application. This step is done by the first layer whose role is to format the received raw traces into sequences of valued actions and to delete those considered as unnecessary. The resulting structured trace set, denoted ST, is then given to the next layer. This process is incrementally done, i.e. every time new traces are given to the Models generator, these are formatted and filtered before being given to Layer 2. The remaining layers yield an IOSTS each $\mathcal{S}_i (i \geq 1)$, which has a tree structure derived from the traces. The role of Layer 2 is to carry out a first IOSTS transformation from the structured traces. The next layers 3 to N (with N a finite integer) are composed of rules that emulate the ability of a human expert to simplify transitions, to analyse the transition syntax for deducing its meaning in connection with the application, and to construct more abstract actions that aggregate a set of initial ones. Theses deductions are often not done in one step. This is why the Models generator supports a finite but not defined number of layers. Each of these layers i takes the IOSTS $\mathcal{S}_{i-1}$ given by the direct lower layer. This IOSTS, which represents the current base of facts, is analysed by the rules to infer another IOSTS whose expressiveness is more abstract than the previous one. We state that the lowest layers (at least Layer 3) should be composed of generic rules that can be reused on several applications of the same type. In contrast, the highest layers should own the most precise rules that may be dedicated to one specific application.

For readability purpose, we chose to represent inference rules with this format: *When conditions on facts Then actions on facts* (format taken by the Drools inference engine [1]). Independently on the application type, the Layers 2 to N handle the following fact types: *Location* which represents an IOSTS location, and *Transition*, which represents an IOSTS transition, composed of two Locations Linit, Lfinal, and two data collections Guard and Assign. Now, it is manifest that the inference of models has to be done in a finite time and in a deterministic way. To reach that purpose, we formulate the following hypotheses on the inference rules:

1. (finite complexity): a rule can only be applied a limited number of times on the same knowledge base,
2. (soundness): the inference rules are Modus Ponens (simple implications that lead to sound facts if the original facts are true),
3. (no implicit knowledge elimination): after the application of a rule r expressed by the relation $r : T_i \rightarrow T_{i+1} (i \geq 2)$, with T_i a Transition base, for all transition $t = (l_n, l_m, a(p), G, A)$ extracted from T_{i+1}, l_n is reachable from l_0.

In the following, we detail these layers in the context of Web applications while giving some rule examples.

3.1 Layer 1: Trace filtering

Traces of Web applications are based upon the HTTP protocol, conceived in such a way that each HTTP request is followed by only one HTTP response. Consequently, the traces, given to Layer 1, are sequences of couples (HTTP request, HTTP response). This layer begins formatting these couples so that these might be analysed in a more convenient way.

For a couple (HTTP request, HTTP response), we extract the following information: the HTTP verb, the target URI, the request content which is a collection of data (headers, content) and the response content which is the collection (HTTP status, headers, response content). An header may also be a collection of data or may be null. Contents are texts e.g., HTML texts. Since we wish translating such traces into IOSTSs, we turn these textual items into a structured valued action $(a(p), \theta)$ with a the HTTP verb and θ a valuation over the variable set $p = \{URI, request, response\}$. This is captured by the following proposition:

Definition 3 (Structured HTTP Traces) *Let $t = req_1$, $resp_1$, ..., req_n, $resp_n$ be a raw HTTP trace composed of an alternate sequence of HTTP request req_i and HTTP response $resp_i$. The structured HTTP trace σ of t is the sequence $(a_1(p), \theta_1)...(a_n(p), \theta_n)$ where:*

- *a_i is the HTTP verb used to make the request in req_i,*
- *p is the parameter set $\{URI, request, response\}$,*
- *θ_i is a valuation $p \rightarrow D_p$ which assigns a value to each variables of p. θ is deduced from the values extracted from req_i and $resp_i$.*

The resulting trace set derived from raw HTTP traces is denoted ST.

Now, the structured traces can be filtered. For a main request performed by a user, many other sub-requests are also launched by a browser in order to fetch images, CSS and JavaScript files. Generally speaking, these do not enlighten a peculiar functional behaviour of the application. This is why we propose to add rules in Layer 1 to filter these sub-requests out from the traces. Such sub-requests can be identified by different ways, e.g., by focussing on the file extension found at the end of the URI, or on the Content-type value of the request headers. Consequently, we created a set of rules, constituted of conditions on the HTTP content found in an action, that remove valued actions when the condition is met. A straightforward rule example, which removes the actions relative to the retrieval of PNG images, is given in Figure 3.

After the instantiation of the Layer 1 rules, we obtain a formatted and filtered trace set ST composed of valued actions. Now, we are ready to extract the first IOSTSs.

Completeness, soundness, complexity: HTTP traces are sequences of valued actions modelled with positive facts. Typically, they form Horn clauses. Furthermore, inference

[1] http://www.jboss.org/drools/

```
rule "Filter PNG images"
when
    \$va: Get(request.mime_type = ’png’ or
    request.file_extension = ’png’)
then
    retract(\$va);
end
```

Figure 3: Filtering rule example

rules are Modus Ponens (soundness hypothesis). Consequently, Layer 1 is sound and complete. Keeping in mind the (finite complexity) hypothesis, its complexity is proportional to $\mathcal{O}m(k+1)$ with m the valued action number and k the rule number. (at worst, every action is covered $k+1$ times).

3.2 Layer 2: IOSTS transformation

Intuitively, the IOSTS transformation relies upon the IOLTS semantics transformation that is achieved in a backward manner. In order to generate the first IOSTS denoted $\mathcal{S}_1$, the associated runs are first computed from the structured traces by injecting states between valued actions.

These steps are detailed below:

3.2.1 Traces to runs

Given a trace σ, a run r is firstly derived by constructing and injecting states on the right and left sides of each valued action of σ. Keeping in mind the IOLTS semantics definition, a state shall be modelled by the couple $((URI, k), v_\emptyset)$ with $v_\emptyset$ the empty valuation. (URI, k) is a couple composed of a URI and of an integer ($k \geq 0$). Typically, a couple (URI, k) shall be a location of the future IOSTS. All the runs r of SR start with the same state $(l0, v_\emptyset)$. Then, a run is constructed by incrementally covering one trace: for an action actions (a_i, θ_i) found in a trace, we extract the valuation $URI = val$ from θ_i giving the URI value of the next resource reached after the action a_i.And we complete the current run r with (a_i, θ_i) followed by the state $((val, k), v_\emptyset)$. Since we wish to preserve the sequential order of the actions found in the traces, when a URI previously encountered is once more detected, the resulting state is composed of the URI accompanied with an integer k, which is incremented to yield a new and unique state. Due to lack of room, the algorithm translating the structured traces into a run set is not provided in this paper but can be found in [14].

3.2.2 IOSTS generation

The first IOSTS $\mathcal{S}_1$ is derived from the run set SR in which runs are disjoint except for the initial state $(l0, v_\emptyset)$. Intuitively, traces are translated into IOSTS paths that are assembled together (IOSTS disjoint union). The IOSTS forms a tree composed of paths, each expressing one trace, starting from the same initial location.

Definition 4 *Given a run set SR, the IOSTS $\mathcal{S}_1$ is called the IOSTS tree of SR and corresponds to the tuple $< L_{\mathcal{S}_1}, l0_{\mathcal{S}_1}, V_{\mathcal{S}_1}, V0_{\mathcal{S}_1}, I_{\mathcal{S}_1}, \Lambda_{\mathcal{S}_1}, \rightarrow_{\mathcal{S}_1} >$ such that:*

- $L_{\mathcal{S}_1} = \{l_i \mid \exists r \in SR, (li, v_\emptyset)$ *is a state found in* $r\}$,
- $l0_{\mathcal{S}_1}$ *is the initial location such that* $\forall r \in SR$, *r starts with* $(l0_{\mathcal{S}_1}, v_\emptyset)$,
- $V_{\mathcal{S}_1} = \emptyset$, $V0_{\mathcal{S}_1} = v_\emptyset$,
- $\Lambda_{\mathcal{S}_1} = \{a_i(p) \mid \exists r \in SR, (a_i(p), \theta_i)$ *is a valued action in* $r\}$,
- $\rightarrow_{\mathcal{S}_1}$ *is defined by the following inference rule applied on every element* $r \in SR$:

$$\begin{array}{l} s_i(a_i(p), \theta_i)s_{i+1} \textit{ is a term of } r, s_i = (l_i, v_\emptyset), \\ s_{i+1} = (l_{i+1}, v_\emptyset), G_i = \bigwedge_{(x_i = vi) \in \theta_i} x_i == vi \\ \vdash \\ l_i \xrightarrow{a_i(p), G_i, (x:=x)_{x \in V}}_{\mathcal{S}_1} l_{i+1} \end{array}$$

Here, locations could be merged to reduce the IOSTS size with the classical learning algorithms based upon L^* [3, 10]. Nonetheless, these would create an extrapolation of this IOSTS. We prefer rejecting such a solution to preserve the trace equivalence of the IOSTS $\mathcal{S}_1$ against the structured trace set ST before applying inference rules. Instead, we propose to use a minimisation technique.

3.2.3 IOSTS minimisation

This IOSTS tree can be reduced in term of location size by applying a bisimulation minimisation technique which still preserves the functional behaviours expressed in the original model. Intuitively, this minimisation constructs the state sets (blocks) that are bisimilar equivalent. Two states are said bisimilar equivalent, denoted $q \sim q'$ iff they simulate each other and go to states from where they can simulate each other again. Due to lack of room, we only refer to the bisimulation minimisation algorithm of [7].

When receiving new traces from the Monitor, the model yield by this layer is not fully regenerated, but rather completed on the fly. New traces are translated into IOSTS paths that are disjoint from $\mathcal{S}_1$ except from the initial location. We perform an union between $\mathcal{S}_1$ and IOSTS paths. Then, the resulting IOSTS is minimised.

Completeness, soundness, complexity: Layer 2 takes any structured trace set obtained from HTTP traces. If the trace set is empty then the resulting IOSTS $\mathcal{S}_1$ has a single location l_0. A structured trace set is translated into an IOSTS in finite time: every valued action of a trace is covered once to construct states, then every run is lifted to the level of one IOSTS path starting from the initial location. Afterwards, the IOSTS is minimised with the algorithm presented in [7]. Its complexity is proportional to $\mathcal{O}(mlog(m+1))$ with m the number of valued actions. The soundness of Layer 2 is based upon the notion of traces: an IOSTS $\mathcal{S}_1$ is composed of transition sequences derived from runs in SR, itself obtained from the structured trace set ST. As defined, the behaviours encoded in ST and $\mathcal{S}_1$ are equivalent since ordered runs are transformed into ordered IOSTS sequences.

For sake of readability, we do not provide here the rules of

Layer 2, which match the above definitions and algorithms. Instead, we illustrate an IOSTS generation example below:

Example 3.1 We take as example a trace obtained from the GitHub Web site [2] after having executed the following actions: login with an existing account, choose an existing project, and logout. These few actions already produced a large set of requests and responses. Indeed, a web browser sends thirty HTTP requests on average in order to display a GitHub page. The trace filtering form this example returns the following structured traces where the request and response parts are concealed for readability purpose:

```
GET(https://github.com/)
GET(https://github.com/login)
POST(https://github.com/session)
GET(https://github.com/)
GET(https://github.com/willdurand)
GET(https://github.com/willdurand/Geocoder)
POST(https://github.com/logout)
GET(https://github.com/)
```

After the application of Layer 2, we obtain the IOSTS of Figure 4. Locations are labelled by the URI found in the request and by an integer to keep the tree structure of the initial traces. Actions are composed of the HTTP verb enriched with the variables URI, request, and response. This IOSTS exactly reflects the trace behaviour but is still difficult to interpret. More abstract actions shall be deduced by the next layers.

3.3 Layers 3-N: IOSTS abstraction

As stated earlier, the rules of the upper layers analyse the transitions of the current IOSTS for trying to enrich its semantics while reducing its size. Given an IOSTS $\mathcal{S}_1$, every next layer carries out the following steps:
1. apply the rules of the layer and infer a new knowledge base (new IOSTS $\mathcal{S}_i$, $i \geq 2$),
2. apply a bisimulation minimisation,
3. store the resulting IOSTS.

Without loss of generality, we now restrict the rule structure to keep a link between the generated IOSTSs. Thereby, every rule of Layer i ($i \geq 3$) either enriches the sense of the actions (transition per transition) or aggregates transition sequences into one unique new transition to make the obtained IOSTSs more abstract. It results in an IOSTS $\mathcal{S}_i$ exclusively composed by some locations of the first IOSTS $\mathcal{S}_1$. Consequently, for a transition or path of $\mathcal{S}_i$, we can still retrieve the concrete path of $\mathcal{S}_1$. This is captured by the following proposition:

Proposition 5 *Let $\mathcal{S}_1$ be the first IOSTS generated from the structured trace set ST. The IOSTS $\mathcal{S}_i(i > 1)$ produced by Layer i has a location set $L_{\mathcal{S}_i}$ such that $L_{\mathcal{S}_i} \subseteq L_{\mathcal{S}_1}$.*

Completeness, soundness, complexity: the knowledge base is exclusively constituted by (positive) Transition facts that have an Horn form. The rules of these layers are Modus Ponens (soundness hypothesis). Therefore, these inference rules are sound and complete. Furthermore, a behaviour encoded in an IOSTS $\mathcal{S}_i$ cannot be lost in $\mathcal{S}_i$. With regards to the (no implicit knowledge elimination) hypothesis and to Proposition 5, the transitions of $\mathcal{S}_i$ are either unchanged, enriched or combined together into a new transition. The application of these layers ends in a finite time ((finite complexity) hypothesis) and the complexity of each is proportional to $\mathcal{O}m(k)$ with m the transition number and k the rule number.

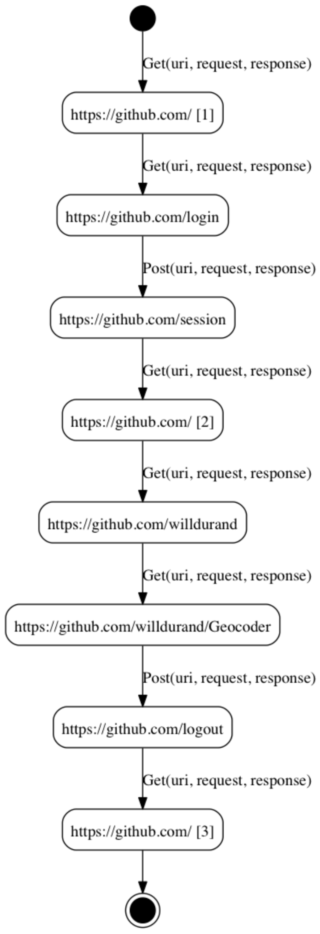
Get(uri, request, response)
https://github.com/ [1]
Get(uri, request, response)
https://github.com/login
Post(uri, request, response)
https://github.com/session
Get(uri, request, response)
https://github.com/ [2]
Get(uri, request, response)
https://github.com/willdurand
Get(uri, request, response)
https://github.com/willdurand/Geocoder
Post(uri, request, response)
https://github.com/logout
Get(uri, request, response)
https://github.com/ [3]

Figure 4: IOSTS $\mathcal{S}_1$

In the following, we detail two layers specialised for Web applications:

3.3.1 Layer 3

As stated in Section 1.2, Layer 3 should correspond to a set of generic rules that can be applied on a large set of applications belonging to the same category. This layer has two roles:

- the enrichment of the meaning captured in transitions. In this step, we have chosen to mark the transitions with new internal variables. These shall help deduce more abstract actions in the upper layers. For example, the rules depicted in Figure 5 aims at recognising the receipt of a login or logout page. The first rule

[2] https://github.com/

means that if the response content, which is received after a request sent with the *GET* method, contains a login form, then this transition is marked as a "login page" with the assignment on the variable isLoginPage,

- the generic aggregation of some successive transitions. Here, some transitions (two or more) are analysed in the conditional part of the rule. When the rule condition is met then the successive transitions are replaced by one transition carrying a new action. The rule of Figure 6 corresponds to a simple transition aggregation. It aims at recognising the successive sending of information with a POST request followed by a redirection to another Web page. If a request sent with the *POST* method has a response identified as a redirection, (identified by the status code 301 or 302), and a *GET* request comes after, both transitions are reduced into a single one carrying the new action *PostRedirection*.

```
rule "Identify Login Page"
when
    $t: Transition(Action == GET, Guard.
      response.content contains('login-form'))
then
    modify ($t) { Assign.add("isLoginPage:=true") }
end

rule "Identify Logout Request"
when
    $t: Transition(Action == GET, Guard.
      uri matches("/logout"))
then
    modify ($t1) { Assign.add("isLogout:=true") }
end
```

Figure 5: Login and Logout page recognition rules

```
rule "Identify Redirection after a Post"
when
    $t1: Transition(Action == POST and
       (Guard.response.status = 301 or Guard.response.
       status = 302) and $l1final := Lfinal)
    $t2: Transition(Action == GET, linit == $l1final,
       $l2linit:=Linit)
    not (Transition (Linit == $l2linit))
then
    insert(new Transition("PostRedirection", Guard(
       $t1.Guard, $t2.Guard), Assign($t1.Assign,
       $t2.Assign), $t1.Linit, $t2.Lfinal );
    retract($t1);
    retract($t2);
end
```

Figure 6: Simple aggregation

Example 3.2 When we apply these rules on the IOSTS example of Figure 4, we obtain a new IOSTS illustrated in Figure 7. Its size is reduced since it has 6 transitions instead of 9 previously. However, this new IOSTS does not reflect clearly the initial scenario yet. Rules deducing more abstract actions are required. These are found in the next layer.

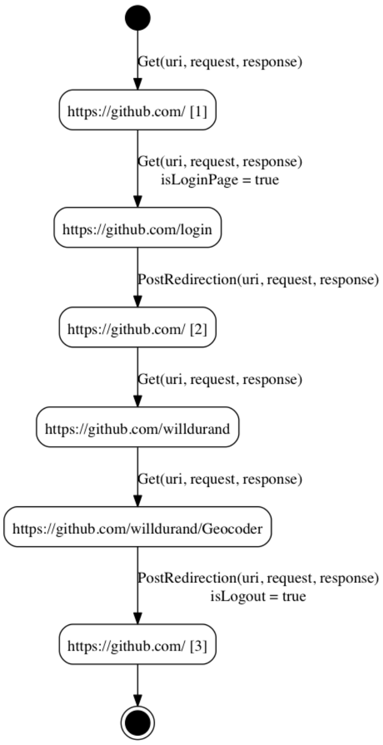
Get(uri, request, response)
https://github.com/ [1]
Get(uri, request, response)
isLoginPage = true
https://github.com/login
PostRedirection(uri, request, response)
https://github.com/ [2]
Get(uri, request, response)
https://github.com/willdurand
Get(uri, request, response)
https://github.com/willdurand/Geocoder
PostRedirection(uri, request, response)
isLogout = true
https://github.com/ [3]

Figure 7: IOSTS $\mathcal{S}_2$

3.3.2 Layer 4

This layer aims to infer a more abstract model composed of more expressive actions and whose size should be reduced. Its rules may have different forms:

- they can be applied on one transition only. In this case, the rule replaces the transition action to add more sense to the action. The rule of Figure 8 is an example which recognises a user de-authentication and adds a new action "Deauthentication". This rule means that if a *PostRedirection* action is triggered against a "Logout" endpoint (given by the variable *isLogout* added by Layer 3), then this is a deauthentication,
- the rules can also aggregate several successive transitions up to complete paths into one transition labelled by a more abstract action. For instance, the rule illustrated in Figure 9 recognises a user authentication thanks to the variable "isLoginPage" added by Layer 3. This rule means that if a "Login" page is displayed, followed by a redirection triggered by a *POST* request, then this is an authentication step, and the two transitions are reduced into a single one composed of the action "Authentication".

Other rules can also be application-specific, so that these bring specific new knowledge to the model. For instance, the GitHub Web application has a dedicated URL grammar (a.k.a. routing system). GitHub users own a profile page that is available at: *https://github.com/*{*username*} where {*username*} is the nickname of the user. However, some

```
rule "Identify Deauthentication"
when
    $t: Transition(action == PostRedirection,
       Assign contains "isLogout:=true")
then
    modify ($t) (setAction "Deauthentication"));
end
```

Figure 8: Deauthentication recognition rule

```
rule "Identify Authentication"
when
    $t1: Transition(Action == GET,
       Assign contains "isLoginPage:= true",
    $t1final:=Lfinal)
    $t2: Transition(Action == PostRedirection,
       Linit == $t1lfinal, $t2linit:=Linit)
    not (Transition (Linit == $t2linit))
then
    insert(new Transition("Authentication",
       Guard($t1.Guard,$t2.Guard), Assign($t1.Assign,
       $t2.Assign), $t1.Linit, $t2.Lfinal );
    retract($t1);
    retract($t2);
end
```

Figure 9: Authentication recognition

items are reserved e.g., *edu* and *explore*. The rule given in Figure 10 is based upon this structure and produces a new action ”Showprofile” offering more sense. Similarly, a GitHub page describing a project has a URL that always matches the pattern: *https://github.com/*{*username*}/{*project_name*}. The rule of Figure 11 captures this pattern and derives a new action named ”ShowProject”.

```
rule "GitHub profile pages"
when
    $t: Transition(action == GET, (
       Guard.uri matches "/[a-zA-Z0-9]+$",
       Guard.uri not in [ "/edu", "/explore" ]))
then
    modify ($t) (SetAction("Showprofile"));
end
```

Figure 10: User profile recognition

Example 3.3 The application of the four previous rules leads to the final IOSTS depicted in Figure 12. Now, it can be used for application comprehension since most of its actions have a precise meaning and clearly describe the application behaviour.

3.4 Strategy layer

Rather than using a static traversal strategy as in [11, 2, 12, 1, 15], we propose the addition of an orthogonal layer in the Models generator to describe any kind of exploration strategy by means of rules.

The simplified Algorithm of the Strategy layer is given in Algorithm 1. The latter applies the rules on any stored IOSTS $\mathcal{S}_i$. It emerges a location list *Loc* that are marked with ”explored” by the rules to avoid re-using them twice (line 4).

```
rule "GitHub project pages"
when
    $t: Transition(action == GET,
       Guard.uri matches "/[a-zA-Z0-9]+/.+$" $uri:=Guard.uri)
then
    String s=ParseProjectName($uri);
    modify ($t) (SetAction("Showproject")
      Assign.add("ProjectName:="+s) );
end
```

Figure 11: Project choice recognition

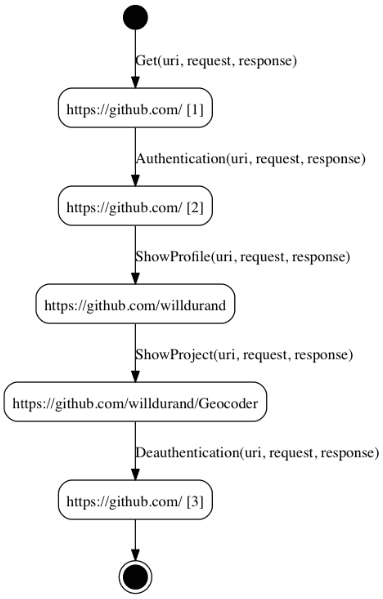
Get(uri, request, response)
https://github.com/ [1]
Authentication(uri, request, response)
https://github.com/ [2]
ShowProfile(uri, request, response)
https://github.com/willdurand
ShowProject(uri, request, response)
https://github.com/willdurand/Geocoder
Deauthentication(uri, request, response)
https://github.com/ [3]

Figure 12: IOSTS $\mathcal{S}_3$

Then, the algorithm goes back to the first generated IOSTS $\mathcal{S}_1$ in order to extract one complete and executable path p ended by a location l of *Loc* (line 7). This step is sound since all the locations of $\mathcal{S}_i$ belong to the locations set of $\mathcal{S}_1$ (Proposition 5). Such an IOSTS preamble is required by the Robot explorer for trying to reach the location l by executing every action of p. The algorithm finally returns a list of paths $List$, which is sent to the Robot explorer. The exploration ends once all the locations of $\mathcal{S}_i$ or of $\mathcal{S}_1$ are visited (line 3). The algorithm only returns unexplored locations even if, while the execution of the algorithm, the IOSTS $\mathcal{S}_i$ has been regenerated several times since the marked locations are also stored in the set L. Hence, if a location of $\mathcal{S}_i$ is chosen a second time by the rules, the algorithm checks if it has been previously visited (line 7).

The rules of the Strategy layer can encode different strategies. We propose two examples below:

- classical traversal strategies can still be established. For example, Figure 13 depicts two rules expressing the choice the next location to explore in a breadth-wise order first. First, the initial location $l0$ is chosen and marked as explored (rule BFS). Then, the transitions having an initial location marked as explored and a

Algorithm 1: Exploration Strategy

```
input  : IOSTS $\mathcal{S}_1$, $\mathcal{S}_i$
output: List of preambles
$L := \emptyset$ List of explored locations of $\mathcal{S}_1$;
BEGIN;
while $L \neq L_{\mathcal{S}_1}$ $and$ $L \neq \mathcal{S}_i$ do
    1) Apply the rules on $\mathcal{S}_i$ and extract a Location List $Loc$;
    Goback to $\mathcal{S}_1$;
    foreach $l \in Loc$ do
        if $l \notin L$ then
            Compute a preamble $p$ from $l0_{\mathcal{S}_1}$ which reaches $l$;
            $L := L \cup \{l\}$;
            $List := List \cup \{p\}$;
END;
```

final location not yet explored are collected by the rule BFS2, except for the transitions carrying an HTTP error (response status upper or equal to 400). These locations are marked as explored in the IOSTS $\mathcal{S}_i$ with the method *Setexplored* in the "then" part of the rule,

- a semantic-driven strategy could also be applied, when the meaning of some actions is recognisable. For instance, for e-commerce applications, the login step and the term "buy" are usually important. Thereby, a strategy targeting firstly the locations of transitions carrying theses actions can be defined by the rule "semantic-driven strategy" of Figure 14. It is manifest that the semantic-driven strategy domain can be tremendously vast since it depends on the number of recognised actions and on their relevance.

```
rule "BFS"
when
  $l:  Location (name == l0, explored == false)
then
    modify ($l) ( explored=true );
end

rule "BFS2"
when
    $Loc : ArrayList<Location> () from accumulate(
    $t : Transition ( Guard.response.status >199 &&
       Guard.response.status <400 && Linit.explored==
       true && Lfinal.explore==false  ),
     init( ArrayList<Transition> Loc=new ArrayList<
       Transition>(); ),
     action( Loc.add( $t.Lfinal ); ),
     result( Loc ) );
then
    Loc.Setexplored();
end
```

Figure 13: BFS strategy

4. EXPERIMENTATION

The framework presented in Section 1.2 has been implemented in a prototype tool called *Autofunk* (Automatic Functional model inference). A user interacts with Autofunk through a Web interface and either gives a URL or a file containing traces. These have to be stored in the HTTP Archive (HAR) format as it is the defacto standard to describe HTTP traces, used by various HTTP related tools. Such traces can be obtain from many HTTP monitoring tools (Mozilla Firefox or Google Chrome included). Then, Autofunk produces IOSTS models which are stored in a database. The last model is depicted in the Web interface. The JBoss Drools Expert tool has been chosen to implement the rule-based system. Such an engine leverages Oriented Object Programming in the rule statements and takes knowledge bases given as Java objects (Location, Transition, GET, POST objects in this work).

```
rule "semantic-driven strategy"
when
    $t: Transition (Assign contains
    "isLogin:=true" || Guard.response matches "*buy*")
then
    ArrayList Loc = new ArrayList();
    Loc.add($t.Linit, $t.Lfinal );
    Loc.Setexplored();
end
```

Figure 14: Semantic-driven strategy

The GitHub Web site is an example of application giving significant results. We recorded a trace set composed of 840 HTTP requests / responses. Then, we applied Autofunk on them with a Models generator composed of 5 layers gathering 18 rules whose 3 are specialised to GitHub. After having performed trace filtering (Layer 1), we obtained a first IOSTS tree composed of 28 transitions. The next 4 layers automatically infer a last IOSTS tree $\mathcal{S}_4$ composed of 12 transitions whose 9 have a clear and intelligible meaning.

5. CONCLUSION

This paper presents an original approach combining model inference, expert systems and automatic testing to derive IOSTSs models. Our proposal yields several models, reflecting different levels of abstractions of the same application with the use of inference rules that capture the knowledge of an expert. The first contribution lies in the flexibility and scalability brought by the inference rules since they can be applied on several applications or on one application only when the rules are specific. The whole framework has not to be re-implemented for each application. Our approach can be applied on event-driven applications since our framework supports their exploration. Furthermore, it can also be applied on other application types on condition that these produce traces.

We designed our framework for Web applications as a premise. In the future, we intend to apply it on industrial systems to ease their diagnostics. But this kind of system brings several issues not yet addressed in the model inference area. For instance, industrial systems may include asynchronous actions and timed properties. At the moment, our solution does not yet support this kind of properties. Furthermore, writing rules may be as tough as writing models in some cases. This is why we are working on a human interface which helps design rules from a trace set example. We also plan to add a test case generation module for regression testing.

6. REFERENCES

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping

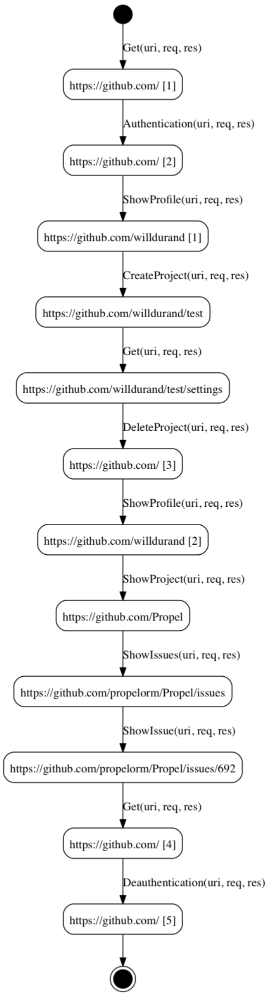
Get(uri, req, res)
https://github.com/ [1]
Authentication(uri, req, res)
https://github.com/ [2]
ShowProfile(uri, req, res)
https://github.com/willdurand [1]
CreateProject(uri, req, res)
https://github.com/willdurand/test
Get(uri, req, res)
https://github.com/willdurand/test/settings
DeleteProject(uri, req, res)
https://github.com/ [3]
ShowProfile(uri, req, res)
https://github.com/willdurand [2]
ShowProject(uri, req, res)
https://github.com/Propel
ShowIssues(uri, req, res)
https://github.com/propelorm/Propel/issues
ShowIssue(uri, req, res)
https://github.com/propelorm/Propel/issues/692
Get(uri, req, res)
https://github.com/ [4]
Deauthentication(uri, req, res)
https://github.com/ [5]

Figure 15: IOSTS $\mathcal{S}_4$

for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *Software Engineering, IEEE Transactions on*, 36(4):474–494, 2010.

[5] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[6] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: a tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, JSTools '12, pages 11–15, New York, NY, USA, 2012. ACM.

[7] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:13–219, 1989.

[8] L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[9] M. E. Joorabchi and A. Mesbah. Reverse engineering ios mobile applications. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 177–186, Washington, DC, USA, 2012. IEEE Computer Society.

[10] B. Lambeau, C. Damas, and P. Dupont. State-merging dfa induction algorithms with mandatory merge constraints. In A. Clark, F. Coste, and L. Miclet, editors, *Grammatical Inference: Algorithms and Applications*, volume 5278 of *Lecture Notes in Computer Science*, pages 139–153. Springer Berlin Heidelberg, 2008.

[11] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 260–, Washington, DC, USA, 2003. IEEE Computer Society.

[12] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[13] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[14] S. Salva and W. Durand. Model inference combining expert systems and formal models. Technical report, LIMOS, http://sebastien.salva.free.fr/RR-14-04.pdf, 2014. LIMOS Research report RR-14-04.

[15] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

[16] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring specifications for resources from natural language api documentation. *Autom. Softw. Eng.*, 18(3-4):227–261, 2011.