Domain-Driven Model Inference Applied To Web Applications.

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

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

***Abstract*—Model inference methods are attracting increased attention from industrials and researchers since they can be used to generate models for software comprehension, for test case generation, or for helping devise a complete model (or documentation). In this context, this paper presents an original inference model approach which recovers models from Web application HTTP traces. This approach combines formal model inference with domain-driven expert systems. Our framework, whose purpose is to simulate this human behaviour, is composed of inference rules, translating the domain expert knowledge, organised into layers. Each yields partial IOSTSs (Input Output Symbolic Transition System), which become more and more abstract and intelligible.**

***Index Terms*—Model inference, formal model, IOSTS, rule-based system**

I. Introduction and Contribution

In the Industry, legacy applications are often problematic as they are hard to maintain, poorly documented, and usually not covered by tests. When it comes to this situation, there is a high risk of introducing a bug, and options left to developers are weak. The only way to ensure stability while fixing a bug is to learn how the application behaves.

A first classic solution is to express these behaviours with formal models, for instance Input/Output Symbolic Transition Systems (IOSTS) [3]. Such models are particularly interesting to automatically generate test suites using Model- based testing techniques. But, the complete model writing is often an heavy task, and is error prone; hence the need for model inference approaches.

Model inference is a relatively recent research field aiming at recovering the application behaviours captured by a model. Zong et al. [9] proposed to infer specifications from API documentations to check whether implementations match them. Such specifications do not reflect the implementation behaviours though. In [6], specifications, which are extremely detailed, show the method calls observed from a related set of objects. Some works [4], [5], [1], [8] proposed to derive models by automatically testing an application. These are often based upon crawling techniques, which can produce either basic models or too detailed models. In both cases, it is not suitable for test case generation. For instance, Memon et al. [4] initially presented GUITAR, a tool for scanning desktop applications which produces event flow graphs and trees showing the GUI execution behaviours. The generated models are quite simple and many false event sequences have to be weeded out later. Mesbah et al. [5] proposed the tool Crawljax specialised in AJAX applications, which produces state machine models that are too complex and unreadable.

In this paper, we leverage model inference techniques in order to obtain a model from an existing application, running in a production environment. We decided to record incoming and outgoing data (traces) by monitoring applications, rather than crawling the entire application to prevent the limitations described above. Our proposal takes another direction to infer models. We do not suppose that the application being analysed is event-driven but at least yields traces. It emerges from the following idea: a domain expert, which is able to conceive specifications, is also able to diagnose the behaviour of the corresponding implementation by reading and interpreting its execution traces. His knowledge can be formalised and exploited to automatically infer models. Our approach is based upon this notion of domain knowledge, implemented with an expert system which includes inference rules. The originality of our approach also resides in the incremental production of several models, expressing the behaviour of the same application at different abstraction layers. This approach can be applied on any application producing traces, i.e. not only event-driven applications.

Below, we describe the architecture of our model inference framework. Then, we recall some definitions on the IOSTS formalism used throughout the paper in Section III. We concretely describe and define this framework in the context of Web applications in Section IV. We give some experimentation results in Section V. Conclusions are drawn in Section VI together with directions for further research and improvements.

II. Architecture of our framework

Our framework is divided into several modules as depicted in Figure 1. The *Models generator* is the centrepiece of the 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 domain expert by inferring a set of rules representing the expert

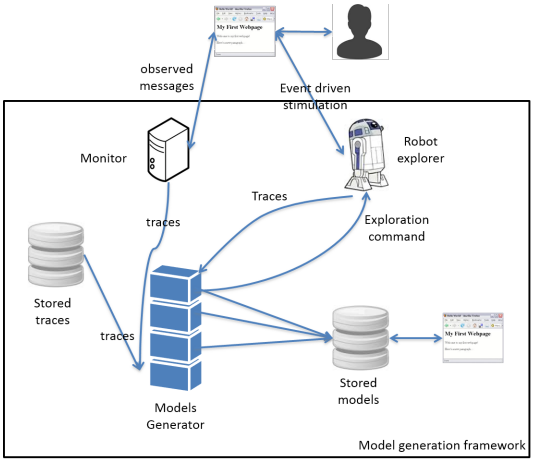


Fig. 1: Model generation framework

knowledge. This knowledge is organised into a hierarchy with several layers. Each gathers a set of inference rules written with a first order predicate logic. Typically, each layer creates two IOSTSs (except the first one), and the higher the layer is, the more abstract the IOSTSs become. These models are then successively stored and can be later analysed by experts, verification tools, etc. This number of layers is not strictly bounded even though it is manifest that it must be finite. The domain knowledge encapsulated in the expert system can be used to cover trace sets coming from several applications of the same category thanks to generic rules. But, rules can also be specialised and refined for a given application in order to yield more precise models, easing application comprehension.

Our approach allows to take a predefined set of traces collected from any kind of applications producing traces. For event-driven applications, traces could be produced using automatic testing techniques. We provide a Robot explorer which will not be presented here because of lack of room. It generates traces and find new application states in an efficient manner using strategies, that are expressed with inferences rules as well, tackling the issue related to the amount of traces needed to decently cover an application.

Our proposal 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 type. Each model, expressing the application 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 detail the different framework parts in the context of Web applications, except for the Monitor, which is here a classical proxy.

III. Model Definition and Notations

We consider the Input/Output Symbolic Transition System (IOSTS) formalism [3] 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 with ? to express actions expected by the system, and with outputs beginning with ! 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 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 variables, 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 $Q = L \times D_V$, $q_0 = (l_0, V_0)$ which is the initial one, Σ which is the set of valued symbols, and $\rightarrow$ which is the transition relation. The IOLTS semantics definition of can be found in [3]. 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* $F \times D_V$*.*

IV. Model inference

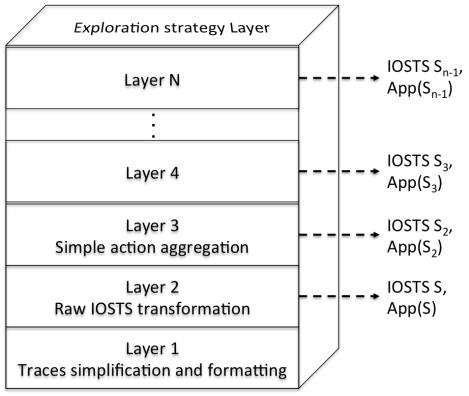


Fig. 2: Models generator layers

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 that are relevant to him. 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 two IOSTSs each: the first one $\mathcal{S}_i (i \geq 1)$ has a tree structure derived from the traces. The second IOSTS, denoted $App(\mathcal{S}_i)$, is an approximation which captures more behaviours than $\mathcal{S}_i$. Both IOSTSs are minimised with a bisimulation minimisation technique. The role of Layer 2 is to carry out a first IOSTS transformation from the structured traces. The obtained IOSTSs are not regenerated each time new traces are received but are completed on the fly. 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. The lowest layers (at least Layer 3) are composed of generic rules that can be reused on several applications of the same type. In contrast, the highest layers own the most precise rules that may be dedicated to one specific application.

In the following, and for readability purpose, we chose to represent inference rules using this format: *When conditions on facts Then actions on facts* (format borrowed from 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,
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.

A. 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:

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

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. Given a main request performed by a user, many other sub-requests are also sent 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.

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 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).

B. Layer 2: transformation of the traces into IOSTSs

Intuitively, the IOSTS transformation relies upon the IOLTS semantics transformation that is achieved in a backward manner. Two IOSTSs are built: the former, structured as a tree, represents the original traces modelled with an IOSTS formalism. The latter is an over approximation of the former. These IOSTSs are generated by performing the following steps:

1. the associated runs are computed from the structured traces by injecting states between valued actions,
2. the first IOSTS denoted $\mathcal{S}_1$ is derived from these runs and minimised,
3. a second IOSTS, denoted $App(\mathcal{S}_1)$, is obtained from $\mathcal{S}_1$ by merging some of its locations, and by also applying a minimisation technique.

These steps are detailed below:

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. 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, 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 [7].

2) IOSTS generation: The first IOSTS $\mathcal{S}_1$ is derived from the run set SR. It corresponds to 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 $r \in SR$:*

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

From an IOSTS tree $\mathcal{S}_1$, an over-approximation IOSTS can now be straightforwardly deduced by merging together all the locations of the form $(URI, k)_{k \geq 0}$ into a single location (URI). This possibly cyclic IOSTS usually expresses more behaviours and should be strongly reduced in term of location size. But this is also an approximation in the sense that new action sequences, which do not exist into the initial traces, may appear. This model may be particularly interesting to help establish a complete model or to increase the coverage of specific testing methods e.g., security testing, since more behaviours are represented. In contrast, it is manifest that a conformance testing method must not take this model as a reference to generate test cases.

Definition 5 *Let $\mathcal{S}_1$ be an IOSTS tree of SR. The approximation of $\mathcal{S}_1$, denoted $App(\mathcal{S}_1)$, is the IOSTS $< L_{App}, l0_{App}, V_{App}, V0_{App}, I_{App}, \Lambda_{App}, \rightarrow_{App} >$ such that:*

- *$L_{App} = \{(URI) \mid (URI, k) \in L_{\mathcal{S}_1}, k \geq 0\}$,*
- *$l0_{App} = l0_{\mathcal{S}_1}$, $V_{App} = V_{\mathcal{S}_1}$, $V0_{App} = V0_{\mathcal{S}_1}$, $\Lambda_{App} = \Lambda_{\mathcal{S}_1}$,*
- *$\rightarrow_{App} = \{(URI_m) \xrightarrow{a(p), G, A} (URI_n) \mid (URI_m, k) \xrightarrow{a(p), G, A} (URI_n, l) \in \rightarrow_{\mathcal{S}_1}$*

$\} \cup \{l0_{App} \xrightarrow{a(p),G,A} (URI_n) \mid l0_{\mathcal{S}_1} \xrightarrow{a(p),G,A} (URI_n, l) \in \rightarrow_{\mathcal{S}_1}\} \cup \{(URI_m) \xrightarrow{a(p),G,A} l0_{App} \mid (URI_m, k) \xrightarrow{a(p),G,A} l0_{\mathcal{S}_1} \in \rightarrow_{\mathcal{S}_1}\} (k \geq 0, l \geq 0)$.

3) IOSTS minimisation: Both IOSTSs are 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. A bisimulation minimisation algorithm can be found in [2].

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 [2]. 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 $\mathcal{S}_1$ and its approximation are 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. On the other hand, the approximation of $\mathcal{S}_1$ shares the behaviours found in $\mathcal{S}_1$ and ST but also describes new behaviours. This is captured by the following Proposition:

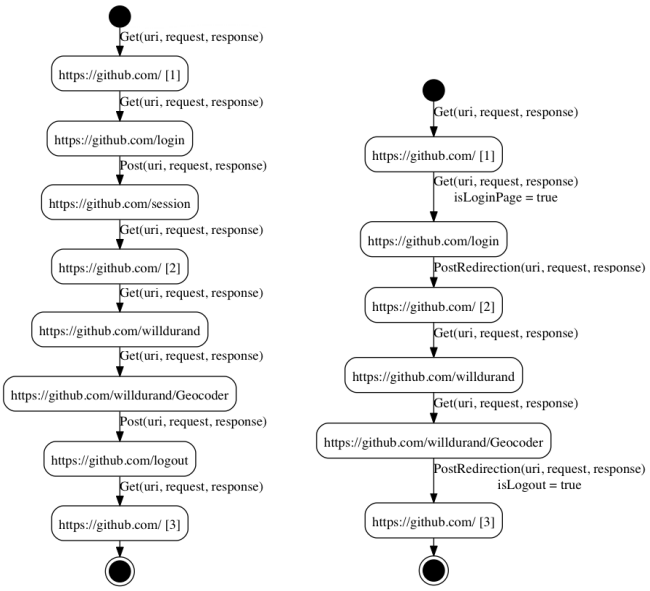
Proposition 6 *Let ST be a trace set and SR be is corresponding run set. If $\mathcal{S}_1$ is the IOSTS tree of SR, we have $Traces(\mathcal{S}_1) = ST$ and $Traces(App(\mathcal{S}_1)) \supseteq ST$.*

The proof is this proposition is Given in [7]. For sake of readability, we do not provide the rules of Layer 2, which match the above definitions and algorithms. Instead, we illustrate an IOSTS generation example below:

Example IV.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. The trace filtering from this example returns the following structured traces where the request and response parts are concealed for readability:

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

[2]https://github.com/



(a) IOSTS tree $\mathcal{S}_1$ (b) IOSTS $App(\mathcal{S}_2)$

Fig. 3: Approximation models

After having applied rules of Layer 2, we obtain the IOSTS of Figure 3(a). Locations are labelled by the URI found in the request plus 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.

C. 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. derive $App(\mathcal{S}_i)$ and apply a bisimulation minimisation on both,
3. store the two IOSTSs.

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 7 *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 7, 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.

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

1) Layer 3: As stated above, Layer 3 corresponds 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 rule depicted in Figure 4 aims at recognising the receipt of a login page: 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 5 corresponds to a simple transition aggregation, introducing a new $PostRedirection$ action.

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

Fig. 4: Login page recognition rule

Example IV.2 When we apply these rules on the IOSTS example of Figure 3(a), we obtain a new IOSTS, illustrated in Figure 3(b), which has 6 transitions instead of 9 initially. However, it does not reflect clearly the initial scenario yet. Rules deducing more abstract actions are required. These are found in the next layer.

2) Layer 4: This layer allows to infer a more abstract model composed of more expressive actions. Its rules may have different forms:

- they can be applied on a single transition. In this case, the rule replaces the transition action to add more sense,
- 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 6 recognises a user authentication thanks to the variable "isLoginPage" added by Layer 3.

```
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
```

Fig. 5: Redirection 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
```

Fig. 6: Authentication recognition rule

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 items are reserved e.g., *edu* and *explore*. The rule given in Figure 7 is based upon this structure and produces a new action $Showprofile$ offering more sense. We did the same for projects as well, introducing a $Showproject$ action.

```
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
```

Fig. 7: User profile recognition rule

Example IV.3 The application of the previous rules leads to the final IOSTS depicted in Figure 8, owning actions that have a precise meaning, and now clearly describing the application behaviour.

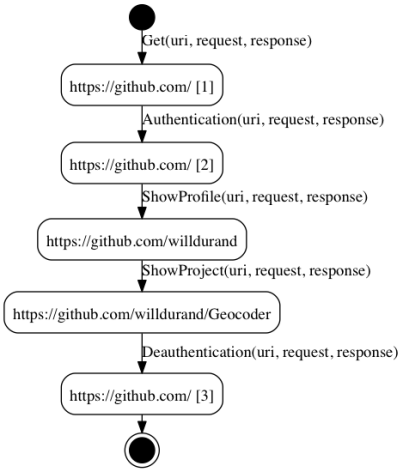


Fig. 8: IOSTS $App(\mathcal{S}_3)$ obtained from Layer 4

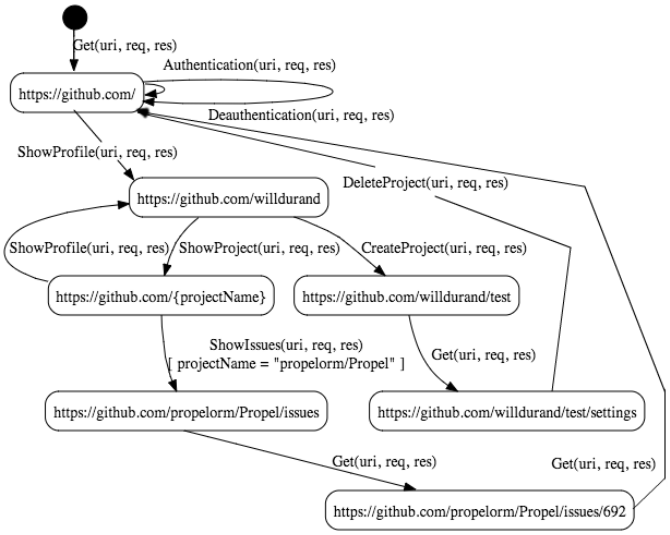


Fig. 9: IOSTS $App(\mathcal{S}_4)$ obtained from the Github Web site

V. Experimentation

The framework presented earlier 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 formatted with the HTTP Archive (HAR) format, the defacto standard for describing HTTP traces, used by various HTTP related tools (many HTTP monitoring tools, and Web browsers such as Mozilla Firefox and Google Chrome). 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 gt stobjects in this work).

From the Github Web site, 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 the trace filtering (Layer 1), we obtain a first IOSTS tree composed of 28 transitions. The next 4 layers automatically infer a last IOSTS tree $\mathcal{S}_4$ composed of 13 transitions whose 7 have a clear and intelligible meaning. Its approximation $App(\mathcal{S}_4)$ is illustrated in Figure 9. Most of its actions have a precise meaning reflecting the user interactions while the trace recording. Now, one can easily deduce that the user created, chose, deleted some projets and read the issues of others.

VI. Conclusion

This paper presents an original approach combining model inference and expert systems 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. Our approach can be applied on all applications that are able to produce traces.

We applied our framework on 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.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.