Autofunk, a fast and scalable framework for building formal models from production systems

Sébastien Salva, William Durand

Industry Paper: Autofunk, a fast and scalable framework for building formal models from production systems.

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

William Durand
Manufacture Francaise des
Pneumatiques Michelin, France
william.durand@fr.michelin.com

ABSTRACT

This paper proposes a model inference framework for production systems distributed over multiple devices exchanging thousands of events. Building models for such systems and keeping them up to date is time consuming and expensive, thus not adequately taken care of. Our framework, called *Autofunk* and designed with the collaboration of our industrial partner Michelin, combines formal model-driven engineering and expert systems to infer formal models that can be used to perform analyses, e.g. test case generation, or help diagnose faults in production by highlighting faulty behaviours. Given a large set of production events, we infer exact models that only capture the functional behaviours of a system under analysis. In this paper, we introduce and evaluate our framework on a real Michelin manufacturing system, showing that it can be used in practice.

Keywords

Model inference, STS, expert system, production system, event-driven system.

1. INTRODUCTION

Models are essential while working on the design of complex systems to build reliable implementations. But, they are also particularly useful when systems reach maintenance cycle, easing comprehension of the overall design and describing how these systems work under the hood. This is important because people who are responsible for maintaining or improving systems are most likely not the same who designed and built them. It is nearly impossible for one person to know all the details related to a particular system, hence the need for creating and maintaining models.

In the industry, building models for production systems, i.e. event-driven systems that run in production environments and are distributed over several devices and sensors, is frequent since these are valuable in many situations like testing and fault diagnosis for instance. Models may have been written as storyboards or with languages such as the Unified Modelling Language (UML) or even more formal languages. Usually, these models are designed when brand-new systems are built. It has been pointed out by our industrial partner that production systems have a life span of many years, up to 20 years, and are often incrementally updated, but not their corresponding models. This leads to a major issue which is to keep these models up to date and synchronised with the respective systems. This is a common problem with documentation in general, and it often implies rather underspecified or not documented systems that no one wants to maintain because of lack of understanding.

In this paper, we focus on this problem for production systems that exchange thousands of events a day. Model inference, a.k.a. model learning or model reverse-engineering, is a recent research field that addresses this issue. Models are built from documentation or execution traces (sequences of observed events). Several approaches have been proposed for different types of systems, usually for GUI applications, e.g. desktop or mobile applications. However, we noticed that these approaches are not tailored to support production systems. From the literature, we deduced the following key observations:

- model inference approaches give approximate models capturing the behaviours of a system and more. In our context, we want exact models that could be used for regression test case generation and fault diagnosis,
- most of these approaches perform active testing on systems to learn models. Applying active testing on running systems is not possible since these must not be disrupted,
- production systems exchange thousands and thousands of events a day. Most of the model inference approaches cannot take such a huge amount of information to build models.

Based on these observations, we propose a pragmatic model inference approach that aims at building formal models describing functional behaviours of a system. Our goal is to quickly build exact models from large amounts of production events. Execution speed takes an important place for building up to date models. Such models could also be used for diagnosis every time an issue would be experienced in production. The strong originality of our approach lies in

DEBS'15, June 29 - July 3, 2015, OSLO, Norway.

DOI: http://dx.doi.org/10.1145/2675743.2771876

the combination of two domains for model inference: model-driven engineering and expert systems. We consider formal models and their definitions to infer models by means of different transformations. But we also take into consideration the knowledge of human experts captured by expert systems. 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 events. His knowledge could then be formalised and exploited to automatically infer models. A part of our approach is based upon this notion of knowledge implemented with inference rules.

The paper is structured as follows: Section 2 gives some research directions considered in model inference and explains our choices regarding our design. Section 3 presents an overview of the framework of our approach, called *Autofunk*. This work has been conducted in the context of production systems for one of the world's leading tire manufacturer Michelin. Therefore, we describe this context, its assumptions, and a case study. We give the theoretical aspects of *Autofunk* in Section 4, and an empirical evaluation in Section 5. We conclude in Section 6.

2. RELATED WORK

Several papers dealing with model generation approaches were issued in the last decade. We present here some of them related to our work, and introduce some key observations. We only consider the approaches that infer models by observing the application behaviours at runtime, even though other papers, e.g. [13], propose to build models from documentation.

White-box techniques. Many works were proposed to infer specifications from source code or APIs, e.g. [12, 11]. Specifications are inferred in [11] from correct method call sequences on multiple related objects. The approach preprocesses method traces to identify small sets of related objects and method calls which can be analysed separately. The approach was implemented in a tool which supports more than 240 million runtime events. Other methods [3, 5] focus on mobile and web applications. They rely upon concolic testing to explore symbolic execution paths of an application and to detect bugs. These white-box approaches theoretically offer good code coverage. However, the number of paths being explored concretely is limited to short paths only. Furthermore, the constraints must not be too complex for being solved. As a consequence, the code coverage of these approaches may be lower in practice, and models tend to be too detailed, thus hard to read.

Black-box automatic techniques. Many other methods [9, 10] were proposed to build models from event-driven applications seen as black-boxes, e.g. desktop, web and more recently mobile applications. Such applications have GUIs to interact with users and which respond to user input sequences. Automatic testing methods are applied to experiment such applications through their GUIs to learn models. For instance, Memon et al. [9] introduced the tool GUITAR for scanning desktop applications. This tool produces event flow graphs and trees showing the GUI execution behaviours. The tool Crawljax [10], which is specialised in AJAX applications, produces state machine models to capture the changes of DOM structures of web documents by means of events (click, mouseover, etc.). To prevent a state space explosion, these approaches [9, 2] require state-abstractions specified by users, and given in a high level of abstraction. This decision is particularly suitable for comprehension aid, but these models often lack information for test case generation. In contrast, other approaches try to reduce models on the fly. The algorithm introduced in [10] reduces the model size by concatenating identical states of the model under construction. But this cannot be generically applied on all applications, and a state abstraction definition must be manually given.

Active learning. The $\mathcal{L}^*$ algorithm [4] is still widely considered with active learning methods for generating finite state machines. The learning algorithm is used in conjunction with a testing approach to learn models, and to guide the generation of user input sequences based on the model. The testing engine aims at interacting with the application under test to discover new application states, and to build a model accordingly. If an input sequence contradicts the learned model, the learning algorithm rebuilds a new model that meets all the previous scenarios. This approach has successfully been applied to various domains from network protocol inference to mobile applications [8, 6].

Based on these works, we concluded that active methods cannot be applied on production systems. In our context, we only assume having a set of events passively collected, i.e. collected without disrupting the system. Furthermore, the event set may be vast. We observed that most of the previous methods are not tailored for supporting large scale systems and thus millions of events. Only a few of them, e.g. [11], can take huge event sets as input and still infer models quickly. Likewise, the previous techniques often leave aside the notion of correctness regarding the learned models, i.e. whether these models only express the observed behaviours while testing one or more behaviours. The approaches [8, 6] based upon the $\mathcal{L}^*$ learning algorithm [4] do not aim at yielding exact models. The others use abstraction mechanisms to reduce model sizes. For comprehension aid, an exact model is not mandatory but the model correctness is extremely important if the model is later used for analysis purpose. In the case of production systems, it is highly probable that executing incorrect test cases can raise false positives, and it may even lead to severe damages on the devices.

That is why we propose a framework that aims at inferring models from collected events as in [11], but similarities end here. We focus on exact and formal model generation, using expert systems and inference rules to emulate human knowledge, and transition systems to embrace formal tools.

3. OVERVIEW

3.1 Context and assumptions

Michelin is a worldwide tire manufacturer and designs mos of its factories, production systems, and software by itself. Like many other industrial companies, Michelin follows the Computer Integrated Manufacturing (CIM) approach, using computers and software to control the entire manufacturing process. In this paper, we focus on the Level 2 of the CIM approach, i.e. all the applications that monitor and control several production devices and points, i.e. locations where a

production line branches into multiple lines, in a workshop. In a factory, there are different workshops for each step of the tire building process. At a workshop level, we observe a continuous stream of products from specific entry points to a finite set of exit points, i.e. where products go to reach the next step of the manufacturing process, and disappear of the workshop frame in the meantime. Millions of *production events* are exchanged among the industrial devices of the same workshop every day, allowing some factories to build over 30,000 tires a day.

Although there is a finite number of applications, each has different versions deployed in factories all over the world, potentially highlighting even more different behaviours and features. Even if a lot of efforts are put into standardizing applications and development processes, different programming languages and different frameworks are used by development teams, making difficult to focus on a single technology. Last but not least, the average lifetime of these applications is 20 years. This set is large and too disparate to apply conventional testing techniques, however most of the applications exchange events using dedicated custom internal protocols.

Our industrial partner needs a safe way to infer up to date models, independent of the underlying technical details, and without having to rely on any existing documentation. Additionally, Michelin is interested in building regression test suites to decrease the time required to deploy or upgrade systems. We came up to the conclusion that, in order to target the largest part of all Michelin's Level 2 applications, taking advantage of the production events exchanged among all devices would be the best solution, as it would not be tied to any programming language or framework, and these events contain all information needed to understand how a whole industrial system behaves in production. All these events are collected synchronously through a (centralised) logging system. Such a system logs all events with respect to their order, and does not miss any event. From these, we chose not to use extrapolation techniques to infer models, meaning our proposal generates exact models, exclusively describing what really happens in production.

This context leads to some assumptions that have been considered to design our framework:

- Black-box systems: production systems are seen as black-boxes from which a large set of production events can be passively collected. Such systems are compound of production lines fragmented into several devices and sensors. Hence, a production system can have several entry and exit points. In this paper, we denote such a system with *Sua* (System under analysis),
- Production events: an event of the form $a(\alpha)$ must include a distinctive label a along with a parameter assignment α. Two events $a(\alpha_1)$ and $a(\alpha_2)$ having the same label a must own assignments over the same parameter set. The events are ordered and processed with respect to this order,
- Traces identification: traces are sequences of events $a_1(\alpha_1)...a_n(\alpha_n)$. A trace is identified by a specific parameter that is included in all event assignments of the trace. In this paper, this identifier is denoted with *pid* and identifies products, e.g. tires at Michelin. Besides this, event assignments include a timestamp to sort them into traces.

3.2 Framework overview

In this section, we introduce our framework called *Autofunk* whose main architecture is depicted in Figure 1. This framework contains different modules (in grey in the figure): four modules are dedicated to build models, and an optional one can be used to derive more abstract and readable models.

We consider Symbolic Transition Systems (STSs) as models for representing industrial system behaviours. STSs are state machines incorporating actions (i.e. events in this context), labelled on transitions, that show what can be given to and observed on the system. In addition, actions are tied to an explicit notion of data. The innovation of this framework lies in the combination of the notion of expert systems with the STS formalism. Intuitively, the STS representation, operators and transformations, can be expressed with deduction rules. On the other hand, the knowledge of a human expert of a system can be transcribed with inference rules following the pattern: *When condition, Then action(s)*. *Autofunk* combines both domains in such a way that each model modification can be expressed and implemented with a rule. As a consequence, the data collections handled by *Autofunk* are always expressed with knowledge bases (Events, Actions, Traces, STS, etc.) on which rules are applied to infer models. Given a system *Sua* and a set of production events, *Autofunk* builds exact models, i.e. the traces of a model $\mathcal{S}$ are included in the traces of *Sua*.

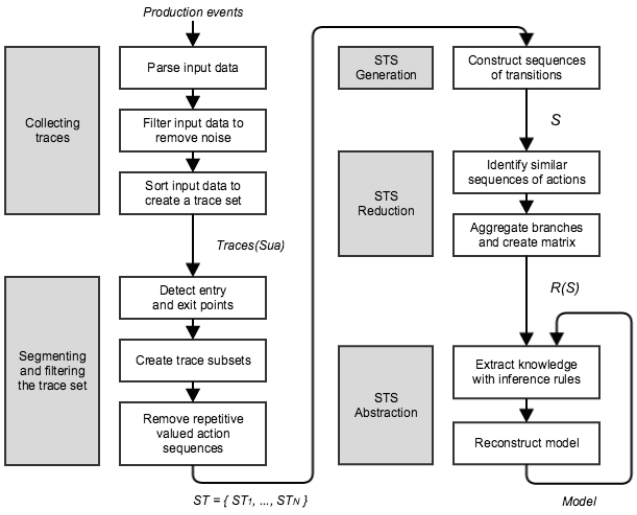

Figure 1: Overview of Autofunk

To explain how *Autofunk* works, we consider a case study based upon the example of Figure 2. It depicts simplified production events similar to those extracted from Michelin's logging system. *INFO*, *7011* and *17021* are labels that are accompanied with assignments of variables e.g. *nsys*, with indicates an industrial device number and *point* which gives the product position. With real events, there are around 20 parameters. Such a format is specific to Michelin but other

```
1 17-Jun-2014 23:29:59.00|INFO|New File
2
3 17-Jun-2014 23:29:59.50|17011|MSG_IN  [nsys: 1]
      [nsec: 8] [point: 4] [pid: 1]
4
5 17-Jun-2014 23:29:59.61|17021|MSG_OUT [nsys: 1]
      [nsec: 8] [point: 4] [tpoint: 8] [pid: 1]
6
7 17-Jun-2014 23:29:59.70|17011|MSG_IN  [nsys: 1]
      [nsec: 8] [point: 4] [pid: 2]
8
9 17-Jun-2014 23:29:59.92|17021|MSG_OUT [nsys: 1]
      [nsec: 8] [point: 4] [tpoint: 8] [pid: 2]
```

Figure 2: Production events

$Traces(Sua) = \{(17011(nsys := 1, nsec := 8, point := 4, pid := 1)\ 17021(nsys := 1, nsec := 8, point := 4, tpoint := 8, pid := 1)),\ (17011(nsys := 1, nsec := 8, point := 4, pid := 2)\ 17021(nsys := 1, nsec := 8, point := 4, tpoint := 8, pid := 2))\}$

Figure 3: Initial trace set

kinds of events could be considered by updating the first module of *Autofunk*.

3.2.1 Production events and traces

Autofunk takes production events as input from a system under analysis *Sua*. These are formatted no matter their initial source, so that it is possible to use data from different providers. We obtain a set of events of the form $a(\alpha)$ with a a label, and α a parameter assignment. In the rest of the paper, we call these formatted events, *valued events*.

Some of these valued events may be irrelevant. For instance, some events may capture logging information and are not part of the functioning of the system. In Figure 2, the event having the type *INFO* belongs to this category and can be removed. Filtering is achieved by an expert system and inference rules. Indeed, a human expert knows which events should be filtered out, and inference rules offer a natural way to express his knowledge. On top of that, expert systems also offer fast processing in this situation.

The remaining valued events are ordered to produce an initial set of traces denoted $Traces(Sua)$. Figure 3 illustrates this set obtained from the events of Figure 2.

In the context of Michelin, we use four inference rules to remove all irrelevant events. Two of them are related to the logging system itself, the two others are used to remove events that have no business meaning, and have been given by Michelin experts.

3.2.2 Traces segmentation

We define a complete trace as a trace containing all events expressing the path taken by a product in a production system, from the beginning, i.e. one of its entry points, to the end, i.e. one of its exit points. In the trace set $Traces(Sua)$, we do not want to keep incomplete traces, i.e. traces related to products which did not pass through one of the known entry points or moved to the next step of the manufacturing process using one of the known exit points.

We chose to split $Traces(Sua)$ constructed in the previous step into subsets ST_i, one for each entry point of the system under analysis *Sua*. Later, every trace set ST_i shall give birth to one model, describing all possible behaviours starting from its corresponding entry point.

In Michelin systems, the parameter *point* stores the product physical location and can be used to deduce the entry and exit points of the systems. We perform a statistical analysis on $Traces(Sua)$ and compute two ratios for each assignment $(point := val)$ found in the first and last valued events of every trace. If $Traces(Sua)$ is sufficiently large (traces collected during more than a week at Michelin), these ratios directly show the entry and exit points, respectively stored into the sets $POINT_{init}$ and $POINT_{final}$. Otherwise, we assume that the number of entry and exit points, N and M, are given and we keep only the first N and M points having the highest ratios. Then, for each entry point, we construct a trace set denoted ST_i made of traces expressing behaviours starting at this entry point and ending at one of the exit points of $POINT_{final}$. The other traces are ignored. We obtain the set $ST = \{ST_1, ..., ST_N\}$ with N the number of entry points of the system *Sua*. Finally, these traces are scrutinised to detect repetitive valued event sequences in order to remove them.

In our straightforward example, we obtain one trace set $ST_1 = Traces(Sua)$.

3.2.3 STS generation

One model is built for each trace set ST_i in ST. Given a set ST_i, a first STS, denoted $\mathcal{S}_i$, is built in a simple but quick manner. Each trace of ST_i is completed to derive a set of runs. A run is an alternate sequence of states and events. Given a trace t in ST_i, states, which are unique, are injected before and after each event of t. States must be unique to keep the ordering of the events in the runs, and to prevent merging different behaviours in the model. The initial state is an exception though as it is shared by all the runs. The model $\mathcal{S}_i$ is obtained by transforming runs into sequences of transitions that are then joined together. We obtain a model having a tree structure and whose traces are equivalent to those of ST_i. At this point, production events are called actions in the STS.

Figure 4 depicts the model obtained from the traces given in Figure 3. Every initial trace is now represented as a STS branch. Parameter assignments are modelled with constraints over transitions, called *guards*. The details about the STS models are given in Section 4.

3.2.4 STS reduction

A model $\mathcal{S}_i$ constructed with the above steps is usually too large, and thus cannot be beneficial as is. Using such a model for testing purpose would lead to too many test cases for instance. That is why our framework adds a reduction step, aiming at diminishing the first model into a second one, denoted $R(\mathcal{S}_i)$ that will be more usable. Most of the existing approaches propose two solutions. Models can directly be

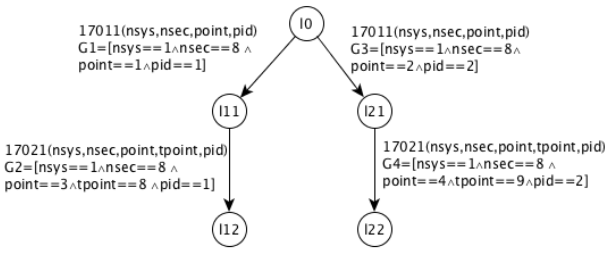

Figure 4: First generated model (STS)

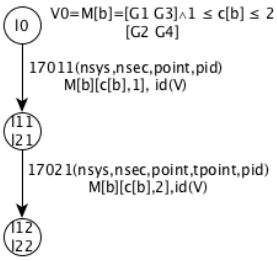

Figure 5: Reduced model (STS)

inferred with high levels of abstraction but these are also approximate, i.e. models express more behaviours than those concretely observed. This approach is not suitable since we do not want to infer extrapolated models. The second solution is to apply a minimisation technique [1] which guarantees trace equivalence. Nonetheless, after investigation, we concluded that minimisation is costly and highly time consuming on large models.

As a result, we chose to apply a simpler approach which consists in combining STS branches that have the same sequences of actions so that we still obtain a model having a tree structure. When branches are combined together, parameter assignments are wrapped into matrices in such a way that trace equivalence between the first model and the new one is preserved. The use of matrices offers here another advantage: the parameter assignments are now packed into a structure that can be more easily analysed later. As described in Section 5, this straightforward approach gives good results in terms of STS reduction and requires low processing time, even with millions of transitions.

Figure 5 depicts the reduced model obtained from the STS of Figure 4. Now we have only one branch where guards are packed into one matrix $M_{[b]}$.

3.2.5 *Model generation for comprehension aid*

For every trace set ST_i, we have built a model $R(\mathcal{S}_i)$ whose size has been drastically reduced. Such a model can be used for testing purpose, which is one of Michelin's goals, but it can also be lifted in abstraction to create more readable models. These could be used for diagnosis when issues are experienced in production.

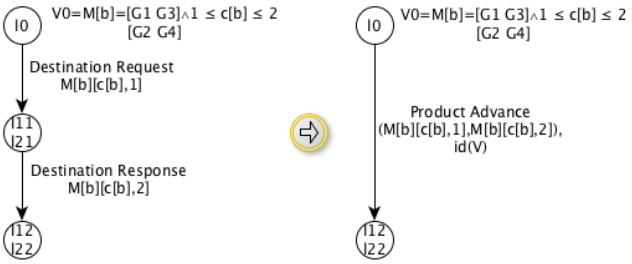

Figure 6: Final model (STS)

To infer more abstract models, we focus once again on the notion of expert knowledge. The reasoning that an expert can apply while reading events are formalised into inference rules. The latters aim at analysing the behaviours captured by a model $R(\mathcal{S}_i)$ to produce another model denoted $S_i^{\uparrow}$ having a higher level of abstraction. In this paper, we consider two kinds of inference rules:

- the inference rules that are used to enrich the meaning of the STS actions, e.g. by replacing some labels with more comprehensive ones. These rules are initially applied on the model $R(\mathcal{S}_i)$,
- we consider a second set of inference rules to analyse the meaning of sequences of transitions and to aggregate such sequences into a single transition.

If we take back our example, we obtain a last model depicted in Figure 6. The initial actions are replaced with more comprehensive ones (Figure 6(a)) by means of two inference rules. Then, the two actions expressing a moving request of a product and a response are aggregated into the action labelled as *Product Advance* (Figure 6(b)). We obtain a STS made of a unique transition whereas we had 5 production events at the beginning. Such models are easier to read and understand, but also seem to be more convenient to diagnose issues.

By writing 20 rules to enrich the meaning of the actions for our case study with Michelin, we were able to generate a model that Michelin experts understood. We then wrote 6 more rules to aggregate sequences of transitions in order to generate reduced models, mimicking some of the existing Michelin specifications.

3.3 Limitations

This framework can be applied to any kind of industrial system that meets the above assumptions. Nonetheless, it is manifest that a prelimiary evaluation on the system has to be done to establish:

1. how to parse production events,
2. the rules for event filtering,
3. the entry and exit point numbers if the production event set is not sufficient to deduce them automatically with a statistical analysis,

4. the name of the identifier parameter in production events,

5. (optional) the rules for improving the model level of abstraction. These rules may be deduced from documentation or human experts but this step may be as difficult and long as writing a model.

At the moment, the implementation of *Autofunk* does not yet support a continuous incoming flow of production events to incrementally build a model (this is not the priority of Michelin). Nevertheless, the theoretical aspects of our approach have been designed to enable this feature in the future.

4. INFERENCE-BASED MODEL GENERATION FRAMEWORK

In this section, we describe more formally the different steps illustrated in Figure 1. As stated in the overview, *Autofunk* is conceived upon the notion of expert system adopting a forward chaining. Such a system separates the knowledge base, a.k.a. facts, from the reasoning: the former is expressed with data and the latter is defined with inference rules that are applied on the facts. All information handled by *Autofunk* (events, traces, models, etc.) are then modelled with bases of facts. *Autofunk* relies upon two kinds of inference rules to infer STSs. On the one hand, we have rules based upon the STS formalism, and on the other hand, we have rules expressing expert knowledge. Now, it is evident that model inference execution has to be done in a finite time and in a deterministic way. To reach that goal, we assume that inference rules used by our framework meet the following hypotheses:

1. (finite complexity): a rule can only be applied a limited number of times on the same facts,

2. (soundness): inference rules are Modus Ponens (simple implications that lead to sound facts if the original facts are true).

4.1 Symbolic Transition Systems

Our model of choice for modelling Michelin systems is the Symbolic Transition System (STS). This model is known as a very general and powerful model for describing several aspects of event-based systems. The use of symbolic variables helps describe infinite state machines in a finite manner. This potentially infinite behaviour is represented by the semantics of a STS, given in terms of Labelled Transition System (LTS). STS operations and transformations are often given with inference rules. This aspect helps combine the two areas we consider in this paper: formal models and expert systems. We briefly give some definitions related to the STS model below, but we refer to [7] for a more detailed description.

Definition 1 (Variable assignment) *We assume that there exist a domain of values denoted D and a variable set X taking values in D. The assignment of variables in $Y \subseteq X$ to elements of D is denoted with a mapping $\alpha : Y \to D$. We denote D_Y the assignment set over Y. We also denote id_Y the identity assignment over Y, and $v_\emptyset$ the empty assignment.*

Definition 2 (STS) *A Symbolic Transition System (STS) is a tuple $< L, l_0, V, V_0, I, \Lambda, \to >$, where:*

- *STSs do not have states but locations and L is the finite location set, with l_0 being the initial one,*
- *V is the finite set of internal variables, while I is the finite set of parameters. The internal variables are initialised with the condition V_0 on V,*
- *Λ is the finite set of symbolic events $a(p)$, with $p = (p_1, ..., p_k)$ a finite set of parameters in $I^k (k \in \mathbb{N})$,*
- *$\to$ is the finite transition set. A transition $(l_i, l_j, a(p), G, A)$, from the location $l_i \in L$ to $l_j \in L$, also denoted $l_i \xrightarrow{a(p),G,A} l_j$ is labelled by:*
  - *an action $a(p) \in \Lambda$,*
  - *G is a guard over $(p \cup V \cup T(p \cup V))$ which restricts the firing of the transition. $T(p \cup V)$ are boolean terms, a.k.a. predicates over $p \cup V$,*
  - *internal variables are updated with the assignment function A of the form $(x := A_x)_{x \in V}$, A_x is an expression over $V \cup p \cup T(p \cup V)$.*

For readability purpose, we also use the generalised transition relation $\Rightarrow$ to represent STS paths: $l \xRightarrow{(a_1,G_1,A_1)...(a_n,G_n,A_n)} l' =_{def} \exists l_0, ...l_n, l = l_0 \xrightarrow{a_1,G_1,A_1} l_1 ... l_{n-1} \xrightarrow{a_n,G_n,A_n} l_n = l'$.

A STS is also associated with a LTS (Labelled Transition System) to formulate its semantics. The LTS semantics corresponds to a valued automaton without any symbolic variables, which is often infinite: the LTS states are labelled by internal variable assignments, and transitions are labelled by actions associated with parameter assignments. The semantics of a STS $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \to >$ is the LTS $||\mathcal{S}|| =< Q, q_0, \sum, \to >$ composed of valued states in $Q = L \times D_V$, $q_0 = (l_0, V_0)$ is the initial one, $\sum$ is the set of valued actions, and $\to$ is the transition relation.

Intuitively, for a STS transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain a LTS transition $(l_1, v) \xrightarrow{a(p),\alpha} (l_2, v')$ with v an assignment over the internal variable set if there exists a parameter value set α such that the guard G evaluates to true with $v \cup \alpha$. Once the transition is fired, the internal variables are assigned with v' derived from the assignment $A(v \cup \alpha)$.

Finally, runs and traces, which represent executions and event sequences, can also be derived from LTS semantics:

Definition 3 (Runs and traces) *Given a STS $\mathcal{S} = < L, l_0, V, V_0, I, \Lambda, \to >$, interpreted by its LTS semantics $||\mathcal{S}|| =< Q, q_0, \sum, \to >$, a run $q_0 \alpha_0 ... \alpha_{n-1} q_n$ is an alternate sequence of states and valued actions. $Run(\mathcal{S}) = Run(||\mathcal{S}||)$ is the set of runs found in $||\mathcal{S}||$.*

```
1  rule "Remove INFO events"
2  when:
3    $a: ValuedEvent(assignment.valueOf("type")
         == TYPE_INFO)
4  then
5    retract($a)
6  end
7
8  rule "Remove events that are repeated"
9  when
10   $a: ValuedEvent(
11     assignment.valueOf("key") matches "
           KEY_NAME_[0-9]+",
12     assignment.valueOf("inc") != null,
13     assignment.valueOf("inc") != "1"
14   )
15 then
16   retract($a)
17 end
```

Figure 7: Inference rules example for filtering

A trace of a run r is defined as the projection $proj_{\sum}(r)$ *on the actions.*

We consider this theoretical background in a backward manner to infer models from trace sets. These are collected from running production systems, then filtered out, and transformed into runs. From these, we construct STSs that are reduced and built over assignments compound of matrices of guards.

In the following, we describe *Autofunk*'s modules.

4.2 Production events and traces

Production events are collected, filtered, and formatted into traces (Figure 1). To avoid disrupting the (running) system under analysis *Sua*, we do not instrument the industrial equipments composing the whole system. Everything is done offline with a logging system or with monitoring.

As stated in the framework overview, production events are formatted into a base of valued events of the form $a(\alpha)$ with a a label and α a parameter assignment. Right after, the valued event base is filtered. These steps are performed with inference rules of the form: *When* $a(\alpha)$*, condition on* $a(\alpha)$*, Then retract(*$a(\alpha)$*).*

Figure 7 shows two concrete rules applied on Michelin systems. These two rules are written with the *Drools* [1] formalism. Drools is a rule-based expert system where knowledge bases are expressed with Java objects. The first rule removes valued events including the *INFO* parameter which do not contain any business value. The second rule removes valued events extracted from very specific events, i.e. those whose *key* matches a pattern and having a *inc* value that is not equal to 1. This rule, given by a Michelin expert, removes some duplicate events.

From this filtered valued event base, we reconstruct the corresponding traces from the trace identifier *pid*, present in each valued event, and timestamps. We call the resulting trace set $Traces(Sua)$:

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

Definition 4 ($Traces(Sua)$**)** *Given a system under analysis Sua,* $Traces(Sua)$ *denotes its formatted trace set.* $Traces(Sua)$ *includes traces of the form* $(a_1,\alpha_1)....(a_n,\alpha_n)$ *such that* $(a_i,\alpha_i)_{(1\leq i\leq n)}$ *are (ordered) valued events having the same identifier assignment.*

We can now state that a STS model $\mathcal{S}$ is said exact iff. $Traces(\mathcal{S}) \subseteq Traces(Sua)$.

Algorithm 1: Trace segmentation algorithm

input : $Traces(Sua)$, optionally entry point number N and/or exit point number M
output: $ST = \{ST_1, ..., ST_n\}$

Step 1. $Traces(Sua)$ *segmentation*
foreach $t = (a_1,\alpha_1)...(a_n,\alpha_n) \in Traces(Sua)$ **do**
  $Rinit((point := val) \subset \alpha_1) + +$;
  $Rfinal((point := val2) \subset \alpha_n) + +$;
$POINT_{init} = \{(point := val) \mid Rinit((point := val)) >$ 10% or belongs to the N highest ratios$\}$;
$POINT_{final} = \{(point := val) \mid Rfinal((point := val)) >$ 10% or belongs to the M highest ratios$\}$;
foreach $\alpha_i = (point := val) \in POINT_{init}$ **do**
  $ST_i = \{a1(\alpha_1)...an(\alpha_n) \in Traces(Sua) \mid \alpha_i \subset \alpha_1, \exists(point := val2) \subset \alpha_n, (point := val2) \in POINT_{final}\}$;
$ST := \{ST_1, ..., ST_N\}$;
Step 2. trace filtering
foreach $t = \sigma_1 p...p\sigma_n \in ST$ **do**
  if $\exists t' = \sigma'_1 p' \sigma'_n \in ST$ *such that* $p \sim_{(pid)} p'$, $\sigma_1 \sim_{(pid)} \sigma'_1$, $\sigma_n \sim_{(pid)} \sigma'_n$ **then**
    $ST := ST/\{t\}$;

4.3 Trace segmentation and filtering

This module performs two steps which are summarised in Algorithm 1. It starts by splitting $Traces(Sua)$ into several trace sets ST_i, one for each entry point of the system *Sua*, and then removes incomplete traces. Since we want a framework as flexible as possible, we chose to perform a statistical analysis on $Traces(Sua)$ aiming at automatically detecting the entry and exit points. This analysis is performed on the assignments $(point := val)$ found in the first and last valued events of the traces of $Traces(Sua)$ since *point* captures the product physical location and especially the entry and exit points of *Sua*. We obtain two ratios $Rinit(point := val)$ and $Rfinal(point := val)$. Based on these ratios, one can deduce the entry point set $POINT_{init}$ and the exit point set $POINT_{final}$ if $Traces(Sua)$ is large enough. Pragmatically, we observed that the traces collected during one or two days are not sufficient because they do not provide enough differences between the ratios. In this case, we assume that the number of entry and exit points, N and M, are given and we keep the first N and M ratios only. On the other hand, a week seems to offer good results. We chose to set a fixed yet configurable minimum limit to 10%. Assignements $(point := val)$ having a ratio below this limit are not retained. Then, for each assignment $\alpha_i = (point := val)$

in $POINT_{init}$, we construct a trace set ST_i such that a trace of ST_i has a first valued event including the assignment α_i, and ends with a valued event including an assignment ($point := val2$) in $POINT_{final}$. We obtain the set $ST = \{ST_1, ..., ST_N\}$ with N the number of entry points of the system Sua.

Thereafter, *Autofunk* scans the traces in ST and tries to detect repetitive patterns $p, ..., p$. If it finds a trace t having a repetitive pattern p and another equivalent trace including this pattern p once, then t is removed since we suppose that t does not express a new and interesting behaviour. Here, traces are removed rather than deleting the repetitive patterns to prevent from modifying traces and to keep the trace inclusion property between the sets ST_i and $Traces(Sua)$. In Algorithm 1, two traces $t = \sigma_1 p, ..., p\sigma_n$ and $t' = \sigma_1 p' \sigma_n$ are said equivalent if the patterns p, p' and the sub-sequences are equivalent, denoted with the $\sim_{(pid)}$ notation. Intuitively, this relation means that the two equivalent sequences must have the same successive valued events after having removed the assignments of the variable *pid*.

We obtain a set $ST = \{ST_1, ..., ST_N\}$ according to the following proposition:

Proposition 5 *Let $ST = \bigcup_{1 \leq i \leq N} ST_i$ be the trace set obtained from Sua. We have $Traces(ST_i) \subseteq Traces(Sua)$.*

4.4 STS generation

Given a trace set $ST_i \in ST$, the STS generation is incrementally done by transforming traces into runs, and runs into STSs. The translation of ST_i into a run set denoted $Runs_i$ is done by completing traces with states. Each run starts by the same initial state $(l_0, v_\emptyset)$ with $v_\emptyset$ the empty assignment. Then, new states are injected after each event. $Runs_i$ is formally given by the following definition:

Definition 6 (Structured Runs) *Let ST_i be a trace set obtained from Sua. We denote $Runs_i$ the set of runs derived from ST_i with the following inference rule:*

$$\frac{\sigma_{k(1 \leq k \leq n)} = (a_1, \alpha_1)...(a_n, \alpha_n) \in ST_i}{(l_0, v_\emptyset)(a_1, \alpha_1)(l_{k1}, v_\emptyset)...(l_{kn-1}, v_\emptyset)(a_n, \alpha_n)(l_{kn}, v_\emptyset) \in Runs_i}$$

The above definition preserves trace inclusion between $Runs_i$ and $Traces(Sua)$, and we can deduce the following proposition:

Proposition 7 *Let ST_i be a trace set obtained from Sua. We have $Traces(Runs_i) \subseteq Traces(Sua)$.*

The runs of $Runs_i$ have states that are unique except for the initial state $(l_0, v_\emptyset)$. We defined such a set to ease the process of building a STS having a tree structure. Runs are transformed into STS paths that are assembled together by means of a disjoint union. The resulting STS forms a tree compound of branches starting from the location l_0. Parameters and guards are extracted from the assignments found in valued events:

Definition 8 *Given a run set $Runs_i$, $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ is the STS expressing the behaviours found in $Runs_i$ such that:*

- $L_{\mathcal{S}_i} = \{l_i \mid \exists r \in Runs_i, (li, v_\emptyset)$ *is a state found in* $r\}$,
- $l0_{\mathcal{S}_i} = l_0$ *is the initial location such that* $\forall r \in Runs_i$, r *starts with* $(l_0, v_\emptyset)$,
- $V_{\mathcal{S}_i} = \emptyset$, $V0_{\mathcal{S}_i} = v_\emptyset$,
- $\rightarrow_{\mathcal{S}_i}$ *and* $\Lambda_{\mathcal{S}_i}$ *are defined by the following inference rule applied on every element* $r \in Runs_i$*:*

$$\frac{(l_i, v_\emptyset)(a_i, \alpha_i)(l_{i+1}, v_\emptyset) \in r, p = \{x \mid (x := v) \in \alpha_i\}, G_i = \bigwedge_{(x:=v) \in \alpha_i} x == v}{l_i \xrightarrow{a_i(p), G_i, id_V}_{\mathcal{S}_i} l_{i+1}}$$

Now, we have a first STS that has a tree form, describing all behaviours of the system under analysis.

Figure 4 illustrates the STS $\mathcal{S}_1$ obtained from the production events of Figure 2. We have STS actions and each one owns a parameter list. Transitions are labelled with guards derived from parameter assignments. This STS expresses the behaviours found in $Traces(Sua)$ but in a slightly different manner. More generally, trace inclusion between an inferred STS and $Traces(Sua)$ is captured by the following proposition:

Proposition 9 *Let Sua be a system under analysis and $Traces(Sua)$ be its trace set. $\mathcal{S}_i$ is an inferred STS from $Traces(Sua)$.*
We have $Traces(\mathcal{S}_i) = Traces(ST_i) \subseteq Traces(Sua)$.

4.5 STS reduction

A STS $\mathcal{S}_i$ is most likely too large for being analysed in an efficient manner. Given that a production system has a finite number of elements and that there should only be deterministic decisions, the STS $\mathcal{S}_i$ should contain branches capturing the same sequences of events (without necessarily the same parameter assignments). Consequently, it sounds natural to try to reduce the STS obtained from the previous step.

Because our goal is to produce exact models quickly, we propose to apply a lightweight STS reduction method which also aims at gathering data in order to ease the data analysis later on. Our method merges complete branches that have the same action sequences whereas guards, which capture parameter assignments, are merged into matrices. More precisely, a sequence of successive guards found in a branch is stored into a matrix column. By doing this, we reduce the model size and we can still retrieve original behaviours and only these ones. We still preserve trace inclusion between the reduced STS and $Traces(Sua)$.

Given a STS $\mathcal{S}_i$, every STS branch is initially adapted to express sequences of guards in a vector form to ease the STS reduction. Later, the concatenation of these vectors shall give birth to matrices. This adaptation is obtained with the definition of the STS operator Mat:

Definition 10 *Let* $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ *be a STS. We denote* $Mat(\mathcal{S}_i)$ *the STS operator which consists in expressing guards of STS branches in a vector form.*

$Mat(\mathcal{S}_i) =< L_{Mat(\mathcal{S}_i)}, l0_{Mat(\mathcal{S}_i)}, V_{Mat(\mathcal{S}_i)}, V0_{Mat(\mathcal{S}_i)}, I_{Mat(\mathcal{S}_i)}, \Lambda_{Mat(\mathcal{S}_i)}, \rightarrow_{Mat(\mathcal{S}_i)} >$ *where:*

- $L_{Mat(\mathcal{S}_i)} = L_{\mathcal{S}_i}, l0_{Mat(\mathcal{S}_i)} = l0_{\mathcal{S}_i}, I_{Mat(\mathcal{S}_i)} = I_{\mathcal{S}_i}, \Lambda_{Mat(\mathcal{S}_i)} = \Lambda_{\mathcal{S}_i}$,
- $V_{Mat(\mathcal{S}_i)}$, $V0_{Mat(\mathcal{S}_i)}$ *and* $\rightarrow_{Mat(\mathcal{S}_i)}$ *are given by the following rule:*

$$\frac{b_i = l_0 \xrightarrow{(a_1(p_1),G_1,A_1)\ldots(a_n(p_n),G_n,A_n)} l_n}{\begin{array}{c} V0_{Mat(\mathcal{S}_i)} := V0_{Mat(\mathcal{S}_i)} \wedge M_i = [G_1, ..., G_n] \\ l0_{Mat(\mathcal{S}_i)} \xrightarrow{(a_1(p_1),M_i[1],id_V)\ldots(a_n(p_n),M_i[n],id_V)} \rightarrow_{Mat(\mathcal{S}_i)} l_n \end{array}}$$

Given a branch $b_i \in (\rightarrow_{Mat(\mathcal{S}_i)})^n$, *we also denote* $Mat(b_i) = M$ *the vector used with* b_i.

Now, we are ready to merge the STS branches that have the same sequences of actions. This last sentence can be interpreted as an equivalence relation over STS branches from which we can derive equivalence classes:

Definition 11 (STS branch equivalence class) *Let* $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ *be a STS obtained from* $Traces(Sua)$ *(and having a tree structure).* $[b]$ *denotes the equivalence class of* $\mathcal{S}_i$ *branches such that:*
$[b] = \{b_j = l0_{\mathcal{S}_i} \xrightarrow{(a_1(p_1),G_{1j},A_{1j})\ldots(a_n(p_n),G_{nj},A_{nj})} l_{nj} (j \geq 1) \mid b = l0_{\mathcal{S}_i} \xrightarrow{(a_1(p_1),G_1,A_1)\ldots(a_n(p_n),G_n,A_n)} l_n\}$

The reduced STS denoted $R(\mathcal{S}_i)$ of $\mathcal{S}_i$ is obtained by concatenating all the branches of each equivalence class $[b]$ found in $Mat(\mathcal{S}_i)$ into one branch. The vectors found in the branches of $[b]$ are concatenated as well into the same unique matrix $M_{[b]}$. A column of this matrix represents a complete and ordered sequence of guards found in one initial branch of $\mathcal{S}_i$. $R(\mathcal{S}_i)$ is defined as follow:

Definition 12 *Let* $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ *be a STS inferred from a structured trace set* $Traces(Sua)$. *The reduction of* $\mathcal{S}_i$ *is modelled by the STS* $R(\mathcal{S}_i) =< L_R, l0_R, V_R, V0_R, I_R, \Lambda_R, \rightarrow_R >$ *where:*

$$\frac{\begin{array}{c}[b] = \{b_1, ..., b_m\} \\ b = l0_{\mathcal{S}_i} \xrightarrow{(a_1(p_1),G_1,A_1)\ldots(a_n(p_n),G_n,A_n)} Mat(\mathcal{S}_i)\ l_n\end{array}}{\begin{array}{c} V0_R := V0_R \wedge M_{[b]} = [Mat(b_1), ..., Mat(b_m)] \\ \wedge (1 \leq c_{[b]} \leq m), \\ l0_R \xrightarrow{(a_1(p_1),M_{[b]}[1,c_{[b]}],id_V)\ldots(a_n(p_n),M_{[b]}[n,c_{[b]}],id_V)} \rightarrow_R \\ (l_{n1}...l_{nm}) \end{array}}$$

The resulting model $R(\mathcal{S}_i)$ is a STS composed of variables assigned to matrices whose values are used as guards. A matrix column represents a successive list of guards found in a branch of the initial STS $\mathcal{S}_i$. The choice of the column in a matrix depends on a new variable $c_{[b]}$.

Figure 4 has two branches that can be combined since they have the same action sequences. During the construction of the reduced STS depicted in Figure 6, the guards are placed into two vectors $M1 = [G1\ G2]$ and $M2 = [G3\ G3]$. These are combined into the same matrix $M_{[b]}$. The variable $c_{[b]}$ is used to take either the guards of the first column or the guards of the second one.

The STS $R(\mathcal{S}_i)$ has less branches but still expresses the initial behaviours described by the STS $\mathcal{S}_i$. This is captured with the following proposition:

Proposition 13 *Let Sua be a system under analysis and* $Traces(Sua)$ *be its traces set.* $R(\mathcal{S}_i)$ *is a STS derived from* $Traces(Sua)$. *We have* $Traces(R(\mathcal{S}_i)) = Traces(ST_i) \subseteq Traces(Sua)$.

4.6 STS abstraction

Given the trace set $ST_i \in ST$, the generated STS $R(\mathcal{S}_i)$ can be used for analysis purpose but is still difficult to manually interpret, even for experts. This *Autofunk* module aims to analyse $R(\mathcal{S}_i)$ to produce a new STS $\mathcal{S}_i^{\uparrow}$ whose level of abstraction is lifted by using more intelligible actions. This process is performed with inference rules, which encode the knowledge of the expert of the system. These are triggered on the transitions of $R(\mathcal{S}_i)$ to deduce new transitions. We consider two types of rules:

- the rules replacing some transitions by more comprehensive ones. These rules are of the form: *When Transition* $l_1 \xrightarrow{a(p),G,A}_{R(\mathcal{S}_i)} l_2$, *condition on* $a(p), G, A$, *Then add* $l_1 \xrightarrow{a'(p'),G',A'}_{\mathcal{S}_i^{\uparrow}} l_2$ *and retract* $l_1 \xrightarrow{a(p),G,A}_{R(\mathcal{S}_i)} l_2$.
- the rules that aggregate some successive transitions to a single transition compound of a more abstract action. These rules are of the form *When Transition* $l_1 \xrightarrow{(a_1,G_1,A_1),\ldots(a_n,G_n,A_n)} l_n$, *condition on* $(a_1, G_1, A_1), ...(a_n, G_n, A_n)$, *Then add* $l_1 \xrightarrow{a(p),G,A}_{\mathcal{S}_i^{\uparrow}} l_n$, *and retract* $l_1 \xrightarrow{(a_1,G_1,A_1),\ldots(a_n,G_n,A_n)} l_n$.

The generated STSs represent recorded scenarios modelled at a higher level of abstraction. These can be particularly useful for generating documentation or better understanding how the system behaves, especially when issues are experienced in production. However, it is manifest that the trace inclusion property is lost with the STSs constructed by this module since sequences are modified.

If we take back our example, the actions of the STS of Figure 5 are replaced with the rules of Figure 8 which change the labels *17011* and *17021* to more intelligible ones. The third rule of Figure 9 aggregates the two transitions into a unique transition indicating the movement of a product in its production line. These rules are also written using the *Drools* formalism. Here, $Transition$ are facts modelling

```
1  rule "Mark destination requests"
2  when:
3    $t: Transition(name matches "17011")
4  then
5    $t.changeAction("Destination Request")
6  end
7
8  rule "Mark destination responses"
9  when:
10   $t: Transition(name matches "17021")
11 then
12   $t.changeAction("Destination Response")
13 end
```

Figure 8: Two rules adding value to existing transitions

```
1 rule "Aggregate destination requests/responses"
2 when
3   $t1: Transition(action == "Destination Request
        ", $lfinal := Lfinal)
4   $t2: Transition(action == "Destination
        Response" , Linit == $lfinal)
5 then
6   insert(new Transition("Product Advance", Guard
        ( $t1.Guard, $t2.Guard), Assign($t1.Assign
        , $t2.Assign), $t1.Linit, $t2.Lfinal))
7   retract($t1)
8   retract($t2)
9 end
```

Figure 9: STS transition aggregation rule

STS transitions. From 5 initial production events that are not self-explanatory, we generate a simpler STS constituted of one transition, clearly expressing a part of the functioning of the system.

5. IMPLEMENTATION AND EXPERIMENTATION

In this section, we briefly describe the implementation of our model inference framework for Michelin. Then, we give an evaluation on a real production system.

5.1 Implementation

Our framework *Autofunk* is developed in Java and mainly based on *Drools* [2], a Java rule-based expert system engine. Drools supports knowledge bases with facts given as Java objects. In our context, we have several bases of facts used throughout the different *Autofunk* modules: Events, Trace sets ST_i, Runs, Transitions and STSs. We chose to target performance and simplicity while implementing *Autofunk*. That is why most of the steps are implemented with parallel algorithms (except the production event parsing) which are based upon the inference rules given in Section 4.

The input trace collection is constructed with a classical parser with returns *Event* Java objects. By now, we are not able to parallelise this part because of an issue we faced with Michelin's logging system. The resulting drawback is that the time to parse traces is longer than expected and heavily depends on the size of data to parse. The *Event* base is then filtered with Drools inference rules as presented in Section 4.2. Then, we call a straightforward algorithm for reconstructing traces: it iterates over the *Event* base and creates a set for each assignment of the identifier *pid*. These sets are sorted to construct traces given as *Trace* Java objects. These objects correspond to $Traces(Sua)$. The generation of the trace subsets $ST = \{ST_1, ..., ST_N\}$ and of the first STSs are done with Drools inference rules as described in Section 4, but applied in parallel. The STS reduction, and specifically the generation of STS branches equivalence classes, has been implemented with a specific algorithm for better performance. Indeed, comparing every action in STS branches in order to aggregate them is time consuming. Given a STS $\mathcal{S}$, this algorithm generates a signature for each branch b, i.e. a hash (SHA1 algorithm) of the concatenation of the signatures of the actions of b. The branches which have the same signature are gathered together and establish branch equivalence classes (as described in Section 4.5). Thereafter, the reduced $R(\mathcal{S})$ is constructed thanks to the inference rule given in Section 4.5.

[2]http://www.drools.org/

5.2 Evaluation

We conducted several experiments with real sets of production events, recorded in one of Michelin's factories at different periods of time. We executed our implementation on a Linux (Debian) machine with 12 Intel(R) Xeon(R) CPU X5660 @ 2.8GHz and 64GB RAM.

We present here the results of 6 experiments on the same production system with different event sets collected during 1, 8, 11, 20, and 23 days. These results are depicted in Figure 10. For confidentiality reasons, we are not able to provide results related to the generation of more abstract models. The third column gives the number of production events recorded on the system. The next column shows the trace number obtained after the parsing step. N and M represent the entry and exit points automatically computed with the statistical analysis. The column Trace Subsets shows how $Traces(Sua)$ is segmented into subsets $\{ST_1, ..., ST_N\}$ and the number of traces included in each subset. These numbers of traces also correspond to the numbers of branches generated in the STSs $\mathcal{S}_1, ..., \mathcal{S}_N$. The eighth column, $\# R(\mathcal{S}_i)$, represents the number of branches found in each reduced STSs $R(\mathcal{S}_1), ..., R(\mathcal{S}_N)$. Finally, execution times are rounded and expressed in minutes in the last column.

First, these results show that our framework can take millions of production events and still builds models quickly (less than half an hour). With sets collected during one day up to one week (experiments A, B, C, and D), models are inferred in less than 10 minutes. Hence, *Autofunk* can be used to quickly infer models for analysis purpose or to help diagnose faults in a system. Experiment F handled almost 10 million events in less than half an hour to build two models including around 1,600 branches. As mentioned in Section 5.1, the parsing process is not parallelized yet, and it took up to 20 minutes to open and parse around 1,000 files (number of Michelin log files for this experiment). This is an issue we want to tackle in the next version of *Autofunk*. The graph shown in Figure 13 summarises the performances of our framework and how fast it is at transforming production events into models (experiments B, C and D run in about 9 minutes). It also demonstrates that doubling the event set does not involve doubling its execution time. The exponential trend line reveals that the overall framework scales well, even with the current parsing implementation.

Exp.	# Days	# Events	$Card(Traces(Sua))$	N	M	# Trace Subsets	# $R(\mathcal{S}_i)$	Exec Time (min)
A_1	1	660,431	16,602	2	3	4,822	332	1
A_2						1,310	193	
B_1	8	3,952,906	66,880	3	3	28,555	914	9
B_2						18,900	788	
B_3						6,681	51	
C_1	11	3,615,215	61,125	3	3	28,302	889	9
C_2						14,605	681	
C_2						7,824	80	
D_1	11	3,851,264	73,364	2	3	35,541	924	9
D_2						17,402	837	
E_1	20	7,635,494	134,908	2	3	61,795	1,441	16
E_2						35,799	1,401	
F_1	23	9,231,160	161,035	2	3	77,058	1,587	24
F_2						43,536	1,585	

Figure 10: Results of 6 experiments on a Michelin industrial system

In Figure 10, the difference between the number of trace subsets (7th column) and the number of branches included in the STSs $R(\mathcal{S}_i)$ (8th column) clearly shows that our STS reduction approach is effective. For instance, with experiment B, we reduce the STSs by 91.88% against the initial trace set $Traces(Sua)$. In other words, 91% of the original behaviours are packed into matrices.

We also extracted the values of columns 4 and 7 in Figure 10 to depict the stacked bar chart illustrated in Figure 11. This chart shows, for each experiment, the proportion of complete traces kept by *Autofunk* to build models, over the initial number of traces in $Traces(Sua)$. *Autofunk* has kept only 37% of the initial traces in Experiment A because its initial trace set is too small and contains many incomplete behaviours. During a day, most of the recorded traces do not start or end at entry or exit points, but rather start or end somewhere in production lines. Indeed, a workshop contains storage areas where products can stay for a while, depending on the production campaigns or needs for instance. That is why, on a single day, we can find so many incomplete traces. With more production events, such a phenomenon is limited because we absorb these storage delays.
We can also notice that experiments C and D have similar initial trace sets but experiment C owns more complete traces than experiment D by 12%, which is significant. Furthermore, experiments B and C take 3 entry points into account while the others only take 2 of them. This is related to the fixed limit of 10% we chose to ensure truly entry points to be automatically selected. The workshop we analysed has three entry points whose two are mainly used. The third entry point is employed to equilibrate the production load between this workshop and a second one located close to it in the same factory. Depending on the period, this entry point may be more or less sollicitated, hence the difference between experiments B, C and experiment D. Increasing the limit of 10% to a higher value would change the value of N for experiments B and C, but would also impact experiment A by introducing false results since incorrect entry points could be selected. By means of a manual analysis, we concluded that 10% was the best ratio for removing incomplete traces in our experiments. 30% of initial traces have been removed, which is close to the reality. However, this simple analysis could be improved in the future.

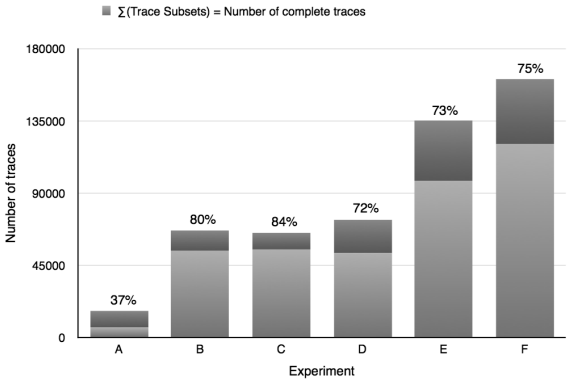

Figure 11: Proportions of complete traces

Another potential issue with our parsing implementation is that every event has to be loaded in memory, so that we can perform computation and apply our algorithms on them. However working with millions of Java objects requires enough memory, i.e. memory consumption depends on the amount of initial traces. We compared execution time and memory consumption in Figure 12, showing that memory consumption tends to follow a logarithmic trend. In the next version of Autofunk, we plan to work on improving memory consumption even if it has been considered acceptable as is by Michelin.

6. CONCLUSION

This paper presents *Autofunk*, a fast and scalable framework combining model inference and expert systems to generate models from production systems. Given a large set of production events, our framework infers exacts models whose traces are included in the initial trace set of a system under analysis. We chose to design *Autofunk* for targeting high performance. Our evaluation shows that this approach is suitable in the context of production systems since we quickly obtain STS trees reduced by 90% against the origi-

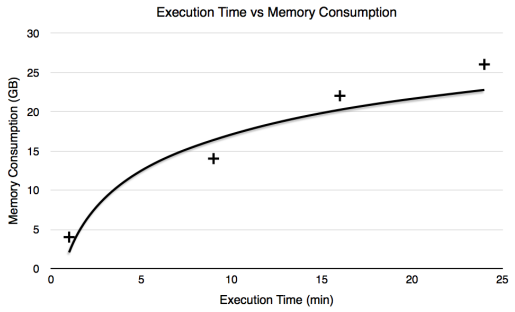

Figure 12: Memory consumption vs execution time

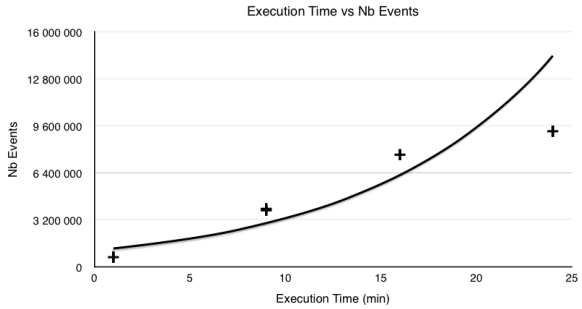

Figure 13: Execution time vs events

nal trace sets of the system under analysis.

Nevertheless, many aspects need to be investigated and improved in the future. From a technical perspective, our implementation should be enhanced to speed up the event parsing, for instance by considering a message queuing protocol, and to optimize memory consumption. Our STS reduction approach could also be improved by concatenating partial equivalent STS branches. However, a naive solution would affect performance as partial branch concatenation is time consuming, and we do not want to sacrifice execution speed. We also plan to use this framework to propose a new testing approach taking advantage of the inferred models. In short, inferred models could be used to generate event-based scenarios to test production systems, and an improved version of *Autofunk* could check the compliance of the recorded event sets against inferred models.

7. REFERENCES

[1] P. A. Abdulla, L. Kaati, and J. Hogberg. Bisimulation minimization of tree automata. Technical report, In Proc. 11th Int. Conf. Implementation and Application of Automata, volume 4094 of LNCS, 2006.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar – a tool for automated model-based testing of mobile apps. *IEEE Software*, NN(N):NN–NN, 2014.

[3] 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.

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

[5] 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.

[6] 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 & Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[7] 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.

[8] H. Hungar, T. Margaria, and B. Steffen. Model generation for legacy systems. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future, 9th International Workshop, RISSEF 2002, Venice, Italy, October 7-11, 2002, Revised Papers*, volume 2941 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2002.

[9] 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.

[10] 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.

[11] 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.

[12] M. Salah, T. Denton, S. Mancoridis, and A. Shokouf. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *In ICSM*, pages 155–164. IEEE Computer Society, 2005.

[13] 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.