Combining model generation and passive testing in the same framework to test industrial systems

Sébastien Salva [1], William Durand [2]

Research Report LIMOS/RR-17-03

20 juin 2017

1. LIMOS, UMR 6158, University Clermont Auvergne, IUT d'AubiÃ¨re, FRANCE, sebastien.salva@uca.fr

2. Manufacture Francaise des Pneumatiques Michelin, LIMOS, william.durand@fr.michelin.com

Abstract

Many software engineering approaches often rely on formal models to automate some steps of the software life cycle, particularly the testing phase. Even though automation sounds attractive, writing models is usually a tedious and error-prone task. In addition, with industrial software systems, models, when they exist, are often not up-to-date. Hence, testing these systems becomes problematic. In this context, this paper proposes a method and a framework called *Autofunk* to test production systems by combining two approaches: model generation and passive testing. Given a large set of events collected from a production system, Autofunk combines the notions of expert system, formal models and machine learning to infer symbolic models while preventing over-generalisation (i.e., the models should not capture more behaviours than those possible in the real system). Afterwards, these models are considered to passively test whether another system is conforming to the models. As the generated models do not express all the possible behaviours that should happen, we define conformance with four specialised implementation relations. Two relations are dedicated for offline passive testing (events are collected, while the system is running, the tester gives verdicts afterwards). The two remaining relations are for the online mode (events are analysed on the fly).

Keywords: Model inference ; Passive testing ; Industrial systems ; Symbolic models

1 Introduction

This paper tackles the problem of testing production systems such as those of our industrial partner Michelin, one of the three largest tire manufacturers in the world. A production system is defined as a set of production machines controlled by a software, in a factory. Such systems are composed of heterogeneous devices, interconnected with specialised networks and controlled by a software in a factory. Testing them is often performed manually with simulations to replicate human operations and to not damage real devices. This testing phase usually requires a long period of time, from some weeks up to several months.

Passive testing is an approach that can partially automate this stage and shorten its deadline. Generally speaking, a passive tester (also known as observer) collects observations from the system and aims at checking if its behaviour meets requirements expressed in a model. Testing can be performed in either online or offline mode. Online passive testing means that sequences of observations, called *traces*, are computed and analysed on-the-fly for the detection of defects; in offline mode, traces are collected and analysed later. However, passive testing suffers from a common issue : we need of a specification (models or properties). And writing a specification is known as a long and error-prone task.

Model inference is a research field, which brings appealing concepts to bypass this issue. It proposes a set of techniques that infer models describing how a system behaves by analysing system executions. A model, inferred from an initial production system, could help in the test of a new or updated one. Here comes the context proposed by our partner Michelin who wishes a way to automate the testing of new or updated systems, but without having models. To cope with this problematic, we have chosen to devise a framework, called *Autofunk* (for Automatic Functional model inference), which combines model inference and passive testing. This paper presents this framework, i.e., the theoretical background that we considered and preliminary results.

1.1 Context

Michelin is a worldwide tire manufacturer and designs most of its factories, production systems, and related software. Like many other industrial companies, Michelin follows the Computer Integrated Manufacturing (CIM [WYD07]) approach, using computers and software to control the entire manufacturing process and acquire data. The CIM approach segments the manufacturing process and production strategies into several hierarchical levels: CMI1 is the device level, CMI2 includes all the applications that monitor and control devices. Levels 3 and 4 focus on the factory management.

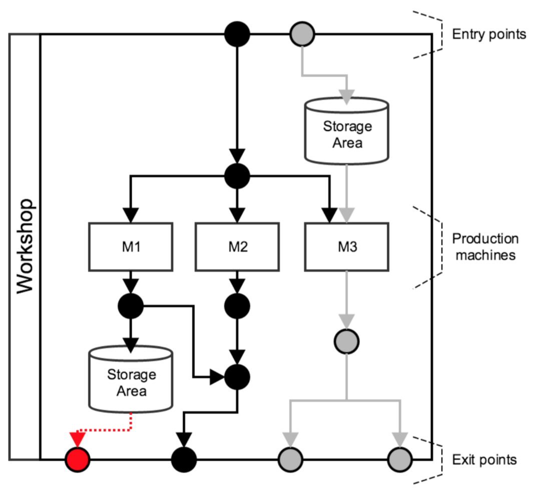


Figure 1 – Simplified representation of a workshop

A factory includes workshops, each devoted to a step of the tire building process, e.g., tire assembling (assembling the components onto a tire drum) or curing (applying pressure on assembled tires in molds to give their final shapes). A workshop gathers devices, branch points, conveyor belts and human operators that perform specific actions (removal of products to assess their qualities, etc.). A workshop is controlled by a set of CIM2 applications (except for the operators): every order (move, stop, change state, etc.), product modifications or alerts passing among industrial devices and software are materialised with messages that we call *production events*. These applications are continuously updated and sometimes replaced, for instance when the physical configuration of the workshop is modified, when new machines are added, when bugs are detected, or when it is decided that applications are too old and are no more maintainable.

As depicted in Figure 1, at the workshop level, we observe a continuous stream of products following assembly lines from specific *entry points* to *exit points*, i.e., where products go to reach the next step of the manufacturing process. Some factories produce over 30,000 tires a day, resulting in thousands of production events at the CIM2 level, which are collected and persisted in databases.

Production systems are tested when they are set up and every time they are updated with new applications, parameters, devices, etc. We do not focus on the device level here (CIM1), but on the CIM2 level (although physical devices are tested too). For readability, when we refer to production systems in the remainder of the paper, we actually focus on the software of the CIM2 level, which acquires and sends production events to the devices.

For testing a production system, Michelin engineers firstly build simulations by mocking most of the devices. Then, they run hundreds of scenarios composed of production events, collect all the observable production events and manually inspect them to detect abnormal behaviours. As simulations are not sufficient to run all the possible scenarios, production events are again collected when the system is running, and events are scrutinised every time an issue is detected. This manual testing process can be followed for a long period time, depending on the modifications made on the system (up to 6 months). Michelin wished to partially automate this phase to:

- quicker detect potential regressions when CIM2 applications are modified or when devices are replaced to ensure that they are interoperable with the current application versions,
- test an updated system, different from the original one and focus on its new behaviours to later seek for potential faults,
- reduce the testing delay.

The main problem faced by Michelin lies in the lack of up-to-date (and hence exact) documentation. Indeed, the lifetime of the applications deployed in their factories goes up to twenty years. During this long lifetime, applications are independently updated many times in every factory all over the world, potentially highlighting different behaviours and features. Initially, these applications are documented with models, which become outdated in the long run as models are often not modified. Furthermore, even if a lot of effort is put into standardising applications and development processes, different programming languages and frameworks are still used by development teams, making difficult to focus on a single technology. This application set appears too disparate and insufficiently documented to apply conventional testing techniques. This is why our industrial partner firstly needs a safe way to generate up to date and ”the most exact” models in the sense that they do not capture more behaviours than those possible in the real system.

In addition, Michelin engineers have need for a scalable tool since a production system produces thousands of tires a day, along with thousands of production events. When an issue is detected in a production system or when the latter is getting jammed, they are interested in getting models as quickly as possible to help them diagnose failure causes.

1.2 Related work and motivation

Model inference

Model inference can be defined as a set of methods that infer a specification by gathering and analysing system executions and concisely summarising the frequent interaction patterns as state machines that capture the system behaviour [ABL02]. These models, even if partial, can be examined by designers to complete them, to identify errors, and can be utilised for analysis, etc. Models can be generated from different kinds of data samples such as affirmative/negative answers [Ang87] , execution traces [KBP$^+$10], documentation [ZZXM11], source code [PG09], or network traces [ANV11]. After reviewing the literature, we observed that two main categories of methods emerged that we call active and passive methods. The first category contains methods interacting with systems or humans to extract knowledge, which is studied to build models. Active inference approaches repeatedly query systems or humans to collect positive or negative observations. With production systems, this active functioning may give rise to some inconvenient and should not be used. Indeed, to stimulate the system, it must be shut-down, interrupted or reset for some time. Resetting such a system is difficult and often long.

This is why we prefer focusing on the second category. It includes techniques that infer models from a given set of samples, e.g., a set of execution traces. As there is no interaction with the system to model, these techniques are said passive. Models are often constructed by representing sample sets with automata whose equivalent states are later merged. A substantial part of the papers covering this topic proposes approaches either based upon event sequence abstraction or state-based abstraction to infer models.

With event sequence abstractions, the abstraction level of the models is raised by merging the states having the same event sequences. This process stands on two main algorithms: kTail [BF72] and kBehavior [MP07]. KTail generates models from trace sets in two steps. First, it builds a Prefix Tree Acceptor, which is a tree whose edges are labelled with the event names found in traces. Then kTail transforms this tree into a Finite State Automaton (FSA) by merging every pair of states if they exhibit the same future of length k, i.e., if they have the same set of event sequences having the maximum length k, which all are accepted by the two states. kBehavior generates models from a set of traces by taking every trace one after one and by completing the FSA in such a way that it now accepts the trace. More precisely, whenever a new trace is submitted to kBehavior, it first identifies the sub-traces that are accepted by sub-automata in the current FSA (the sub-traces must have a minimal length k, otherwise they are considered too short to be relevant). Then, kBehavior extends the model with the addition of new branches that suitably connect the identified sub-automata, producing a new version of the model that accepts the entire trace. Both Algorithms were enhanced to support events combined with data values [LMP08].

The approaches, which use state-based abstraction, adopted the generation of state-based invariants to define equivalence classes of states that are combined together to form final models. The *Daikon* tool [ECGN99] were originally proposed to infer invariants composed of data values and variables found in execution traces. An invariant is a property that holds at a certain point or points in a software. An invariant generator mines the data found in traces and specific to related objects or components, and then reports properties that are *true* over the observed executions. Several works were proposed to infer models from traces produced by software components (classes) [KBP$^+$10] or source codes [YEB$^+$06].

Passive testing

Observing the behaviour of an implementation and testing if it adheres to a given user-provided specification has been referred under different names such as passive testing or runtime verification. Passive testing (and runtime verification) offers the advantage to not disturb the implementation under test by collecting observations or samples and by checking if these meet a specification or properties. With runtime verification, specifications are usually written with CTL or LTL properties, which is out of the scope of the paper. We prefer referring to [LS09]. Several works, dealing with the passive testing of protocols or components, have also been proposed over the last decade. We propose to group some of them related to our work in two different categories:

— *Invariant satisfiability:* invariants represent properties that are always true. An invariant is constructed by hand, and later checked on a set of traces collected from an implementation. This approach is very similar to runtime verification and allows the test of complex and formal properties. It gave birth to several works, e.g., [CMdO09, BCNZ05, MMC13, MMC$^+$10, AMNn12]. For instance, the passive testing method presented in [CMdO09] aims to test the satisfiability of invariants on Mobile ad-hoc network (MANET) routing protocols. Different steps are required: definition of invariants, extraction of execution traces with sniffers, verification of the invariants on the trace set. Other works focused on Component-based System Testing: in this case, passive methods are usually used for conformance or security testing. For instance, the TIPS tool [MMC$^+$10] performs an automated analysis of the captured trace sets to determine if a given set of timed extended invariants hold. As in [CMdO09], invariants are constructed from a specification and traces are collected with network sniffers. Andrès et al. presented another methodology to perform the passive testing of timed systems [AMNn12]. The paper gives two algorithms, which check whether timed invariants hold on logs recorded from an implementation under test;

— *Forward checking:* implementation reactions are observed by a tester, which detects incorrect behaviours by covering a model every time a new event is collected [LCH$^+$06, UX07]. For instance, Lee et al. proposed some methods dedicated to wired protocols in [LCH$^+$06]. Protocols are modelled with Event-driven Extended Finite State Machines (EEFSM), which are specialised FSM composed of variables and constraints over these variables. Several algorithms on the EEFSM model are provided as well as their applications on the protocols TCP and Open Shortest Path First (OSPF). Algorithms check whether partial traces, composed of actions and parameters, meet an EEFSM on-the-fly: every time a new event is received, new configurations are computed from an existing configuration set. A configuration represents a tuple gathering a reached state of the model and a set of assignments and guards modelling the variable states. When no configuration can be computed, an error is detected.

1.2.1 Key observations and motivations

After having studied both research fields and discussing with our industrial partner, it quickly turned out that passive inference appeared to be a good solution. Nevertheless, the proposed approaches still reflect some limitations that plague the final quality of the models. Most of the model inference techniques are over-approximating system behaviours, i.e., models often admit more behaviours than those observed. When models are employed for verification or testing, over-approximated models often bring false positives. Indeed, infeasible test cases can be generated from these models, i.e., test cases that cannot be executed or that expect observations the system cannot produce. Such test cases give incorrect verdicts. We observed that over-approximation often comes from the state merging process, which combines the states having the same properties. Specifically, this issue comes from the state equivalence relations (k-future, congruence equivalences, etc.) that raise the abstraction level of the model. Furthermore, most of the above algorithms have a complexity polynomial in time with respect to the model size or require a polynomial number of queries. However, we observed that only few methods [YEB$^+$06, PG09] focus on scalability and propose algorithms that can take huge event sets as inputs and still infer models quickly. To do so, these use a context-specific state merging process.

Based on these observations, we chose to devise a (context-specific) model inference approach that aims at recovering models as Symbolic Transition Systems (STS [FTW05]) from production event sets. As models are used for testing, we want to prevent (control) their over-generalisations. This approach is hence initially based upon trace abstraction and model compression to avoid the construction of models composed of over-approximations. During this process, we remove the information related to products, which we call *normalisation*. The originality of the model construction resides in the combination of an expert system to encode expert knowledge (given by Michelin engineers or found in documentation) and of transition systems to embrace formal tools. This means that the STS transformations and compression, called STS reduction, are defined with inference rules thanks to the STS theory, and are triggered by the same expert system. The STS reduction, is based on an event sequence abstraction. We also show that our approach is scalable: it can take millions of production events and still build models quickly thanks to our specific state merging process.

Concerning passive testing, we noticed that the above techniques assume having either complete specifications encoding all the correct behaviours, or properties. We observed that these are not specifically tailored to support partial models, which are neither complete specifications nor properties. Hereinafter, we propose two passive testing techniques to test whether a production system under test is conforming to an initial system under analysis. The first technique uses an offline mode, the second one checks conformance in an online way. These two techniques are based on specific implementation relations to define conformance.

Finally, in [SD15] and [DS15], we proposed algorithms for inferring models from industrial systems and for passively testing them (offline mode). The main differences with this paper lie in the model inference goals. In [SD15], model inference is applied to learn more abstract and over-approximated models. In contrast, models are here built for testing purposes, hence, we define particular trace sets (complete and filtered traces), we consider model normalisation and adapt the inference steps so that both offline and online passive methods reuse some formal definitions. The offline passive tester, introduced in [DS15], is revisited with new propositions on the implementation relations to prepare the proof of the algorithm soundness.

Paper organisation:
The paper is structured as follows: Section 2 recalls some definitions and notations about the STS formalism. We present the theoretical aspects of Autofunk in Section 3 and 4. The first section is related to the model generation of production systems into STSs with a view to perform testing. We also describe the practical assumptions that guided the design of Autofunk. Section 4 details the offline and online passive testers, which embody the implementation relations. Afterwards, we evaluate Autofunk on a real production system, in Section 5. We show that Autofunk can infer models from millions of production events in reasonable time. We also apply offline passive testing on different kinds of system under test. We traditionally conclude in Section 6.

2 Model Definition and Notations

In this paper, we focus on models called Symbolic Transition Systems (STS) ([FTW05]) to represent how production systems behave. A STS is a kind of symbolic automaton compound of states called *locations*. Transitions between locations are labelled with events including a label and parameters. One can also find guards and variable assignments.

Definition 1 (Variable assignment) *We assume that there exist a finite 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 $\alpha : Y \rightarrow D$.*
We denote D_Y the assignment set over Y. For $y \in Y$, $\alpha(y)$ returns the assignment of the y variable. We also denote id_Y the identity assignment over Y.

For instance, $\alpha = \{x := 1, y := 3\}$ is a variable assignment of $D^{\{x,y\}}$. $\alpha(x) = \{x := 1\}$ is the variable assignment related to the variable x.

Definition 2 (STS) *A Symbolic Transition System (STS) is a tuple $(L, l0, V, V0, I, \Lambda, \rightarrow)$, where:*

- *L is the finite location set, with l_0 being the initial one,*
- *V is the finite set of internal variables, I is the finite set of parameters,*
- *V_0 is the initial condition, a predicate with variables in V,*
- *Λ is the finite set of symbolic events $a(p)$, with $p = \langle p_1, ..., p_k \rangle$ a finite tuple of parameters in $I^k (k \in \mathbb{N})$,*
- *$\rightarrow$ is the finite transition set. A transition $(l_1, l_2, a(p), G, A)$, from the location $l_1 \in L$ to $l_2 \in L$, also denoted $l_1 \xrightarrow{a(p),G,A} l_2$, is labelled by:*
  - *an event $a(p) \in \Lambda$, with $p = \langle p_1, ..., p_k \rangle$,*
  - *a guard G, which is a predicate with variables in $V \cup \{p_1, ..., p_k\}$ that restricts the firing of the transition. For simplicity (and since this is sufficient in our context), we restrict to the guards of the form:*
    $G \rightarrow PG \mid G\ op\ G$,
    $PG \rightarrow Variable == Constant$,
    $op \rightarrow \wedge \mid \vee$,
  - *internal variables are updated with the assignment function A of the form $(x := A_x)_{x \in V}$, A_x is an expression over $V \cup \{p_1, ..., p_k\}$.*

Below, we define some notations on STSs. In particular, we use the notion of projection on guards, denoted $Proj_X(G)$, which aims to only keep the equalities of G using the variables of the set X. A projection comes down to eliminating the equalities using the variables in $(I \cup V) \backslash X$. For the definition of variable elimination (a.k.a. forgetting), we refer to [LLM03].

Definition 3 *Given a STS $\mathcal{S} = (L, l0, V, V0, I, \Lambda, \rightarrow)$ and l, $l' \in L$, we use the following notations:*

- $l_1 \xrightarrow{(a_1,G_1,A_1)...(a_n,G_n,A_n)} l_{n+1} =_{def} \exists l_i, l_{i+1}, a_i, G_i, A_i$ $(1 \leq i \leq n) : l_1 \xrightarrow{a_1,G_1,A_1} l_2, ..., l_n \xrightarrow{a_n,G_n,A_n} l_{n+1}$;

— *$l \nrightarrow =_{def} \neg\exists l', a(p), G, A : l \xrightarrow{a(p),G,A} l'$. We say that l is a deadlock location;*
— *$l \xrightarrow{a(p),G} l' =_{def} \exists, a(p), G, A = id_V : l \xrightarrow{a(p),G,id_V} l'$;*
— *$Proj_X(G)$, the projection of the guard G over the variable set $X \subseteq I \cup V$, which eliminates from G the equalities on the variables of $(I \cup V) \backslash X$.*

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). The LTS semantics can be assimilated to a valued automaton, which is often infinite: the LTS states are labelled by internal variable assignments, and transitions are labelled by valued events, composed of parameter assignments. The semantics of a STS $\mathcal{S} = (L, l0, V, V0, I, \Lambda, \rightarrow)$ is the LTS $||\mathcal{S}|| = (Q, q_0, \sum, \rightarrow)$ 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 events, and $\rightarrow$ is the transition relation.

The complete definition of the relation between a STS and its LTS semantics is given in [FTW05]. For simplicity, we only give its insight in this paper. For a STS transition $l \xrightarrow{a(p),G,A} l'$, we have LTS transitions of the form $(l, v) \xrightarrow{a(p),\alpha} (l', v')$ 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 $A(v \cup \alpha)$.

From the LTS semantics, one can derive runs and traces, which reflect the concrete functioning of the system modelled with $\mathcal{S}$ and $||\mathcal{S}||$:

Definition 4 (Runs and traces) *Let $\mathcal{S}$ be a STS and $||\mathcal{S}|| = (Q, q_0, \sum, \rightarrow)$ be its LTS semantics.*

— *A run $q_0 a_0(\alpha_0) ... q_{k-1} a_{k-1}(\alpha_{k-1}) q_k$ is an alternate sequence of states and valued events such that: $\exists q_i, q_{i+1}, a_i, \alpha_i (0 \leq i \leq k-1) : q_0 \xrightarrow{a_0(\alpha_0)} q_1 ... q_{k-1} \xrightarrow{a_k(\alpha_k)} q_k \in \rightarrow^*$.*
$Runs(\mathcal{S}) = Runs(||\mathcal{S}||)$ is the set of runs found in $||\mathcal{S}||$. $Runs_F(\mathcal{S})$ is the set of runs that end in a state q of $F \times D_V$ with $F \subseteq L$.
— *the trace of a run $r = q_0 a_0(\alpha_0) ... q_{k-1} a_{k-1}(\alpha_{k-1}) q_k$, denoted $Trace(r)$ is the sequence $a_0(\alpha_0) ... a_{k-1}(\alpha_{k-1})$.*
$Traces_F(\mathcal{S}) = Traces_F(||\mathcal{S}||) = \{Trace(r) \mid r \in Runs_F(\mathcal{S})\}$.

3 Model generation of production systems

After having studied the above background, we came up to the conclusion that, in order to target the largest part of all the Michelin CIM Level 2 applications, the most appropriate solution would be to take advantage of the production events exchanged among devices. Indeed, these events hold a lot of interesting information that can be interpreted to understand how a whole industrial system behaves. These events are exchanged over a network layer and stored into a logging system that guarantee (synchronous) ordering and delivery. This context leads to some assumptions that have been considered to design our framework:

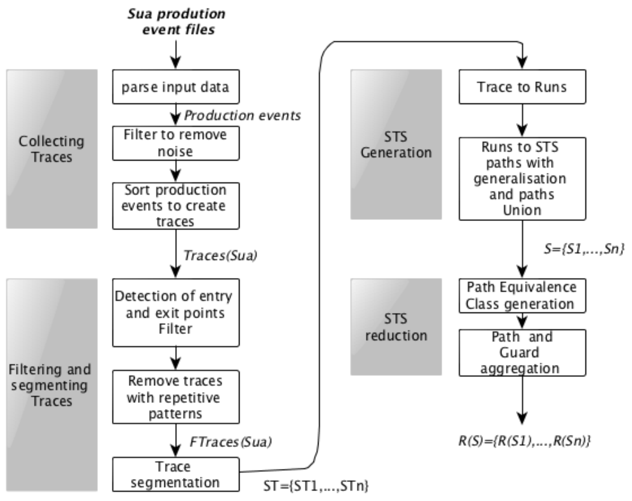
— Black-box systems: production systems are seen as black-boxes from which production events can be passively collected at the CIM2 level. Such systems are compound of assembly lines fragmented into several devices and sensors. A production system has one or more entry and one or more exit points. In the remainder of the paper, we assume that a production system can be modelled by an (unknown) LTS, denoted *Sua* for System under analysis, or *Sut* for System under test. This assumption allows to later write definitions with *Sua* or *Sut*;

— Production events: a production event of the form $a(\alpha)$ includes a distinctive label a along with a parameter assignment α. Two production events $a(\alpha_1)$ and $a(\alpha_2)$ having the same label a must own assignments over the same parameter set. Network protocols guarantee synchronous communications and the event ordering with timestamps assigned to a parameter denoted $time$, which takes values from a global clock. A specific parameter, denoted *point*, stores the physical location of devices;

— Traces identification: execution traces are sequences of production events $a_1(\alpha_1)$... $a_k(\alpha_k)$ identified by a specific parameter that is included in all the event assignments of a trace. In this paper, this identifier is denoted with *pid* and identifies products, e.g., tires at Michelin. At the same time, we cannot have two different traces (for two products) having the same pid;

— Event delivery: network protocols guarantee the event delivery and an assembly line is conceived in such a way that it does not have deadlock states except when a product exits the line.

In the remainder of this section, we describe the model generation of production systems, illustrated in Figure 2. The model generation stage is conceived upon the notion of expert system adopting a forward chaining. Such a system separates the data (events, traces, transitions, models), from the reasoning: the former are expressed with knowledge bases, a.k.a. facts while the latter is defined with inference rules that are applied on the facts. Our framework Autofunk relies upon two kinds of inference rules: on the one hand, we have rules capturing the



Sua prodution event files
Collecting Traces
parse input data
Production events
Filter to remove noise
Sort production events to create traces
Traces(Sua)
Filtering and segmenting Traces
Detection of entry and exit points Filter
Remove traces with repetitive patterns
FTraces(Sua)
Trace segmentation
ST={ST1,...,STn}
STS Generation
Trace to Runs
Runs to STS paths with generalisation and paths Union
S={S1,...,Sn}
STS reduction
Path Equivalence Class generation
Path and Guard aggregation
R(S)={R(S1),...,R(Sn)}

Figure 2 – Model inference detailed steps

knowledge of a human expert or found in documentation. On the other hand, the remaining rules relate to STS transformations. The use of an expert system is a strong originality of our approach. And the possibility to change rules for matching other kinds of systems is a manifest benefit. Nevertheless, such rules have to be triggered a finite number of times to ensure the model inference termination ans must always give identical results with the same bases of facts (soundness). To reach that goal, we assume that inference rules used by our framework meet these hypotheses:

Model inference assumptions:

1. inference rules are Modus Ponens (simple implications that lead to sound facts if the original facts are true);
2. the facts in knowledge bases have an Horn form.

These assumptions guarantee that the resolution of the inference rules (based on Modus Ponens) with a knowledge base (in Horn form) is sound and complete. These assumptions will be used to discuss about the soundness and complexity of the model generation. We are now ready to present the steps depicted in Figure 2.

3.1 Trace collecting and filtering

As depicted in Figure 2, Autofunk starts by collecting input data from files gathering the events passing through the network of the system under analysis *Sua*. In this way, it is not disrupted since production events are collected by its

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

Figure 3 – Production event examples

```
rule "Remove INFO events"
when:
$a: ValuedEvent(assignment.valueOf("type") == TYPE_INFO)
then
retract($a)
end
```

Figure 4 – Inference rules example for filtering

logging system. A production message is mainly compound of a label along with kinds of variable assignments. An example of messages is given in Figure 3. It includes simplified production events similar to those extracted from the Michelin logging system. INFO, 7011 and 17021 are labels that are accompanied with assignments of variables, e.g., *nsys* that indicates an industrial device number, or *point* that gives the device position in a workshop. With real events, there are around 20 parameters.

Production events are formatted into a knowledge base of valued events of the form $a(\alpha)$, with a a label and α a parameter assignment. Thereafter, this base is filtered by means of inference rules of the form: *When* $a(\alpha)$*, condition on* $a(\alpha)$*, Then retract(*$a(\alpha)$*).* Figure 4 shows a rule example applied on Michelin systems. This rule is written with the *Drools* [3] formalism. Drools is a rule-based expert system where knowledge bases are expressed with Java objects. Straightforwardly, this rule removes the production events that hold the *INFO* label. Indeed, human experts confirmed us that it does not worth keeping this kind of event since they do not express a behaviour part.

From this filtered valued event base, we reconstruct the corresponding traces by linking together the events $a(\alpha)$ holding the same trace identifier *pid*, and by ordering them with respect to their timestamps assignments. We call the resulting trace set $Traces(Sua)$:

Definition 5 **($Traces(Sua)$)** *Given a system under analysis Sua,* $Traces(Sua)$ *denotes its trace set.*

$Traces(Sua)$ *includes finite traces i.e., finite sequences of production events of*

3. http://www.drools.org/

the form $a_1(\alpha_1)....a_k(\alpha_k)$ *such that:* $\exists$ *unique* $v \in D, \forall \alpha_i(1 \leq i \leq k) : \alpha_i(pid) = (pid := v)$.

The trace set $Traces(Sua)$ is then filtered two times. Firstly, it is analysed for recovering the different entry and exit points of the production lines of *Sua*. The entry and exit point sets are denoted $Entry(Sua)$ and $Exit(Sua)$. A trace of $Traces(Sua)$, starting with an assignment that does not include an entry point of $Entry(Sua)$, is considered as incomplete and removed.

Autofunk relies on a machine learning technique to compute $Entry(Sua)$ and $Exit(Sua)$. Intuitively, every trace is analysed to collect on the first and last production events, the assignments on the variable point. Indeed, this variable captures the product physical locations. We then obtain two raw sets $Entry(Sua)$ and $Exit(Sua)$. In order to determine the entry and exit points of *Sua*, we rely on the well-known *K-means* clustering method [WH79], a machine learning algorithm, which is both fast and efficient, and does not need to be trained before being effectively used (that is called unsupervised learning, and it is well-known in the machine learning field). K-means clustering aims to partition n observations into K clusters. Here, observations are represented by the assignments of the variable *point* in $Entry(Sua)$ or $Exit(Sua)$. As we want to group the outliers together and leave the others in another cluster, we use $K = 2$.

This method is suitable for production systems because they hold physical assembly lines where almost all the products flow from real entry and exit points. Hence, on sufficiently large production event sets, we should observe high ratios of products moving from the same (entry) points to the same exit points. Using K-means helps automate this step, but, as explained in the evaluation of Autofunk (Section 5), when there are only few production events, we need an expert to provide them.

Once the entry points are learnt, we filter $Traces(Sua)$ and keep in $CTraces(Sua)$ the *complete traces*, i.e., those that are obtained from one entry point of *Sua* and that lead to a deadlock state of *Sua*:

Definition 6 (Complete traces) *Let* $Traces(Sua)$ *be the trace set of the production system under analysis* $Sua = (Q, q_0, \Sigma, \rightarrow)$*, and* $Entry(Sua)$ *be its (physical) entry point set. Let* $Qd \subseteq Q$ *be the set of states* q *such that* $q \not\rightarrow$.
$CTraces(Sua) =_{def} \{a_1(\alpha_1)...a_n(\alpha_n) \in Traces_{Qd}(Sua) \mid \alpha_1(point) \in Entry(Sua)\}$

$CTraces(Sua)$ may still include traces that are not relevant to recover a model describing the normal functioning of *Sua*. Indeed, it often appeared during our experimentations that some traces represent abnormal product lifes in the production system. Some unfinished products are indeed abruptly removed for different purposes. We have chosen to remove such traces. This step is done by only keeping the traces whose last assignments on the variable *point* give an exit point of *Sua* in $Exit(Sua)$. In addition, we observed that the physical devices may repetitively query other devices, depending on the use of the load of the devices. The resulting traces express this repetitive behaviour with repetitive sequences of events that we call *repetitive patterns* of events. We have chosen to remove the traces including repetitive patterns in order to reduce the final model size. The detection of repetitive patterns in traces is done by removing the variable assignments related to product since these assignments are always different among the traces. We call this step *normalisation*. In our case, the variables designed as specific to product are *pid* and *time*. We denote the trace normalisation with the operator *Norm*. Autofunk tries to detect repetitive patterns in this way: if it finds a trace t of the form $t_1p...pt_2$ and another trace $t'_1p't'_2$ such that $Norm(t_1) = Norm(t'_1)$, $Norm(t_2) = Norm(t'_2)$, $Norm(p) = Norm(p')$, then $Norm(p)$ is a designated as a repetitive pattern and t is removed from $CTraces(Sua)$ since we suppose that t does not express a new and interesting behaviour. Traces are completely removed rather than cleaning them (i.e. deleting the repetitive patterns) to prevent from encoding behaviours not observed from *Sua*.

A repetitive pattern detection algorithm is provided in [SD15]. Here, we prefer giving the definition of the *filtered trace* set, denoted $FTraces(Sua)$:

Definition 7 (Filtered traces) *Let $Traces(Sua)$ be the trace set of the production system $Sua = (Q, q_0, \Sigma, \rightarrow)$, $Entry(Sua)$ and $Exit(Sua)$, be its sets of entry and exit points respectively.*
Let $a_1(\alpha_1)...a_n(\alpha_n) \in CTraces(Sua)$, X be the variable set and $Y \subseteq X$ the variables related to products.

1. $Norm(a_1(\alpha_1)...a_n(\alpha_n)) =_{def} a_1(\alpha'_1)...a_n(\alpha'_n)$ *with* $\alpha'_i = \alpha_i(X \setminus Y)(1 \leq i \leq n)$
2. $R =_{def} \{t_1p...pt_2 \in CTraces(Sua) \mid t'_1pt'_2 \in CTraces(Sua),\ Norm(t_1) = Norm(t'_1),\ Norm(t_2) = Norm(t'_2),\ Norm(p) = Norm(p')\}$
3. $Pattern(Sua) =_{def} \{Norm(p) \mid t_1p...pt_2 \in R\}$
4. $FTraces(Sua) =_{def} \{a_1(\alpha_1)...a_n(\alpha_n) \in CTraces(Sua) \mid \alpha_n(point) \in Exit(Sua)\} \setminus R$

Traces are assembled and filtered to achieve a set $FTraces(Sua)$ that is not over-approximated, trace inclusion with $Traces(Sua)$ is preserved:

Proposition 1 $FTraces(Sua) \subseteq CTraces(Sua) \subseteq Traces(Sua)$

3.2 STS generation

This step aims at building a STS $\mathcal{S}$ from $FTraces(Sua)$ in such a way that $\mathcal{S}$ only encodes the behaviours found in $FTraces(Sua)$. The STS generation is incrementally done by lifting traces into runs, and runs into STS paths. The translation of $FTraces(Sua)$ into a run set denoted *Runs* is done by completing traces with states. Each run starts by the same initial state $(l0, v_\emptyset)$ with $v_\emptyset$ an empty condition. Then, new states are injected after each valued event. *Runs* is defined as :

Definition 8 (Structured Runs) *Let $FTraces(Sua)$ be a trace set obtained from Sua. We denote Runs the set of runs derived from $FTraces(Sua)$ with the following inference rule:*

$$\frac{t_{id}=(a_1,\alpha_1)\dots(a_k,\alpha_k)\in FTraces(Sua), \alpha_1(pid)=(pid:=id)}{(l0,v_\emptyset)a_1(\alpha_1)(l_{id1},v_\emptyset)\dots(l_{idk-1},v_\emptyset)a_k(\alpha_k)(l_{idk},v_\emptyset)\in Runs}$$

The runs of *Runs* have states that are unique except for the initial state $(l0, v_\emptyset)$. 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 union. The resulting STS forms a tree compound of branches starting from the location $l0$. Parameters and guards are extracted from the assignments found in valued events. In this step, the events of the runs are normalised with the $Norm$ operator to remove the parameter assignments related to products (assignment of the variables pid and $time$ in our context). We obtain more generalised STSs, which express the behaviours of *Sua*, independently of the manufactured products.

Definition 9 *Given a run set Runs and Y the set of variables related to products, $\mathcal{S} = (L_\mathcal{S}, l0_\mathcal{S}, V_\mathcal{S}, V0_\mathcal{S}, I_\mathcal{S}, \Lambda_\mathcal{S}, \rightarrow_\mathcal{S})$ is the STS expressing the (generalised) behaviours found in Runs such that:*

— $L_\mathcal{S} = \{l \mid \exists r \in Runs, (l, v_\emptyset)$ *is a state of* $r\}$,

— $l0_\mathcal{S} = l0$ *is the initial location such that* $\forall r \in Runs$, *r starts with* $(l0, v_\emptyset)$,

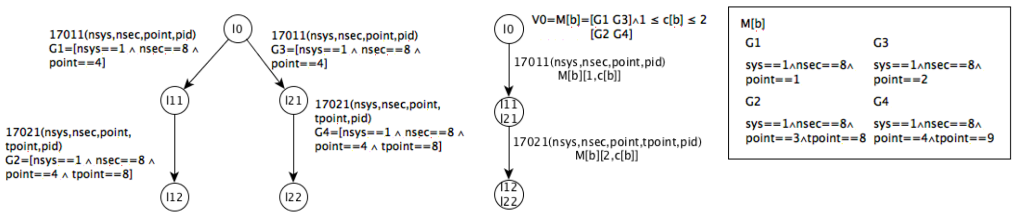
— $V_\mathcal{S} = \emptyset$, $V0_\mathcal{S} = v_\emptyset$,

— $\rightarrow_\mathcal{S}$ *and* $\Lambda_\mathcal{S}$ *are defined by the following inference rule applied on every element* $r \in Runs$:

$$\frac{(l,v_\emptyset)a(\alpha)(l',v_\emptyset)\in r, a(\alpha')=Norm_Y(a(\alpha)), p=\{x \mid \alpha\in D^X, x\in X\},\quad G=\bigwedge_{(x:=v)\in\alpha'} x==v}{l\xrightarrow{a(p),G}_\mathcal{S} l'}$$

This first "raw" STS has a tree form, one branch exactly modelling one trace of $FTraces(Sua)$. Figure 5(a) illustrates the STS $\mathcal{S}$ obtained from the production events of Figure 3. We have STS events, each associated with its own parameters. Transitions are labelled with guards directly derived from parameter assignments. It is manifest that this STS meets our initial goal in terms of trace inclusion with $FTraces(Sua)$:

Proposition 2


l0
17011(nsys,nsec,point,pid)
G1=[nsys==1 ∧ nsec==8 ∧ point==4]
17011(nsys,nsec,point,pid)
G3=[nsys==1 ∧ nsec==8 ∧ point==4]
l11
l21
17021(nsys,nsec,point,tpoint,pid)
G2=[nsys==1 ∧ nsec==8 ∧ point==4 ∧ tpoint==8]
17021(nsys,nsec,point,tpoint,pid)
G4=[nsys==1 ∧ nsec==8 ∧ point==4 ∧ tpoint==8]
l12
l22
l0
V0=M[b]=[G1 G3]∧1 ≤ c[b] ≤ 2
[G2 G4]
17011(nsys,nsec,point,pid)
M[b][1,c[b]]
l11
l21
17021(nsys,nsec,point,tpoint,pid)
M[b][2,c[b]]
l12
l22
M[b]
G1
G3
sys==1∧nsec==8∧ point==1
sys==1∧nsec==8∧ point==2
G2
G4
sys==1∧nsec==8∧ point==3∧tpoint==8
sys==1∧nsec==8∧ point==4∧tpoint==9

(a) First inferred STS example (b) Reduced STS example

Figure 5

$Normy_Y(FTraces(Sua)) = Normy_Y(Traces_{Ld}(\mathcal{S}))$ with $Ld \subseteq L_{\mathcal{S}}$ the deadlock location set of $\mathcal{S}$.

3.3 STS reduction

The above STS $\mathcal{S}$ is most likely too large for being analysed in a scalable manner. Yet, production systems are often conceived with finite physical paths and build product with finite steps. As a consequence, the STS $\mathcal{S}$ should contain paths capturing the same sequences of events (without necessarily the same parameter assignments) and could be minimised. Initially, we studied the state merging techniques used with passive inference methods, briefly presented in Section 1.2. As stated earlier, these techniques build over-approximated models though, which may lead to false positives when used for testing.

Consequently, we have chosen to add in our framework a context-specific and lightweight STS reduction technique, which aims at reducing a STS $\mathcal{S}$ into another STS denoted $R(\mathcal{S})$. This technique can be assimilated to a formal classification method (data mining) whose role consists in combining STS paths that have the same sequences of events. The resulting STS $R(\mathcal{S})$ still keeps its tree structure. In addition, when STS paths are merged, their guards are wrapped into matrices. The guard compression is performed in such a way that trace equivalence between $\mathcal{S}$ and $R(\mathcal{S})$ is preserved.

Given a STS $\mathcal{S}$, its paths are firstly adapted to express sequences of guards in a vector form. 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 $\mathcal{S} =< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}}>$ be a STS inferred from Sua. We denote Mat the STS operator that consists in expressing guards of STS paths in a vector form.*

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

— $L_{Mat(\mathcal{S})} = L_{\mathcal{S}}$, $l0_{Mat(\mathcal{S})} = l0_{\mathcal{S}}$, $I_{Mat(\mathcal{S})} = I_{\mathcal{S}}$, $\Lambda_{Mat(\mathcal{S})} = \Lambda_{\mathcal{S}}$,

— $V_{Mat(\mathcal{S})}$, $V0_{Mat(\mathcal{S})}$ and $\rightarrow_{Mat(\mathcal{S})}$ are given by the following rule:

$$\boxed{\dfrac{b = l0 \xrightarrow{(a_1(p_1),G_1)\dots(a_n(p_n),G_n)}_{\mathcal{S}} l_n}{V0_{Mat(\mathcal{S})} = V0_{Mat(\mathcal{S})} \wedge M_b == [G_1,\dots,G_n] \quad l0_{Mat(\mathcal{S})} \xrightarrow{(a_1(p_1),M_b[1])\dots(a_n(p_n),M_b[n])}_{Mat(\mathcal{S})} l_n}}$$

Given a branch $b \in \rightarrow_{Mat(\mathcal{S})}$, we also denote $Mat(b) = M$ the vector storing guards of b.

The STS paths having the same sequences of events can now be assembled. These paths are grouped in path equivalence classes:

Definition 11 (STS path equivalence class) *Let $\mathcal{S} = < L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}} >$ be a STS obtained from Sua (and having a tree structure).*

$[b]$ denotes the equivalence classes of $\mathcal{S}$ paths such that:
$[b] = \{b' = l0_{\mathcal{S}} \xrightarrow{(a'_1(p'_1),G'_1),\dots(a'_n(p'_n),G'_n)} l'_n \mid b = l0_{\mathcal{S}} \xrightarrow{(a_1(p_1),G_1)\dots(a_n(p_n),G_n)} l, a_i(p_i) = a'_i(p'_i)\ (1 \leq i \leq n)\}$

The reduced STS $R(\mathcal{S})$ of $\mathcal{S}$ is obtained by concatenating the paths of an equivalence class $[b]$ found in $Mat(\mathcal{S})$ into one path. For an equivalence class $[b]$, the vectors found in the paths of $[b]$ are joined into the matrix $M_{[b]}$. A vector, which collects an ordered sequence of guards of one path of $\mathcal{S}$, is placed in one of the matrix columns. Furthermore, we take advantage of this step to label all the final locations of $R(\mathcal{S})$ with "Pass". We denote these locations as verdict locations and gather them in the set $Pass \subseteq L_R(\mathcal{S}))$. $R(\mathcal{S})$ is defined as follows:

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

— $I_R = I_{\mathcal{S}}$, $\Lambda_R = \Lambda_{\mathcal{S}}$,

— $L_R, l0_R, V_R, V0_R$ and $\rightarrow_R$ are given by the following rule:

$$\boxed{\dfrac{[b] = \{b_1,\dots,b_m\} \text{ with } b_i = l0_{\mathcal{S}} \xrightarrow{(a_1(p_1),G_{i1})\dots(a_n(p_n),G_{in})}_{Mat(\mathcal{S})} l_{in}}{V0_R = V0_R \wedge M_{[b]} == [Mat(b_1),\dots,Mat(b_m)] \wedge (1 \leq c_{[b]} \leq m), \quad l0_R \xrightarrow{(a_1(p_1),M_{[b]}[1,c_{[b]}])\dots(a_n(p_n),M_{[b]}[n,c_{[b]}])}_R (Pass)}}$$

We also denote $Entry(R(\mathcal{S})) = Entry(Sua)$, $Exit(R(\mathcal{S})) = Exit(Sua)$, $Pattern(R(\mathcal{S})) = Pattern(Sua)$.

$R(\mathcal{S})$ is a STS whose paths are composed of guards, which refer to a matrix $n \times m$ denoted $M_{[b]}$, with $[b]$ an equivalence class of paths of $Mat(\mathcal{S})$. The choice of the column in a matrix depends on a new variable $c_{[b]}$, which takes a value between 1 and m within the initial condition $V0_R$.

The STS depicted in Figure 5(a) has two paths that can be combined since they have the same sequence of labels. The guards are placed into two vectors

$M1 = [G1\ G2]$ and $M2 = [G3\ G4]$. These are combined into the matrix $M_{[b]}$, as illustrated in Figure 5(b). By means of the initial condition $V0$ and the variable $c_{[b]}$, the two initial paths can be easily recovered with the assignments $c_{[b]} := 1$ or $c_{[b]} := 2$.

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

Proposition 3 $Traces(R(\mathcal{S})) = Traces(\mathcal{S})$.

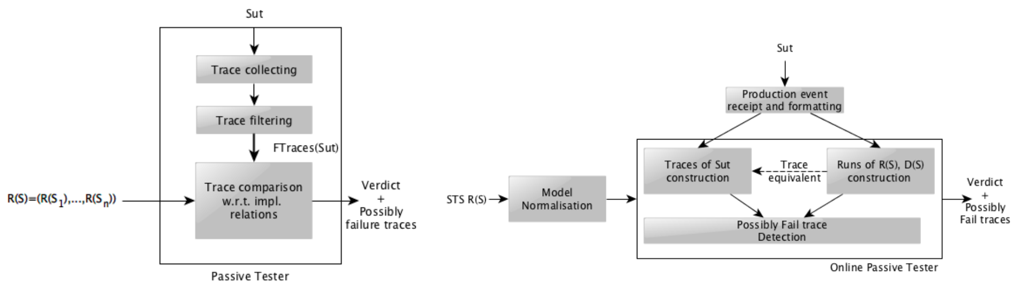
3.4 Soundness complexity and termination of the model generation

The soundness and the termination of the model inference stage mainly depend on the inference rules and the knowledge bases. The latter hold by (positive) facts (Events, Traces, Transitions, Models) that have an Horn form. We have assumed that the inference rules are Modus Ponens. Therefore, the resolution of the inference rules and the inference of knowledge bases is sound and complete (is achieved in finite time). From this fact and by considering the propositions 1, 2 and 3, we can state the soundness of the model inference stage with:

Proposition 4

1. $FTraces(Sua) \subseteq Traces(Sua)$ *and* $FTraces(Sua) \subseteq Traces_{Pass}(R(\mathcal{S}))$
2. $Norm_Y(Traces_{Pass}(R(\mathcal{S}))) = Norm_Y(FTraces(Sua)) \subseteq Norm_Y(Traces(Sua))$

The whole complexity of this step is polynomial in time and is proportional to $O(t + m(t^2 + t + k + log(m)))$ with m production events, t traces in $Traces(Sua)$, k inference rules for filtering (worst case). The complexity to filter m production events with k rules is $O(mk)$. The remaining ones are sorted in $O(m * log(m))$ (with the Java $Collection.sort()$). We mine $Traces(Sua)$ with K-means, whose complexity is proportional to $O(2t)$ with t the number of traces in $Traces(Sua)$. The complexity to extract $CTraces(Sua)$ from $Traces(Sua)$ is $O(t)$, since only the first and last valued events of the traces are read. The algorithm, which builds $FTraces(Sua)$, is proportional to $O(t^2m)$. Traces are lifted to the STS level by covering them with a complexity proportional to $O(m)$. The STS reduction complexity is $O(m + tm)$. Indeed, path equivalence classes are generated with a hash function, called on event sequences ($O(m)$). The paths of a class are grouped with a rule covering all the paths together ($O(tm)$).


Sut
Trace collecting
Trace filtering
FTraces(Sut)
R(S)=(R(S1),...,R(Sn))
Trace comparison w.r.t. impl. relations
Verdict + Possibly failure traces
Passive Tester
Sut
Production event receipt and formatting
STS R(S)
Model Normalisation
Traces of Sut construction
Trace equivalent
Runs of R(S), D(S) construction
Possibly Fail trace Detection
Verdict + Possibly Fail traces
Online Passive Tester

(a) Offline Passive Testing

(b) Online Passive Testing

Figure 6

4 Passive testing

The second part of *Autofunk* relates to the testing phase. It takes a model $R(\mathcal{S})$ inferred from the reference system *Sua* and production events collected from another system under test *Sut*. $R(\mathcal{S})$ expresses some possible behaviours that should happen, which are encoded by the traces $Traces_{Pass}(R(\mathcal{S}))$. We refer to these traces as pass traces. We call the others, possible failure traces because $R(\mathcal{S})$ is a partial model.

Sut can refer to the same production system as *Sua*, which has been updated. In this case, testing comes down to checking that changes have not introduced new faults (regression testing). *Sut* can also be a new system in a new factory, which should behave as *Sua*.

In this section, we introduce two passive testing methods, the first one being offline, and the second one using an online mode.

4.1 Offline passive testing

The offline mode is outlined in Figure 6(a). A set of production events has been collected from *Sut*, in the same way as previously with *Sua*. These are grouped into traces to form the trace set $Traces(Sut)$. The latter is filtered as described in Section 3.1 to obtain a set of filtered traces $FTraces(Sut)$. A passive tester is finally called to check whether *Sut* conforms to $R(\mathcal{S})$ with $FTraces(Sut)$. Conformance is defined with implementation relations.

4.1.1 Implementation relations

We define conformance with a first implementation relation to check whether any filtered trace of *Sut* matches a behaviour captured by $R(\mathcal{S})$. This first implementation relation, denoted with $\leq_{ft}$ (ft for filtered traces), is defined by:

Definition 13 (Implementation relation $\leq_{ft}$) *Let $R(\mathcal{S})$ be an inferred model of Sua and Sut be the system under test.*
$Sut \leq_{ft} R(\mathcal{S}) =_{def} FTraces(Sut) \subseteq Traces_{Pass}(R(\mathcal{S}))$

The model $R(S)$ can be under-approximated and may not include some correct behaviours. If an execution of *Sut* is not captured by $R(\mathcal{S})$, one cannot conclude that *Sut* is a faulty implementation. Michelin engineers hence wished the definition of a second relation "less strict on the parameters on condition that these parameters could be found inside the model". This implementation relation must be considered as a complementary relation of $\leq_{ft}$, which is fully useful when $\leq_{ft}$ returns a possible failure trace of *Sut*. The second relation aims to point out whether this trace might reflect a realistic scenario, composed of a correct sequence of events completed of parameters found in other traces of the model $R(\mathcal{S})$. This helps classify possible failure traces in risk importance. In the Michelin context, when a trace of *Sut* is a possible failure trace for both relations, the likelihood of failure detection is the highest.

This relation, denoted $\leq_{mft}$ (with mft for multiple filtered traces), defines that an implementation *Sut* is correct iff its filtered traces $a_1(\alpha_1)...a_n(\alpha_n)$ can be found in several traces of $Traces_{Pass}(R(\mathcal{S}))$ having the same sequence of labels $a_1...a_n$. The implementation relation $\leq_{mft}$, is defined by:

Definition 14 (Implementation relation $\leq_{mft}$) *Let $R(\mathcal{S})$ be an inferred model of Sua and Sut be the system under test.*
$Sut \leq_{mft} R(\mathcal{S}) =_{def} \forall t = a_1(\alpha_1)...a_n(\alpha_n) \in FTraces(Sut), \forall \alpha_j(x)_{(1 \leq j \leq n)}, \exists t' \in Traces_{Pass}(R(\mathcal{S})) : t' = a_1(\alpha'_1)...a_n(\alpha'_n)$ *and* $\alpha'_j(x) = \alpha_j(x)$.

In the following, we propose to rewrite this relation in a simpler form. Indeed, all the traces of $R(\mathcal{S})$ having the same sequence of labels can be found in one equivalence class $[b]$ (Definition 11), and in one path of the reduced STS $R(\mathcal{S})$ (in which every equivalence class $[b]$ is reduced into one STS path). We remind that a sequence of guards found in a path of an equivalent class $[b]$ is stored into one column of the matrix $M_{[b]}$. Each variable assignment $\alpha_j(x)$ should now satisfies one of the guards of the matrix line j in $M_{[b]}[j,*]$. The implementation relation $\leq_{mft}$ can be reformulated as:

Proposition 5 *Sut* $\leq_{mft} R(\mathcal{S})$ *iff* $\forall t = a_1(\alpha_1) \ldots a_n(\alpha_n) \in FTraces(Sut), \exists b = l0_{R(\mathcal{S})} \xrightarrow{(a_1(p_1),M_{[b]}[1,c_{[b]}]),\ldots,(a_n(p_n),M_{[b]}[n,c_{[b]}])} Pass$ *with* $(1 \leq c_{[b]} \leq m)$, $\forall \alpha_j(x)(1 \leq j \leq n) : \alpha_j(x) \models M_{[b]}[j,1] \vee \cdots \vee M_{[b]}[j,m]$.

If we take back the example of Figure 5(b), the trace $t = (17011(nsys := 1, nsec := 8, point := 1, pid := 1)$ $17021(nsys := 1, nsec := 8, point := 4, tpoint :=$

$9, pid := 1)$ is not a pass trace for the relation $\leq_{ft}$ because this trace cannot be extracted from one of the paths of the STS of Figure 5(b). If we focus on the guards and on the matrix $M_{[b]}$, the parameter assignments of the event $17011(nsys := 1, nsec := 8, point := 1, pid := 1)$ satisfy the guard G1 but those in $17021(nsys := 1, nsec := 8, point := 4, tpoint := 9, pid := 1)$ do not meet the guard G2 on account of the variables *point* and *tpoint*, which do not take the expected values. In the second column of the matrix, the first guard $G3$ does not hold with (nsys:=1,nsec:=8,point:=1,pid:=1).

With the implementation relation $\leq_{mft}$, the parameter assignments of the event $17011(nsys := 1, nsec := 8, point := 1, pid := 1)$ satisfy the guard G1 and those in $17021(nsys := 1, nsec := 8, point := 4, tpoint := 9, pid := 1)$ satisfy the guard G4. Hence, the trace t is a pass trace for the relation $\leq_{mft}$. This example shows that $\leq_{mft}$ is a weaker relation than $\leq_{ft}$ and that:

Proposition 6 *$Sut \leq_{ft} R(\mathcal{S}) \implies Sut \leq_{mft} R(\mathcal{S})$*

The proof of Proposition 6 is given in Annex.

The implementation relation $\leq_{mft}$ can be simplified again by re-writing the disjunction of guards $M_{[b]}[j,1] \vee ... \vee M_{[b]}[j,m]$, found in the matrix $M_{[b]}$. This formula is simplified by gathering the equalities $x == val$ together with disjunctions for every variable x. Such equalities are extractable by means of the the *Proj* operator (see Definition 3). We obtain one guard of the form $\bigwedge_{x \in I}(x == val_1 \vee ... \vee x == val_k)$. If we generalise this idea on all the matrices of $R(\mathcal{S})$, it becomes possible to replace matrices composed of several columns into matrices of one column of guards (vectors). We propose to transform the STS $R(\mathcal{S})$ into another STS, denoted $D(\mathcal{S})$ composed of such vectors.

Definition 15 *Let $R(\mathcal{S}) =< L_R, l0_R, V_R, V0_R, I_R, \Lambda_R, \rightarrow_R >$ be a reduced STS, inferred from Sua. We denote $D(\mathcal{S})$ the STS $< L_D, l0_D, V_D, V0_D, I_D, \Lambda_D, \rightarrow_D >$ derived from $R(\mathcal{S})$ such that:*

— *$L_D = L_R$, $l0_D = l0_R$, $I_D = I_R$, $\Lambda_D = \Lambda_R$,*

— *$V_D, V0_D$ and $\rightarrow_D$ are given by the following inference rule:*

$$\boxed{\frac{b = l0_R \xrightarrow{(a_1(p_1), M'_{[b]}[1, c'_{[b]}]) ... (a_n(p_n), M'_{[b]}[n, c'_{[b]}])}_R l_n (1 \leq c'_{[b]} \leq m) \text{ in } V0_R}{\begin{array}{c} l0_D \xrightarrow{(a_1(p_1), M_{[b]}[1]) ... (a_n(p_n), M_{[b]}[n])}_D l_n, M_{[b]}[j]_{(1 \leq j \leq n)} = \\ \bigwedge_{x \in p_j} (Proj_x(M'_{[b]}[j,1]) \vee \cdots \vee Proj_x(M'_{[b]}[j,m])) \\ V0_D = V0_D \wedge (c_{[b]} == 1) \wedge M_{[b]} \end{array}}}$$

The second implementation relation $\leq_{mft}$ can now be expressed with $D(\mathcal{S})$:

Proposition 7
$Sut \leq_{mft} R(\mathcal{S})$ iff $\forall t = a_1(\alpha_1)...a_n(\alpha_n) \in FTraces(Sut), \exists\ l0_{D(\mathcal{S})} \xrightarrow{(a_1(p_1),M_{[b]}[1]),...,(a_n(p_n),M_{[b]}[n])}$ Pass such that $\forall \alpha_j (1 \leq j \leq n), \alpha_j \models M_{[b]}[j]$.

All these transformations on $\leq_{mft}$ allow to say that $\leq_{mft}$ holds iff the traces of *Sut* also are pass traces of the model $D(\mathcal{S})$. Now, the meaning of $\leq_{mft}$ is not far from the meaning of the first relation $\leq_{ft}$, and it turns out that $\leq_{mft}$ is definable with $\leq_{ft}$:

Proposition 8 *$Sut \leq_{mft} R(\mathcal{S}) \Leftrightarrow Sut \leq_{ft} D(\mathcal{S})$.*

This proposition involves that the same passive tester algorithm can be used to check if a given system under test meets both relations $\leq_{ft}$ and $\leq_{mft}$, by means of the two STSs $R(\mathcal{S})$ and $D(\mathcal{S})$.

4.1.2 Offline passive tester algorithm

The passive tester algorithm is presented in Algorithm 1. It takes the traces of *Sut* and $R(\mathcal{S})$ and starts by constructing the filtered traces $FTraces(Sut)$ and $D(\mathcal{S})$. Then, it covers every trace t of $FTraces(Sut)$ and checks with the procedure $complies\_with(t,\ R(\mathcal{S}))$(lines 22-32) whether t is a trace of $R(\mathcal{S})$. If the function returns True, it is not necessary to check if the traces t satisfies the second relation $\leq_{mft}$ since $\leq_{ft} \implies \leq_{mft}$ (Proposition 6). On the contrary, the trace t is placed into the set T_1, which gathers the possible failure traces w.r.t. $\leq_{ft}$. The algorithm performs the previous step once more but on the STS $D(\mathcal{S})$. If the function returns False, the trace t is placed into the set T_2. The latter gathers the possible failure traces, w.r.t. the relation $\leq_{mft}$. At the end of the algorithm, if both T_1 and T_2 are empty, the verdicts "Pass$\leq_{ft}$" and "Pass$\leq_{mft}$" are returned, which means that both implementation relations hold. Otherwise, when the first relation does not hold (or the second one), T_1 is provided (or T_2).

When one of the implementation relations does not hold, this algorithm offers the advantage to provide the possible failure traces of $FTraces(Sut)$. Such traces can later be analysed to check if *Sut* has defects. That is helpful for Michelin engineers as it allows them to only focus on what are potentially faulty behaviours, reducing debugging time.

Soundness, termination and complexity of Algorithm 1:

The soundness of Algorithm 1 is captured by the following proposition, whose sketch of proof is given in Annex:

Proposition 9

Algorithm 1: Passive testing algorithm

```
   input  : $R(\mathbb{S}), Traces(Sut)$
   output: Verdicts and possible failure trace sets $T_1, T_2$
1  $T_1 = T_2 = \emptyset$;
2  Build $FTraces(Sut)$;
3  Build $D(S)$;
4  foreach $t \in FTraces(Sut)$ do
5      found=false;
6      foreach $i \in 1, \ldots, n$ do
7          if complies_with(t, $R(\mathbb{S}_i)$) then
8              found=true; break;
9      if $found == false$ then
10         $T_1 = T_1 \cup \{t\}$;
11         foreach $i \in 1, \ldots, n$ do
12             if complies_with(t, $D(\mathbb{S}_i)$) then
13                 found=true; break;
14         if $found == false$ then
15             $T_2 = T_2 \cup \{t\}$
16 if $T_1 == \emptyset$ and $T_2 == \emptyset$ then
17     return "Pass$\leq_{ft}$, Pass$\leq_{mft}$"
18 if $T_1 \neq \emptyset$ and $T_2 == \emptyset$ then
19     return $T_1$, "Pass$\leq_{mft}$"
20 if $T_1 \neq \emptyset$ and $T_2 \neq \emptyset$ then
21     return $T_1$, $T_2$

22 Function complies_with(trace t, STS S ) : bool is
23     if $\exists b = l0_S \xrightarrow{(a_1(p_1),G_1)...(a_k(p_k),G_k)} Pass : trace = a_1()\alpha_1) \ldots a_k(\alpha_k)$ then
24         $M_{[b]} = Mat(b)$ is the Matrix $l \times c$ of $b$;
25         $i = 1$;
26         while $i \leq c$ do
27             $C = M_{[b]}[*, i]$;
28             foreach $j \in 1, \ldots, k$ do
29                 if $\alpha_j \not\models C[j]$ then break ;
30             if $j == k$ then return $True$ ;
31             $i++$;
32     return $False$;
```

1. *$Sut \leq_{ft} R(\mathcal{S}) \implies$ Algorithm 1 returns "$Pass_{\leq_{ft}}$", "$Pass_{\leq_{mft}}$"*
2. *$Sut \leq_{mft} R(\mathcal{S}) \implies$ Algorithm 1 returns "$Pass_{\leq_{mft}}$"*

Algorithm 1 terminates if $FTraces(Sut)$ is finite (which is necessarily the case since the filtered traces are built from a finite number of log files). For every trace in $FTraces(Sut)$, it covers the paths of $R(\mathcal{S})$ and $D(\mathcal{S})$. These STS also have a finite number of paths (at worst equal to the number of traces in $FTraces(Sua)$). The complexity of Algorithm 1 is proportional to $O(mk + mlog(m) + m + T + tmc))$ with t the number of filtered traces of *Sut*, m the number of production events, k the number of inference rules, c the highest number of columns in any matrix $M_{[b]}$ and T the transition number of $R(\mathcal{S})$ (in the worst case).

4.2 Online passive testing

The strong disadvantage of offline passive testing concerns the fault detection delay, which may be long since the traces of *Sut* are collected during a fixed period of time and next analysed. In contrast, in online mode, production events of *Sut* are picked up on the fly and conformance is directly checked. Online passive testing can be assimilated to just-in-time fault detection in such a way that users are notified as soon as possible. Nonetheless, this testing mode brings out several issues, which did not appear previously:

— as production events are received one after one, traces cannot be analysed to remove those including repetitive patterns. Instead of focusing on $FTraces(Sut)$, we shall consider the complete traces of *Sut* ($CTraces(Sut)$). Hence, implementation relations are re-written with $CTraces(Sut)$;

— the traces in $CTraces(Sut)$ are composed of repetitive patterns, but the model generation approach removes repetitive patterns in the STS $R(\mathcal{S})$. Therefore, a comparison between $CTraces(Sut)$ and $Traces_{Pass}(R(\mathcal{S}))$ is no more possible. To define implementation relations, we have to consider an augmented STS of $R(\mathcal{S})$, which also encodes the traces composed of the repetitive patterns of $Pattern(R(\mathcal{S}))$. We denote this STS extension with $\mathcal{S}^R$ (and the extensions of its derived models $R(\mathcal{S})^R$ and $D(\mathcal{S})^R$);

— the passive tester algorithm is now conceived on the "checker state" principle [LCH$^+$06], which means that, for every trace t of *Sut*, it maintains a checker state on the STSs $R(\mathcal{S})^R$ and $D(\mathcal{S})^R$ by constructing their runs that have the same event sequence as t. These runs are updated every time new production events are observed from *Sut*. They allow the detection of possible failure traces.

These modifications also require that a production system under test can be modelled by a LTS *Sut* compatible with a STS $R(\mathcal{S})$ inferred from *Sua* :

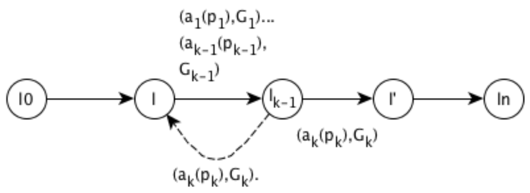


Figure 7 – STS extension $\mathcal{S}^R$

Definition 16 *Let $R(\mathcal{S})$ be a STS obtained from Sua and Sut be a production system. We denote that Sut is compatible with $R(\mathcal{S})$ iff $Pattern(Sut) = Pattern(R(\mathcal{S}))$.*

4.2.1 Implementation relations

We still consider two implementation relations to define conformance between a STS $R(\mathcal{S})$ and a compatible production system *Sut*. Both relations are still founded upon the concept of partial trace inclusion. However, this inclusion is now established between the complete traces of *Sut* ($CTraces(Sut)$) and the traces of the STSs $R(\mathcal{S})^R$ and $D(\mathcal{S})^R$.

Given a STS $R(\mathcal{S})$, obtained from a production system *Sua*, a repetitive pattern $p \in Pattern(R(\mathcal{S}))$ is a sequence of events that appeared several times in a trace of *Sua*. To recover the traces composed of repetitive patterns, we augment $R(\mathcal{S})$ with loops of transitions labelled by p. Figure 7 illustrates this extension. Let $p = a_1(\alpha_1)...a_k(\alpha_k)$ be a repetitive pattern of $Pattern(R(\mathcal{S}))$. In reference to the definition of the filtered traces, there exist transitions from a location l to l' in $R(\mathcal{S})$ labelled with events and guards that accept every valued event of p (solid transitions in Figure 7). The extension of $R(\mathcal{S})$ comes down to adding a loop by doubling the last transition in order to target l again. With this loop, we recover the runs and the traces composed several times of the repetitive pattern p. This STS extension is defined as:

Definition 17 *Let $\mathcal{S}$ be a STS set inferred from a production system Sua, and $Pattern(\mathcal{S})$ be its set of repetitive patterns.*

$\mathcal{S}^R$ is the STS extension $< L_{\mathcal{S}}, l0_{\mathcal{S}}, V_{\mathcal{S}}, V0_{\mathcal{S}}, I_{\mathcal{S}}, \Lambda_{\mathcal{S}}, \rightarrow_{\mathcal{S}^R} >$ such that $\rightarrow_{\mathcal{S}^R}$ is obtained by the following inference rules:

$$\frac{l \xrightarrow{(a(p),G)}_{\mathcal{S}} l'}{l \xrightarrow{(a(p),G)}_{\mathcal{S}^R} l'}$$

$$\frac{l \xrightarrow{(a_1(p_1),G_1)\dots(a_{k-1}(p_{k-1}),G_{k-1})}_{\mathcal{S}} l_{k-1} \xrightarrow{(a_k(p_k),G_k)}_{\mathcal{S}} l', \quad a_1(\alpha_1)\dots a_k(\alpha_k) \in Pattern(\mathcal{S}),\ \alpha_j \models G_j (1 \leq j \leq k)}{l_{k-1} \xrightarrow{(a_k(p_k),G_k)}_{\mathcal{S}^R} l}$$

We also denote $R(\mathcal{S})^R$ and $D(\mathcal{S})^R$ the STS extensions of $R(\mathcal{S})$ and $D(\mathcal{S})$.

Now, we can revisit the two implementation relations used in offline mode. From these, we define two relations denoted $\leq_{ct}$ and $\leq_{mct}$ as they are formalised on the complete traces of *Sut*.

Definition 18 (Implementation relations $\leq_{ct}$, $\leq_{mct}$) *Let $R(\mathcal{S})$ be a STS inferred from Sua and Sut be the system under test, compatible with $R(\mathcal{S})$.*
$Sut \leq_{ct} R(\mathcal{S}) =_{def} CTraces(Sut) \subseteq Traces_{Pass}(R(\mathcal{S})^R)$
$Sut \leq_{mct} R(\mathcal{S}) =_{def} CTraces(Sut) \subseteq Traces_{Pass}(D(\mathcal{S})^R)$

Comparison between the conformance definitions in offline and online modes: although the relations $\leq_{ft}$ and $\leq_{ct}$ (resp. $\leq_{mft}$ and $\leq_{mft}$) sound similar, they exhibit some distinct properties and are actually complementary:

1. $\leq_{ct}$ is written with the complete traces of *Sut*, instead of its filtered traced for $\leq_{ft}$. As a consequence, more traces of *Sut* are considered with $\leq_{ct}$. This implementation relation should reveal more faulty implementations;
2. yet, these complete traces are not filtered on the exit points of *Sut*. As presented early for $CTraces(Sua)$, $CTraces(Sut)$ may hold irrelevant traces, which capture abnormal product life spans within the production system (a product manually removed for instance). As a consequence, the online passive tester should return more possible failure traces. Hence, more traces will have to be manually analysed after the testing step. Naturally, this is time consuming.

4.2.2 Online passive tester Algorithm

The online passive tester takes as input the STS $R(\mathcal{S})$ and receives production events from *Sut* in a synchronous and ordered manner (assumptions given in Section 3). We have chosen to reduce the processing time of the passive tester by designing a parallel algorithm. The received production events are processed among multiple monitor instances, which aim at checking whether both relations $\leq_{ct}$ and $\leq_{mct}$ hold. The receipt of the production events and the management of the monitor instances are done by Algorithm 2, which represents the front-end of the online passive tester.

Algorithm 2 starts by generating $D(\mathcal{S})$ and the STS extensions $R(\mathcal{S})^R$, $D(\mathcal{S})^R$. Then, it continuously receives production events from *Sut*. This flow of incoming events is distributed across different instances of the procedure *Monitor*. A monitor has to build a trace with respect to the trace identification (variable *pid*) and to check whether $\leq_{ct}$ and $\leq_{mct}$ hold for this trace. In line 5 of Algorithm 2, if there already exist two Monitor instances m_1 and m_2 previously launched for the assignment $pid := v$, then the received event is forwarded to them. At this point, there should be two instances of Monitor running the same algorithm per *pid*, i.e., per product being manufactured. On the contrary, if this event is the first event of a trace of *Sut*, Algorithm 2 launches two new Monitor instances, one with the STS $R(\mathcal{S})^R$ and the other one with $D(\mathcal{S})^R$ respectively for $\leq_{ct}$ and $\leq_{mct}$ (lines 8-11). If a Monitor instance has returned a non-empty trace set T (lines 12-17), composed of possible failure traces, then Algorithm 2 produces either the verdict $Fail_{\leq_{ct}}$ or $Fail_{\leq_{mct}}$, which involves that $\leq_{ct}$ or $\leq_{mct}$ does not hold. As for the offline passive tester, it also returns the possible failure trace sets T_1 or T_2, which can later be analysed.

Algorithm 2: The Front-end of the passive tester

```
input  : $R(\mathcal{S})$
output: Trace sets $T_1$, $T_2$ "Fail$\leq_{ct}$","Fail$\leq_{mct}$"
1  Build $D(\mathcal{S})$, $R(\mathcal{S})^R$,$D(\mathcal{S})^R$ ;
2  $Monitors = \varnothing$;
3  while receive $(a(\alpha))$ do
4      $(pid := v) = \alpha(pid)$;
5      if $\exists(m_1 = Monitor((pid := v), R(\mathcal{S})^R) \in Monitors$ and $m_2 = Monitor(pid := v, D(\mathcal{S})^R) \in Monitors)$ then
6          forward $(a(\alpha))$ to $m_1$ and $m_2$;
7      else
8          launch $t_1 = Monitor((pid := v), R(\mathcal{S})^R)$;
9          launch $t_2 = Monitor((pid := v), D(\mathcal{S})^R)$;
10         $Monitors = Monitors \cup \{t_1, t_2\}$;
11         send $(a(\alpha))$ to $t_1$ and $t_2$;
12     if $m_1 = Monitor((pid := v), R(\mathcal{S})^R) \in Monitors$ has returned $T \neq \emptyset$ then
13         $T_1 = T_1 \cup T$; return "Fail$\leq_{ct}$";
14         $Monitors = Monitors \setminus \{m_1\}$;
15     if $m_2 = Monitors((pid := v), D(\mathcal{S})^R) \in Monitors$ has returned $T \neq \emptyset$ then
16         $T_2 = T_2 \cup T$; return "Fail$\leq_{mct}$";
17         $Monitors = Monitors \setminus \{m_2\}$;
```

The procedure Monitor takes as inputs a pid and a STS S (either $R(\mathcal{S})^R$ or $D(\mathcal{S})^R$). It is is based upon a forward checking approach and builds runs of the STS S with respect to the received production events until no more run can be constructed or until a deadlock state of *Sut* is observed. It starts by covering the STS S given as input from its initial state $q_0 = (l0_S, V0_S)$. An index number, initially set to 0 is added to this state. As stated in the definition of the STS reduction, any trace t of $R(\mathcal{S})$ is obtained by following one of its paths b and by assigning values to variables with respect to one column of the matrix $M_{[b]}$. This index, put with a run r, is used to keep track of this column number throughout the building of the run r. Whenever an event $a(\alpha)$ is received (line 4), the procedure Monitor concatenates $a(\alpha)$ to $trace$, which stands for the current trace of Sut. It constructs, from the set Run, the new set of runs $Runs'$ of S that have the same production events as $trace$ by calling the *weak_complies_with* procedure. Two kinds of results may emerge from *weak_complies_with*:

1. if $Runs'$ is empty (lines 9,10), this means that the STS S has no run having the same valued event sequence as $trace$. In this case, Monitor has built a trace of Sut starting from one of its entry points that is not a trace of S. Consequently, there exists a trace of $CTraces(Sut)$, which does not belong to $Traces_{Pass}(S)$, with S either equal to $R(\mathcal{S})^R$ or $D(\mathcal{S})^R$. This trace is thus a possible failure trace either for $\leq_{ct}$ or $\leq_{mct}$. It is returned to the front-end of the passive tester;
2. otherwise (lines 11,12), the algorithm stores the new set of runs into Run and waits for the next event until no more production event is received, i.e., until a deadlock state of Sut is detected (lines 13,14). Deadlock states are detected when no event is received after a given period of time. This delay can be manually set, but it can also be automatically found by analysing the timestamps found in the traces of the system under analysis Sua. If a deadlock state is detected, the procedure Monitor has constructed a trace of Sut, capturing a behaviour that starts from an entry point of Sut and that finishes in one of its deadlock states. Hence, this is a trace of $CTraces(Sut)$. The procedure checks if there exists a run r in $Runs$ such that r ends by a Pass state. If not, this entails that the trace of Sut does not belong to $Traces_{Pass}(S)$, with S either equal to $R(\mathcal{S})^R$ or $D(\mathcal{S})^R$. Once more, this trace is a possible failure trace either for $\leq_{ct}$ or $\leq_{mct}$, and is returned to the front-end of the passive tester.

Given the set $Runs$ of runs of S, the procedure weak_complies_with aims at building the set of runs $Runs'$ of S having the same production event sequence as $trace$. It takes every run (r,c) of Run with r finished by a state (l,v) and l a location of S. It also takes the last received production event $a(\alpha)$. The procedure covers the transitions labelled by $a(p)$ and starting by l to build the runs of $Runs'$. For a transition $l \xrightarrow{a(p),G,A} l_2$, with the guard G referencing a matrix $M_{[b]}$ $Nbl \times Nbc$ (line 19):

1. if $(r,c) = (q_0,0)$, the production event $a(\alpha)$ is the first one received from Sut. For each column cp of the matrix $M_{[b]}$, if $v \cup \alpha$ satisfies the guard

$M_{[b]}[k, cp]$ ($M_{[b]}[k, cp](v \cup \alpha)$ true, line 22), then a new run r_2 is constructed and added to *Runs'*. The state $q_0 = (l0, V0)$ is updated to set the variable $c_{[b]}$ to the value cp. This variable identifies the column number of $M_{[b]}$. $r_2 = q_0 a(\alpha) q_{next}$ is composed of the received production event and of the state q_{next} reached after the firing of the transition $l \xrightarrow{a(p),G,A} l_2$ from the state $q_0 = (l0, V0)$;

2. otherwise (lines 27-31), the matrix column is given by the index c in the couple (r, c). If $v \cup \alpha$ satisfies the guard $M_{[b]}[k, c]$, the run r is completed with $a(\alpha) q_{next}$ with $q_{next} = (l_2, A(v \cup \alpha))$, the state reached after the firing of the transition $l \xrightarrow{a(p),G,A} l_2$ from the state (l, v).

```
1  Procedure $Monitor((pid := v), STS\ S$ is
2      $Runs = \{(q_0, 0) \mid q_0 = (l0_S, V0_S)\}$;
3      $trace = \emptyset$;
4      while $receive\ a(\alpha)$ do
5          $trace = trace.a(\alpha)$;
6          $Runs' = \emptyset$;
7          foreach $(r, c) \in Runs$ do
8              $Runs' = Runs' \cup weak\_complies\_with(a(\alpha), (r, c))$;
9          if $Runs' == \emptyset$ then
10             $T = T \cup \{trace\}$; return $T$;
11         else
12             $Runs = Runs'$;
       // no more receipt -> deadlock detected
13     if $\forall (r, c) \in Runs$, $r$ ends with $q = (l, v)$ and $l \notin Pass$ then
14         return $\{trace\}$;

15 Procedure $weak\_complies\_with(a(\alpha), (r, c), S)$ is
16     $Runs' = \emptyset$;
17     $r = q_0 a_1(\alpha_1) q_1 \dots q$ with $q = (l, v)$ and $l \in L_S$;
18     for $l \xrightarrow{a(p),G,A} l_2 \in \rightarrow_S$ do
19         $G = M_{[b]}[k, c_{[b]}]$, with $M_{[b]}$ the matrix $Nbl \times Nbc$;
20         if $(r, c) == (q_0, 0)$ then
21             foreach $cp$ between 1 to $Nbc$ do
22                 if $M_{[b]}[k, cp](v \cup \alpha)$ true then
23                     $q_0 = (l0, (V0 \wedge (c_{[b]} = cp)))$;
24                     $q_{next} = (l_2, v_2 = A(\alpha \cup v))$;
25                     $r_2 = q_0 a(\alpha) q_{next}$;
26                     $Runs' = Runs' \cup \{(r_2, cp)\}$;
27         else
28             if $G = M_{[b]}[k, c](v \cup \alpha)$ true then
29                 $q_{next} = (l_2, v_2 = A(\alpha \cup v))$;
30                 $r_2 = ra(\alpha) q_{next}$;
31                 $Runs' = Runs' \cup \{(r_2, c)\}$;
32     return $Runs'$;
```

Soundness, termination and complexity of Algorithm 2:
The soundness of Algorithm 2 is captured by the following proposition, whose sketch of proof is given in Annex:

Proposition 10

1. *$Sut \leq_{ct} R(\mathcal{S}) \implies$ Algorithm 2 does not return "Fail$\leq_{ct}$", "Fail$\leq_{mct}$";*
2. *$Sut \leq_{mct} R(\mathcal{S}) \implies$ Algorithm 2 does not returns "Fail$\leq_{mct}$".*

Algorithm 2 terminates upon condition that the number of production events, observed from *Sut*, is finite. Indeed, the procedure Monitor builds runs while receiving events by covering the states of the underlying LTS semantics of the STS given as input (either $R(\mathcal{S})^R$ or $D(\mathcal{S})^R$). If n is the number of locations of the STS, the state number K of the LTS semantics is equal to $K = n \prod_{(v \in I)} card(D_v)$. This state number may be huge, but this corresponds to the worst case. Algorithm 2 builds $D(\mathcal{S})$ with two inference rules. The complexity of this step is proportional to $O(T)$ with T the transition number of $R(\mathcal{S})$. The generation of the STS extensions is $O(T)$ as well. Algorithm 2 launches two Monitor instances per product being constructed within the production system. If p if the number of products, the whole complexity of Algorithm 2 is $O(pK + T)$.

5 Evaluation

We evaluate in this section the benefits brought by Autofunk and its and limitations. We shall express its usability degree with the following criteria:

— *C1 (Accuracy/precision):* are models accurate w.r.t a real production system? Do we always obtain the same results? Does the tool return Pass verdicts when the system under test is identical to the one used to infer models?

— *C2 (Efficiency/Effectiveness):* can a production system be tested in reasonable time? Does Autofunk detect faults? Does it help reduce the testing phase complexity?

— *C3 (Scalability):* can Autofunk take as inputs large trace sets and still build models and test systems in reasonable time?

Initially, while the Autofunk implementation, we executed functional tests in a controlled environment to check how Autofunk builds models and if it can detect faults. In a sense, these functional tests partially evaluate the criteria C1 and C2. *Sua* is materialised by a set of twenty traces constructed manually and composed of ten production events. The functional tests aimed to check that Autofunk always provides the expected STSs. Then, other functional tests were used to check

Exp.	# Days	# Events	$Card(Traces\ (Sua))$	N	M	# FTraces Subsets	# $R(\mathcal{S}_i)$	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	

Table 1 – Results of 6 experiments on model inference

that Autofunk detects faults. To materialise Sut, we kept the same trace set on which we injected faults. We considered the removal/addition of events, of parameters and of repetitive patterns. The functional tests seeded the tester module of Autofunk with this trace set and checked that the faults were detected.

This preliminary phase does not completely answer to the previous questions though. This is why Autofunk was installed inside a real production system (on a Linux machine with 12 Intel(R) Xeon(R) CPU X5660 @ 2.8GHz and 64GB RAM) to assess under what real circumstances these criteria are fulfilled. This system is a workshop (part of a whole factory), in which two main operations are performed: tire assembling (assembling the components onto a tire building drum) and curing (applying pressure to the tire in a mold). It is composed of 3 entry points starting 3 main assembly lines split into a large set of sub-lines to reach devices and operators. Tires can be placed in storage areas in which they may stay several days up to some weeks, or go out of the workshop through 3 exit points. For confidentiality reasons, we cannot provide more details about the system neither a plan of its layout.

5.1 Empirical setup and results

We collected production events from this system by considering several (collection) delays to build models: 1, 8 11, 20 and 23 days.

Table 1 summarises the observed results. 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 number of entry and exit points. The column *Trace Subsets* shows how $FTraces(Sua)$ is segmented per entry point into subsets and the number of traces included in each

Exp.	Sut	*Card* $(Traces(Sua))$	*Card* $(FTraces(Sut))$	Pass$\leq_{ft}$	Pass$\leq_{mft}$	Time (min)
1	$Sut1 = Sua$	61,125	61,125	100%	100%	17
2	$Sut2$ updated from Sua	61,125	25,047	98%	98%	10
3	$Sut3 \neq Sua$	61,125	2,075	3%	30%	4

Table 2 – Results of 3 experiments on passive testing

subset. For instance, in the second experiment, three entry points are detected, hence, $FTraces(Sua)$ is partitioned into three subsets, which give birth to the same number of STSs. The trace numbers, given in this column, also correspond to the numbers of paths generated in the first STSs. The eighth column, # $R(\mathcal{S}_i)$, represents the number of paths found in each reduced STSs $R(\mathcal{S}_1),...,R(\mathcal{S}_n)$, with n the number of entry points. Finally, execution times are rounded and expressed in minutes in the last column.

Based on the recommendations given by the Michelin engineers, we chose to keep as reference model Sua the one obtained in Experiment C. It expresses the system functioning, manufacturing the same product (same tire reference) and it includes the correct number of entry and exit points of the real system. This model was then used to passively test a modified version of the production system in order to detect regressions.

Afterwards, we collected three trace sets from the production system for testing purposes (offline mode) at different periods of time to mostly cover different cases. These experiments are presented in Table 2. The second column shows different kinds of system under test: $Sut1$ that was the same as Sua, $Sut2$ that was slightly updated from Sua with new application versions, and $Sut3$ that was a system much older than Sua and hence very different (older application versions, perhaps other devices, etc.). Column 3 gives the sizes of the trace sets used to infer models, column 4 the sizes of the trace sets collected from the systems under test. The two next columns show the percentage of pass traces w.r.t. the relations $\leq_{ft}$ and $\leq_{mft}$. The last column indicates the execution time for the testing phase.

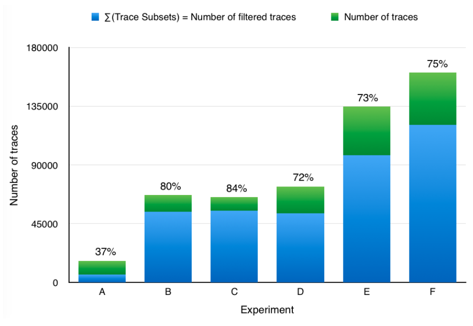
5.2 Evaluation

5.2.1 C1 (Accuracy/precision)

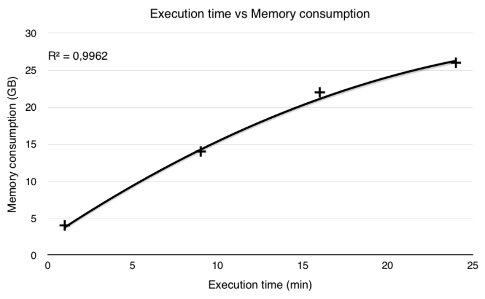
To answer the questions concerning these criteria, we firstly focused on model generation. We extracted the values of columns 4 and 7 in Table 1 to depict the stacked bar chart illustrated in Figure 8(a). This chart shows, for each experiment, the proportion of filtered traces kept to build models, over the initial number of traces in $Traces(Sua)$. The first limitation of Autofunk takes effect in Experiment A. Indeed, only 37% of the initial traces are kept to build models. There are two reasons for this. In one day, too few traces are gathered for using K-means with success. In this trace set, there is not a clear separation between the entry/exit points and the others. Therefore, K-means may return wrong point clusters. In this situation, entry and exit points have to be manually given by an engineer. This is what happened in Experiment A. The second reason concerns the collection delay itself. During one day, most of the recorded traces do not start or end at real entry or exit points of the production system, but rather start or end somewhere within assembly lines. Indeed, the 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, so many incomplete traces are filtered. With more production events, such a phenomenon is limited because these storage delays are absorbed in the period of time considered to collect production events. With the other experiments, the ratios of traces removed from the initial set $Traces(Sua)$ vary between 20 % to 30 %. After some inspections, we observed that, among these traces, around 15 % to 25% appear to be traces composed of repetitive patterns of events. The other traces capture abnormal behaviours of the production system, e.g., unexpected removal of products, device interruptions, etc. and are deleted according to the clusters of entry / exit points given by K-means. We observed that these ratios are acceptable, and few traces expressing normal behaviours of the production system are deleted. But the trace filtering could still be refined with more inference rules or with another cluster analysis technique. For instance, Table 1 revealed strange behaviours not taken into consideration by Autofunk. In experiments B and C, three entry points are detected whereas two are found in the others. Actually, the real production system has three entry points whose two are mainly used. The third one is employed to equilibrate the production load between this system and a second one located close to it in the same factory. Depending on the period, this entry point may be more or less solicited, hence the difference between experiments B, C and experiment D.

The generated models from $FTraces(Sua)$ are accurate in the sense that the normalised traces of the models are equal to the normalised traces of $Traces(Sua)$ (Proposition 9). Regarding the testing phase, we measured accuracy with the first experiment of Table 2. The system under test is here exactly identical to the original production system Sua. This experiment shows that the passive tester yields the expected test verdicts, hence no fault were detected (no regression). Indeed, both relations $\leq_{ft}$ and $\leq_{mft}$ are satisfied. Finally, for a given trace set, if the traces are analysed in a different order, we still obtain the same results, because traces and STS paths are analysed/compared separately by the algorithms of Autofunk. K-means also computes the same clusters from the same trace set.

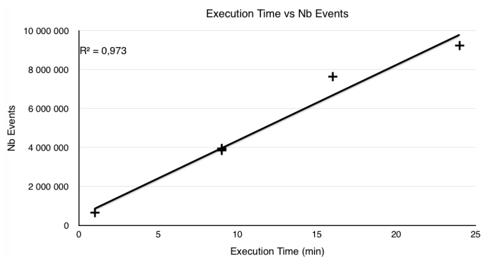
We can conclude that the accuracy and precision of Autofunk depends on the amount of traces collected to build models and on the delay considered to get



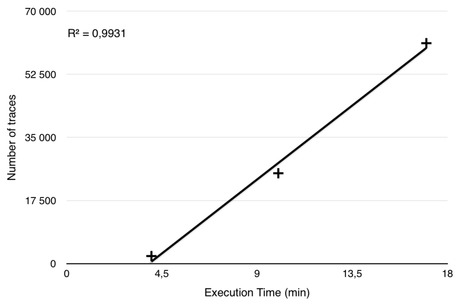
(a) Proportions of filtered traces



(b) Memory consumption vs execution time



(c) Model inference: Execution time vs Nb events



(d) Passive testing: Execution time vs Nb events

Figure 8

these traces. The larger the trace set is and the longer the production events are collected, the more accurate the models are without any loss of precision. Otherwise, the main danger is that K-means might return wrong sets of entry/exit points, which have to be manually given. The best delay for collecting production events depends on the system behaviour. With the production system taken for experimentation, this delay has been set against the storage area functioning. We observed that if events are collected during 7 days or more, we obtain accurate and precise results.

5.2.2 C2 (Efficiency / Effectiveness)

One of the purposes of Autofunk is to automate the testing stage and to quickly return the possible failure traces when non conformance is detected so that they can be later analysed by engineers to diagnose the cause of the non conformance detection (failure, correct behaviour not found in the model, new behaviour obtained after system updates).

Tables 1 and 2 show that Autofunk builds models and gives test verdicts in reasonable time. With production event sets collected during one day up to one week (experiments *A*, *B*, *C*, and *D*), models are inferred in less than 10 minutes. Experiment C in Table 1 corresponds to a typical use of Autofunk for Michelin. Models are generated after 9 minutes from production events collected during 11 days (more than three millions of events).

In Table 1, the difference of STS path numbers between the columns 7 and 8 clearly shows that our STS reduction approach is efficient. For instance, with experiment *C*, we reduce the STSs by 96.7%. In other words, 96% of the original behaviours are packed into matrices. These results mainly stem from our choice to design a context-specific state merging process. It is manifest that these ratios should vary with other kinds of systems. Experiment 2 in Table 2 also corresponds to a typical use of Autofunk for testing. The model were generated from traces collected during 11 days. Traces of the system under test $Sut2$ were collected during 5 days. It took only 10 minutes to check whether Sut is conforming to the STSs inferred from Sua with respect to $\leq_{ft}$ and $\leq_{mft}$.

Experiment 2 in Table 2 shows that Autofunk is effective to detect faults: $Sut2$ was slightly updated from Sua and the tool detected that 98% of the traces are pass traces, the remaining 2% are new behaviours that never occurred before. Here, engineers have to manually analyse the possible failure traces to check if these are the consequence of system failures or of new features of $Sut2$ not present in Sua. 2% means 500 traces, which remains a significant trace amount to inspect. Nonetheless in our manufacturing context, this is still valuable. Before Autofunk, engineers had to manually test the whole system by hand: around several hundreds of scenarios were executed and the resulting traces also had to be manually inspected. Autofunk can test a production system by automatically inspecting thousands of its traces in some minutes. It analyses more the system under test than manual testing, it remains for engineers to manually check a small subset of traces (only those capturing behaviours not found in the model). Such information is essential for Michelin engineers so that they can quickly focus on the potential defects of the system under test. In this example, we observed that almost half of the possible failure traces were caused by unexpected manual interventions done by operators (tires were taken from one position and placed elsewhere). Around a hundred of traces captured new behaviours of $Sut2$ not possible in Sua. None of them revealed real failures in this case.

The third experiment illustrates an abnormal use of our framework. The de-facto usage of our framework is to build models from a production system Sua, which should be older than an new or updated system Sut. Here, the traces of $Sut3$ were collected long before collecting those of Sua used for inferring models. $Sut3$ were indeed a four month older system whose CIM2 applications were different than those of Sua. In this situation, Sua and $Sut3$ can be seen as quite distinctive production systems. Autofunk still detects non-conformance and provides possible failure traces quickly. Unsurprisingly, both implementation relations are unsatisfied. Only 3% of the traces of $Sut3$ are pass traces w.r.t. $\leq_{ft}$. This means that only 60 traces of $Sut3$ exactly match the behaviours captured by the inferred models. With the second implementation relation $\leq_{mft}$, the pass trace ratio is increased to 30%. The second relation shows that roughly a third of the traces of $Sut3$ have the same sequences of events as the traces found in the STSs, but the parameter values (which can be found in other traces) are different. Hence, the second relation shows that 27% of the pass traces appear to be realistic scenarios. This helps focus on the traces having unknown sequences of actions or actions associated with unknown parameters, for which the likelihood of failure detection is the highest. Indeed, Michelin engineers confirmed that these traces often capture system defects.

These experiments revealed that Autofunk is efficient since models and possible failure traces are provided in reasonable time. It can detect non-conformance when the system under test includes behaviours not present in the model and help focus on the unknown behaviours detected from Sut. However, when Sua and Sut have many differences, it detects non-conformance but returns so much possible failure traces that it may become difficult to inspect all of them.

C3 (scalability)

The motivations behind this work and collaboration are to generate models for testing from large sets of production events and to do this as quick as possible so that models may be also used for other purposes, e.g., to diagnose unexpected stops or failures in the production system. The results given in Tables 1 and 2 reveal that our framework can take up to millions of production events and still build models quickly (less than half an hour). Experiment F handled almost 10 millions of events in less than half an hour to build two STSs including around 1,600 paths. As the parsing step is yet not parallelised, it took up to 20 minutes to open and parse around 1,000 files (number of Michelin log files for this experiment). This is a technical issue that needs to be addressed in the future.

The columns 3 and 8 of Table 1 are confronted in the graph of Figure 8(c) to summarise the performances of our framework, and how fast it infers models (experiments B, C and D run in about 9 minutes). Likewise, the columns 4 and 7 of Table 2 give the graph of Figure 8(d). The linear regressions depicted in these figures reveal that the overall framework scales well despite the current production event parsing implementation, by means of the parallelisation of the Autofunk algorithms (STS generation and reduction, trace comparison of the tester).

The memory consumption peak occurs in Autofunk in the beginning of the model inference stage. Every production event is currently loaded in memory and may lead to a memory saturation problem. We compared execution time and memory consumption in Figure 8(b). Memory consumption tends to follow a logarithmic trend line because we partially fixed the memory consumption problem by optimising the object representation, but it is not future-proof. This is an implementation limitation, which needs to be addressed in a next version. At the moment, it has been considered acceptable by Michelin.

6 Conclusion

This paper has proposed Autofunk, a fast and scalable framework combining model generation, expert systems and passive testing to generate formal models and test production systems. Given a large set of production events obtained from a first production system, Autofunk quickly generates STSs, which can be used for passively testing other production systems. Conformance is formally defined with four implementation relations between the system under test and the inferred STSs. The preliminary results obtained by the offline testing method are encouraging, and Michelin engineers see a real potential in this framework.

Nevertheless, many aspects need to be investigated and improved in the future. Firstly, our context-specific STS reduction does not appear to be generalisable while keeping the same performance. Another more general solution, which limits over-approximation, is to guide the model inference with the computation of quality metrics [TNM+13]. At the moment, these approaches are time-consuming though and can only be applied to small systems because the models are incrementally re-generated from scratch to improve the metrics. In Autofunk, another future direction would be to build sub-models of a production system. By now, we consider a whole workshop as a production system to infer models. Focusing on specific locations of a workshop would allow to build smaller models with a generalisable model inference approach. Another future work would be to focus on fault diagnosis. Our preliminary results show that, in normal usage, it still remains a big set of possible failure traces to analyse (we mentioned 2% with our experimentations). Even though the possible failure trace set remains large, Autofunk eases the work of Michelin engineers by highlighting the traces to focus on. We observed that the larger the trace set of *Sua* is, the less under-approximated the model is, and the less possible failure traces we have after testing. Yet, the design of an automatic diagnosis method could be beneficial to reduce the cost of analysing possible failure traces.

Finally, the generated models could also be used for other purposes, such as the generation of documentation, data mining, or predictive maintenance. The main objective of predictive maintenance ([Mob90]) is to decide when to maintain a system according to its state. Maintenance could be here scheduled by mining STSs and particularly the parameter matrices.

A Proof of Proposition 6(sketch)

Proof $Sut \leq_{ft} R(\mathcal{S})$, implies
$\forall t \in FTraces(Sut) : t \in Traces_{Pass}(R(\mathcal{S}))$ (Proposition 13)
Let $t = a_1(\alpha_1)...a_m(\alpha_m) \in FTraces(Sut)$, such that $t \in Traces_{Pass}(R(\mathcal{S}))$
$\exists r = (l0_{R(\mathcal{S})}, V0_{R(\mathcal{S})})a_1(\alpha_1)...a_m(\alpha_m)(Pass, v_m) \in Runs_{Pass}(R(\mathcal{S}))$ (Definition 4), implies
$\exists b = l0_{R(\mathcal{S})} \xrightarrow{(a_1(p_1),G_1,A_1)...(a_m(p_m),G_m,A_m)} Pass \in \rightarrow_{R(\mathcal{S})}$, with $Mat(b) = M_{[b]}$ the matrix $Nbl \times Nbc$ of p, $(1 \leq c \leq Nbc)$ such that $\forall (1 \leq j \leq m) : G_j = M_{[b]}[j,c]$, $(\alpha_j \cup v_{j-1}) \models M_{[b]}[j,c]$ (Definition LTS semantics and 4)
Furthermore, (Definition 15) implies,
$\forall b = l0_{R(\mathcal{S})} \xrightarrow{(a_1(p_1),G_1,A_1)...(a_m(p_m),G_m,A_m)} Pass \in \rightarrow_{R(\mathcal{S})}$, $\exists b' = l0_{D(\mathcal{S})}$
$\xrightarrow{(a_1(p_1),G'_1,A'_1)...(a_m(p_m),G'_m,A'_m)} Pass \in \rightarrow_{D(\mathcal{S})}$ with $G'_j = \bigwedge_{x \in p_j} (Proj_x(M_{[b]}[j,1]) \vee \cdots \vee$
$Proj_x(M_{[b]}[j,Nbc]))$
Consequently, $\forall (1 \leq j \leq m), \exists (1 \leq c \leq Nbc) : (\alpha_j \cup v_{j-1}) \models G_j, (\alpha_j \cup v_{j-1}) \models G'_j$, implies
$t \in Traces_{Pass}(D(\mathcal{S}))$, implies
$Sut \leq_{mft} R(\mathcal{S})$ ■

B Proof of Proposition 9 (sketch)

The proposition can be separated into three points:
1. $Sut \leq_{ft} R(\mathcal{S}) \implies$ Algorithm 1 returns "Pass$\leq_{ft}$".
2. $Sut \leq_{mft} R(\mathcal{S}) \implies$ Algorithm 1 returns "Pass$\leq_{mft}$".
3. $Sut \leq_{ft} R(\mathcal{S}) \implies$ Algorithm 1 returns "Pass$\leq_{ft}$", "Pass$\leq_{mft}$".

For each point, Algorithm 1 relies on the procedure *complies_with*:

Proposition 11 *Let $t \in FTraces(Sut)$ be a trace and $R(\mathcal{S})$ be a STS inferred from Sua.*
$t \in Traces_{Pass}(R(\mathcal{S})))$ $\implies$ the function $complies\_with(t, R(\mathcal{S}))$ returns true.

Proof
$t \in Traces_{Pass}(R(\mathcal{S}))$ implies
$\exists b = l0_{R(\mathcal{S})} \xrightarrow{(a_1(p_1),G_1,A_1)...(a_m(p_m),G_m,A_m)} Pass$ such that $t \in Traces_{Pass}(b)$ (Definitions LTS semantics and 4).
Let $t = a_1(\alpha_1)\dots a_m(\alpha_m)$. The procedure $complies\_with(t, R(\mathcal{S}))$:
— seeks for $b = l0_{R(\mathcal{S})} \xrightarrow{(a_1(p_1),G'_1,A'_1)...(a_m(p_m),G'_m,A'_m)} Pass$, and $M_{[b]}$, the matrix $Nbl \times Nbc$ of b, such that $G'_j = M_{[b]}[j, c_{[b]}]$ with $(1 \leq c_{[b]} \leq Nbc), (1 \leq j \leq m)$ (lines 12-14),
— and checks that $\exists (1 \leq c \leq Nbc), \forall (1 \leq j \leq m), \alpha_j \models M_{[b]}[j, c]$ (lines 15-21).

The procedure *complies_with* seeks for b such that $t \in Traces_{Pass}(b)$ and returns true (line 19) if such b exists.
$\exists b = l0_{R(\mathcal{S})} \xrightarrow{(a_1(p_1),G_1,A_1)...(a_m(p_m),G_m,A_m)} Pass$ such that $t \in Traces_{Pass}(b)$, implies $complies\_with(t, R(\mathcal{S}))$ returns true (line 19).

■

Proof
Proof of 1):
$Sut \leq_{ft} R(\mathcal{S})$ implies
$FTraces(Sut) \subseteq Traces_{Pass}(R(\mathcal{S}))$ (Proposition 13), implies
$\forall t \in FTraces(Sut) : t \in Traces_{Pass}(R(\mathcal{S}))$, implies
$\forall t \in FTraces(Sut)$: the function *complies_with* returns true (Proposition 11), implies
The set $T1$ is empty (Algorithm 1 lines (3,4)), implies
Algorithm 1 returns "Pass$\leq_{ft}$" (line 7).
Proof of 2):
$Sut \leq_{mft} R(\mathcal{S})$ implies
$FTraces(Sut) \subseteq Traces_{Pass}(D(\mathcal{S}))$ (Proposition 8), implies

$\forall t \in FTraces(Sut) : t \in Traces_{Pass}(D(\mathcal{S}))$, implies
$\forall t \in FTraces(Sut)$: the function *complies_with* returns true (Proposition 11), implies
The set $T2$ is empty (Algorithm 1 lines (5,6)), implies
Algorithm 1 returns "Pass$\leq_{mft}$" (line 9).

Proof of 3):
$Sut \leq_{ft} \mathcal{S} \implies Sut \leq_{mft} \mathcal{S}$ (Proposition 6), implies
Algorithm 1 returns "Pass$\leq_{ft}$","Pass$\leq_{mft}$".

■

C Proof of Proposition 10(sketch)

The proposition can be separated into two points:
1. Algorithm 2 returns "Fail$\leq_{ct}$" $\implies \neg(Sut \leq_{ct} R(\mathcal{S})^R)$.
2. Algorithm 2 returns "Fail$\leq_{mct}$" $\implies \neg(Sut \leq_{mct} R(\mathcal{S})^R)$.

For each point, Algorithm 2 relies on the procedures *Monitor* and *weak_complies_with*, which build traces and runs of $R(\mathcal{S})^R$ and $D(\mathcal{S})^R$. To prove 1. and 2., we firstly consider the following proposition on the procedures *Monitor* and *weak_complies_with*:

Proposition 12 *Let $t \in Traces(R(\mathcal{S})^R)$ and $a(\alpha)$ a valued event received by Monitor.*
a) $t.a(\alpha) \in Traces(R(\mathcal{S})^R) \implies \exists (r,c) \in RUNS$: weak_complies_with$(a(\alpha),(r,c))$ returns $RUNS' \neq \emptyset$;
b) $RUNS = \{(r,c) \mid Traces(r) = t$ and $r \in Runs(R(\mathcal{S})^R)\}$ is an invariant.

Proof

1) Let $t = \emptyset$. At the initialisation of *Monitor*, $RUNS = \{(q_0,0) \mid q_0 = (l0_{R(\mathcal{S})}, V0_{R(\mathcal{S})})\}$

a)
$a(\alpha) \in Traces(R(\mathcal{S})^R)$ implies
$\exists b = l0_{R(\mathcal{S})^R} \xrightarrow{a_1(p_1),G,A} l_1$, with $M_{[b]}$ the matrix $Nbl \times Nbc$ and $(1 \leq cp \leq Nbc)$, such that $a = a'$, $G = M_{[b]}[1,cp]$ and $(V0_{R(\mathcal{S})^R} \cup \alpha) \models M_{[b]}[1,c_{[b]}]$
$(V0_{R(\mathcal{S})^R} \cup \alpha) \models M_{[b]}[1,c_{[b]}]$ implies
the procedure *weak_complies_with* (lines 23-27) builds (r_2,cp) such that $r_2 = q_0 a(\alpha)(l_2,$
$A(V0_{R(\mathcal{S})^R} \cup \alpha))$ and $RUNS' \neq \emptyset$.

b)
Both procedures *Monitor* and *weak_complies_with* build all the (r_2,cp) in $RUNS'$ such that $\forall (q_0,0) \in RUNS$ (line 7), $\forall l0_{R(\mathcal{S})} \xrightarrow{a'(p'),G,A} l$ (line 18), $\forall (1 \leq cp \leq Nbc)$ (line 21), $Traces(r_2) = (a,\alpha)$ and $r_2 \in Runs(R(\mathcal{S}))$.
$RUNS$ is replaced by $RUNS'$ (line 13), hence, the invariant holds.

2) Let $t = a_1(\alpha_1)...a_j(\alpha_j)$ and $RUNS = \{(r,c) \mid Traces(r) = t$ and $r \in Runs(R(\mathcal{S})^R)\}$

a)
$t.a(\alpha) \in Traces(R(\mathcal{S})^R)$ implies
$\exists r' = q_0 ... (l,v)a(\alpha)(l_2,v_2) \in Runs(R(\mathcal{S})^R) : Traces(r) = t.a(\alpha)$, implies
$\exists l \xrightarrow{a_1(p_1),G,A} l_2 \in \rightarrow_{R(\mathcal{S})^R}$:, $a_1 = a'$ and $(v \cup \alpha) \models G$.
Furthermore, $\exists (r,c) \in RUNS$ such that $Traces(r) = t$.
$(v \cup \alpha) \models G$ implies
for $l \xrightarrow{a_1(p_1),G,A} l_2$, the procedure *weak_complies_with* builds $r_2 = r(a,\alpha)(l_2,v_2)$ and $RUNS'$ holds (r_2,c) (lines 29-32).
Consequently, $RUNS' \neq \emptyset$

b)
Both procedures *Monitor* and *weak_complies_with* build all the (r_2,c) in $RUNS'$ such that $\forall (r,c) \in RUNS$ with $r = q0...q = (l,v)$ (line 7), $\forall l \xrightarrow{a_1(p_1),G,A} l_2$ (line 18), $Traces(r_2) = t.a(\alpha)$ and $r_2 \in Runs(R(\mathcal{S}))$ (lines 28-31).
$RUNS$ is replaced by $RUNS'$ (line 13), hence, the invariant holds.

By induction on the trace length ($j \geq 0$) and 1), 2), it is then proved that Proposition 12 holds. ■

Now, we can consider Proposition 10 and the two points stated before.

Proof

Proof of 1):

Algorithm 2 returns "Fail$\leq_{ct}$" implies
the procedure *Monitor* has returned $T_1 \neq \emptyset$, implies
the procedure *Monitor* has built a trace $trace \in Traces(Sut)$ and (
— $\forall (r,c) \in RUNS$, r ends with $q = (l,v)$ and $l \notin Pass$ (Procedure Monitor,line 13), or
b) $\forall (r,c) \in RUNS$, $weak\_complies\_with(a(\alpha),(r,c))$ returns $\emptyset$ (Procedure Monitor,lines 9,10).
)

a)
$\forall (r,c) \in RUNS$: r ends with $q = (l,v)$ and $Traces(r) = trace \in Traces(R(\mathcal{S}^R))$ (Proposition 12)
$l \notin Pass$(Procedure Monitor, line 13), implies
$trace \notin Traces_{Pass}(R(\mathcal{S})^R)$.
Furthermore, $trace = a_1(\alpha_1)...a_m(\alpha_m) \in Traces(Sut)$ with: α_1 includes an assignment $(point := val)$ denoting an entry point of *Sut* (Algorithm 2, line 9) and a deadlock state of *Sut* has been detected (Procedure Monitor, line 13), implies
$trace \in CTraces(Sut)$ (Definition 6).
Consequently, $\exists t \in CTraces(Sut)$, $t \notin Traces_{Pass}(R(\mathcal{S})^R)$.

b)
$\forall(r,c) \in RUNS, weak\_complies\_with(a(\alpha),(r,c))$ returns $RUNS' = \emptyset \implies trace \notin Traces(R(\mathcal{S})^R)$ (contrapositive of a) in Proposition 12), implies
$\exists trace \in Traces(Sut) : trace \notin$
$Traces_{Pass}(R(\mathcal{S})^R)$.
Furthermore, $trace = a_1(\alpha_1)...a_m(\alpha_m) \in Traces(Sut)$ and α_1 includes an assignment $(point := val)$ denoting an entry point of Sut (Algorithm 2, line 9).
Let $t' \in (\Sigma_{R(S)})^*$, such that $trace.t' \in CTraces(Sut)$. ($\exists t'$ such that t' leads to a deadlock state of Sut).
$trace \notin Traces_{Pass}(R(\mathcal{S})^R)$, implies
$trace.t' \notin Traces_{Pass}(R(\mathcal{S})^R)$.
Consequently, $\exists t = trace.t' \in CTraces(Sut), t \notin Traces_{Pass}(R(\mathcal{S})^R)$.

With a) or b), Algorithm 2 has built a trace $t \in CTraces(Sut), t \notin Traces_{Pass}(R(\mathcal{S})^R) \implies \neg(Sut \leq_{ct} S)$.
Proof of 2):
Same as 1) except that the set $RUNS$ in the procedure $Monitor$ is initialised with $D(\mathcal{S})^R$. $RUNS = \{(q0,0) \mid q_0 = (l0_{D(\mathcal{S})^R}, V0_{D(\mathcal{S})^R})\}$. ■

References

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, January 2002.

[AMNn12] César Andrès, Mercedes G. Merayo, and Manuel Nuñez. Formal passive testing of timed systems: Theory and tools. *Softw. Test. Verif. Reliab.*, 22(6):365–405, September 2012.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[ANV11] J. Antunes, N. Neves, and P. Verissimo. Reverse engineering of protocols from network traces. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 169–178, Oct 2011.

[BCNZ05] Emmanuel Bayse, Ana Cavalli, Manuel Nunez, and Fatiha Zaidi. A passive testing approach based on invariants: application to the {WAP}. *Computer Networks*, 48(2):247 – 266, 2005.

[BF72] A.W. Biermann and J.A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597, June 1972.

[CMdO09] Ana Cavalli, Stephane Maag, and Edgardo Montes de Oca. A passive conformance testing approach for a manet routing protocol. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 207–211, New York, NY, USA, 2009. ACM.

[DS15] William Durand and Sébastien Salva. Passive testing of production systems based on model inference. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pages 138–147. IEEE, 2015.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[FTW05] L. Frantzen, J. Tretmans, and T.A.C. 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.

[KBP$^+$10] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 179–182, New York, NY, USA, 2010. ACM.

[LCH$^+$06] David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, and Xia Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Trans. Netw.*, 14:424–437, April 2006.

[LLM03] Jérôme Lang, Paolo Liberatore, and Pierre Marquis. Propositional independence - formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 07).

[MMC$^+$10] G. Morales, S. Maag, A. Cavalli, W. Mallouli, E. M. de Oca, and B. Wehbi. Timed extended invariants for the passive testing of web services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 592–599, July 2010.

[MMC13] Pramila Mouttappa, Stephane Maag, and Ana Cavalli. Using passive testing based on symbolic execution and slicing techniques. *Comput. Netw.*, 57(15):2992–3008, October 2013.

[Mob90] R. K. Mobley. *An introduction to predictive maintenance*. Van Nostrand Reinhold's plant engineering series. Van Nostrand Reinhold, 1990.

[MP07] Leonardo Mariani and Mauro Pezze. Dynamic detection of cots component incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[PG09] Michael Pradel and Thomas 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.

[SD15] Sébastien Salva and William Durand. Autofunk, a fast and scalable framework for building formal models from production systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 193–204, 2015.

[TNM$^+$13] Paolo Tonella, Cu Duy Nguyen, Alessandro Marchetto, Kiran Lakhotia, and Mark Harman. Automated generation of state abstraction functions using data invariant inference. In *8th International Workshop on Automation of Software Test, AST 2013, San Francisco, CA, USA, May 18-19, 2013*, pages 75–81, 2013.

[UX07] Hasan Ural and Zhi Xu. An efsm-based passive fault detection approach. In *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems*, TestCom'07/FATES'07, pages 335–350, Berlin, Heidelberg, 2007. Springer-Verlag.

[WH79] M. A. Wong and J. A. Hartigan. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, 28:100–108, 1979.

[WYD07] Cheng Wu, Fan YuShun, and Xiao Deyun. *Computer Integrated Manufacturing*, pages 484–529. John Wiley & Sons, Inc., 2007.

[YEB$^+$06] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.

[ZZXM11] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring specifications for resources from natural language api documentation. *Autom. Softw. Eng.*, 18(3-4):227–261, 2011.