ANOMALY DETECTION IN PRODUCTION PLANTS USING TIMED AUTOMATA
Automated Learning of Models from Observations

Alexander Maier, Oliver Niggemann, Roman Just, Michael Jäger
Institut Industrial IT, OWL Universitiy of Applied Sciences, Lemgo, Germany

Asmir Vodenčarević
Knowledge-Based Systems Research Group, University of Paderborn, Paderborn, Germany

Keywords: Parallelism structure, Behavior model, Timed automata, Anomaly detection, Model-based diagnosis.

Abstract: Model-based approaches are used for testing and diagnosis of automation systems (e.g. (Struss and Ertl, 2009)). Usually the models are created manually by experts. This is a troublesome and protracted procedure. In this paper we present an approach to overcome these problems: Models are not created manually but learned automatically by observing the plant behavior. This approach is divided into two steps: First we learn the topology of automation components, the signals and logical submodules and the knowledge about parallel components. In a second step, a behavior model is learned for each component. Later on, anomalies are detected by comparing the observed system behavior with the behavior predicted by the learned model.

1 INTRODUCTION

Model-based diagnosis uses a model of a production plant to compare the predictions of the model to observations of the running plant. If there arises a difference between the simulation of the model and the running plant, an anomaly is signaled.

The bottleneck in model-based diagnosis is the modeling aspect. Usually, this is done manually by experts who know the plant in detail. After each plant modification, this work has to be repeated.

In this paper, we present a method for the anomaly detection (part of model-based diagnosis) for production plants using probabilistic deterministic timed automata (PDTA) as behavior models. But in contrast to usual approaches, these automata are not created manually but are learned automatically based on observations from the plant. Figure 1 shows our 3-step toolchain for the anomaly detection:

The first step is learning the topology of the automation system: Learning the behavior of parallel, asynchronous components is hard unless this parallelism structure is known beforehand. E.g. learning the timing behavior of 2 asynchronous components, each comprising 1000 states, is hard when the parallelism structure is unknown; obviously up to 1000000 states may be learned.

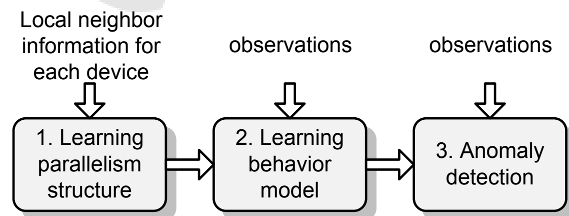

Figure 1: Our toolchain for the anomaly detection.

No data analysis can reveal this parallelism structure. But for the special case of plant signals, the plant structure often mirrors the components' parallelism. And this resembles the topology of the automation system, i.e. the topology of IO devices. So in order to learn the parallelism structure, learning the topology of the automation system is often a good approximation. Further details are given in section 2.1.

In the second step, for each component, a behavior model is learned automatically on basis of recorded plant observations. Section 2.2 gives more details to the model formalism and the learning algorithm.

In the third step anomalies are detected: During runtime of the production plant, we compare the output of the model simulation with observations of the production plant. Typical anomalies in this paper are

Maier A., Niggemann O., Just R., Jäger M. and Vodenčarević A..
ANOMALY DETECTION IN PRODUCTION PLANTS USING TIMED AUTOMATA - Automated Learning of Models from Observations.
DOI: 10.5220/0003538903630369
In *Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics* (ICINCO-2011), pages 363-369
ISBN: 978-989-8425-74-4
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

timing deteriorations or changed signal values. This is discussed in section 3 in detail.

2 LEARNING BEHAVIOR AND PARALLELISM STRUCTURE

In order to learn the overall model, we first have to identify the parallelism structure and finally learn the behavior model for each component individually.

2.1 Learning Parallelism Structure

As mentioned above, the topology of the automation system is used to approximate the parallelism structure. This parallelism structure decomposes the overall model into parallel components—for which sequential behavior models can be learned.

We use AutomationML (AutomationML, 2010) as an exchange format to store the topology of the automation system—and therefore of the parallelism structure. This parallelism structure includes information like the IO devices, Programmable Logical Controllers (PLCs) and communication networks.

For some types of communication networks, the topology of the automation system can be identified automatically. In this paper, the PROFINET standard is used as an example.

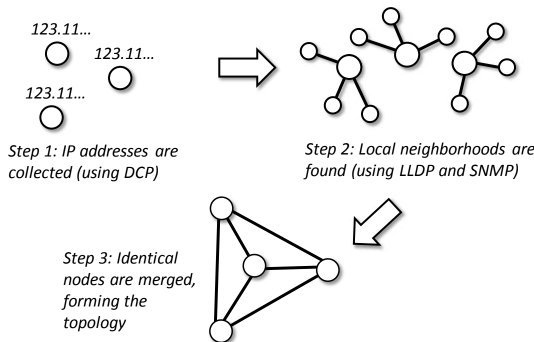

Figure 2: Topology Learning Principle.

The learning procedure is organized in three steps (see figure 2): *(i)* First we collect all IP addresses in the network, using the DCP standard. As result we've got an unsorted collection of all participants in the network. *(ii)* In the next step we look for local neighborhoods for each node. In a PROFINET network each device offers network related data, like local and neighbor information in its own database, called LLDP-MIB. This information is accessible via the SNMP protocol, by addressing each device in the network directly. Collecting local and neighbor information of each network participant leads to raw data, describing single autonomous nodes. *(iii)* Based on these gathered data sets, a topology map can be created, by merging each individual node based on the assignment of neighbor information to local data of other nodes.

In the following, a parallelism structure and its components are defined formally; the definition here is especially tailored for the purpose of model learning.

Definition 1 (Component)**.** *A component C is defined by a behavior function $b_C : \mathbf{R} \times \{0,1\}^m \rightarrow \{0,1\}^n, n,m \in \mathbf{N}$ is a function over m input variables and over time and it returns n output variables.*

The reader may note that we assume a global time base and a deterministic, discrete system; from this it follows that the order of all value changes are predefined, i.e. a component behaves sequentially.

So far, we do not distinguish between components describing plant modules, PLCs, or network devices. While such classifications are necessary from a domain point of view, a general formalism for learning models should abstract from such classifications.

A parallelism structure is now created by connecting several components:

Definition 2 (Parallelism Structure)**.** *A parallelism structure M is defined as a tuple $< \mathcal{C}, z >$ where $\mathcal{C} = \{C_0, \ldots, C_{p-1}\}$ is the set of components and $z : \mathcal{C} \times \mathbf{N} \rightarrow \mathcal{C} \times \mathbf{N}$ maps an output variables of one component onto the input variable of another component.*

I.e. $z(C_i, k) = (C_j, l)$ connects the k's output variable of C_i with the l's input variable of C_j.

Finally we have to learn the behavior model for each component. The following section gives more details about the used formalism and the learning algorithm.

2.2 Learning Behavior Model

In general, model-based diagnosis can use any kind of behavior models. However, the quality of diagnosis depends on the used modeling formalism and the prediction abilities of the models. In the following, we give some requirements to this formalism for the use case of anomaly detection for production plants.

State based Systems. Production plants mainly show a state based behavior, i.e. the system's state is precisely defined by its current and previous discrete IO signals.

Usage of Time. Since actions in automation plants are mostly depending on time, the formalism has to consider it as well.

Probabilistic Information. Here, the behavior models describe the previous, recorded plant behavior. So

unlike in specification models, behavior probabilities must be modeled.

Timed Automata are well suited for the requirements from above. A large number of variants of automata exist. Here we use a timed, probabilistic automaton where the timing is expressed as a relative time span.

In contrast to traditional automata, the formalism here is simplified to ease the learning task: First of all, we allow only for relative timing, e.g. transitions may not refer to a global time base.

Definition 3 (Timed, Probabilistic Automaton)**.** *An automaton is a tuple* $A = (S, S_0, F, \Sigma, T, \delta, Num)$*, where*

- *S is a finite set of states, $S_0 \in S$ is the initial state, and $F \subseteq S$ is a distinguished set of accepted states,*
- *Σ is the alphabet. For a component, Σ equals the set of events.*
- *$T \subseteq S \times \Sigma \times S$ gives the set of transitions. E.g. for a transition $\langle s, a, s' \rangle$, $s, s' \in S$ are the source and destination state and, $a \in \Sigma$ is the trigger event.*
- *A function $Num : T \to \mathbf{N}$ counts the number of observations in which a transition has been used in the system's past. Num can be used to compute a transition probability p for a transition $(v, w) \in T$: $p(v, w) = \frac{Num(v,w)}{\sum_{((v,w') \in T)} Num(v,w')}$.*
- *A transition timing constraint $\delta : T \to I$, where I is a set of intervals. δ always refers to the time spent since the last event occurred. It is expressed as a time range or as a probability density function (PDF), i.e. as probability over time.*

Learning Probabilistic Deterministic Timed Automata

Learning behavior models–i.e. automata–for sequential components follows the methodology from figure 3: First of all, all relevant data is measured from the system. For this, the system is observed during several production cycles. The resulting observation sequences (recorded events and time stamps) are stored in a database.

In a next step, common prefixes of such data sequences are identified. I.e. for the first cycle, the sequence of events is stored in form of an event list. Then for each following cycle, common prefixes with a previous event sequence are identified; if the actual sequence derives at some point, the result is an event tree. The final result is a prefix tree (prefix tree acceptor, PTA) which models all observation sequences in a dense form—dense because common sequences are stored only once.

Now, similar states of the prefix tree are merged. The result is an automaton which models the system.

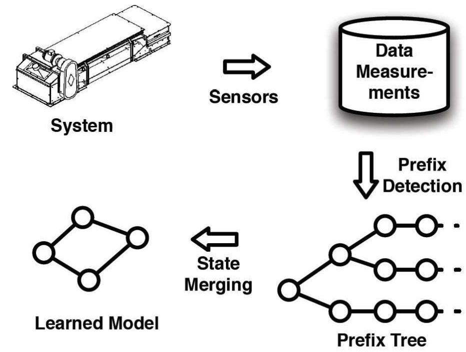

Figure 3: The general learning methodology.

Our algorithm (Bottom Up Timing Learning Algorithm, BUTLA) for learning the behavior models differs to the existing algorithms in two points:
1. Bottom-up merging order: We use a bottom-up merging strategy, i.e. we start with the final (leaf) states and go up to the starting state in a breadth-first-like manner. This eliminates the need for recursive compatibility checks of the sub-trees. This has two advantages: *(i)* the algorithm shows a better runtime behavior and *(ii)* the resulting automaton resembles better the real plant behavior.

This bottom-up strategy works best if all leafs of the prefix trees correspond to final states or to the same states in a cyclic process. Here, the new algorithm applies domain specific knowledge: For measurements of plants, it is normally no problem guaranteeing this constraint.
2. Different time learning operation: Here, we use a different heuristic to learn the correct timing information at the transitions:

(i) First of all, timing is expressed by means of probability density functions instead of time interval; this allows for a much preciser model of the timing.

(ii) At the core of transition timing learning lies one decision: Should a transition with an event e be split into two transitions with different timing information? Unlike other approaches, we base our decision on the timing information itself, not on the sub-tree resemblance; figure 4 shows an example:

s_0 and s_1 are two states in an automaton, the transition timing is a statistic for the transition occurrences in the past and is expressed as a probability density function (shown next to the transition).

Using Verwer's approach, the transition would only be split (new states s_1' and s_1'') if the new resulting sub-trees are different. The motivation is that different states should define different successive behaviors, i.e. sub-trees.

But looking at figure 4, a split could be justified just on the basis of probability density functions: Ob-

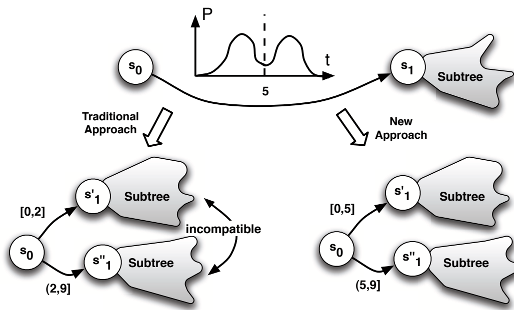

Figure 4: A different timing learning approach.

viously the density function is created by two overlapping Gaussian distribution. So it can be presumed that two different technical processes have created the corresponding event—i.e. here again we apply domain specific knowledge. And different processes must be modeled as different states, because only then can the learning algorithm associate transitions with the correct timing. And only such a precise timing association allows for a correct separation between correct and erroneous behavior (anomaly detection).

Our algorithm *BUTLA* is shown in figure 5 and can be described as follows: First of all, a prefix tree is created (step 1). Then, compatible states are merged in a bottom-up order (steps 2); how state compatibility is defined will be explained later on. If the PDF describing a transition timing is multi-modal (i.e. is the sum of several independent processes, see also figure 4), the transition is split (step 3). Each of the new states created by the split gets a copy of the original sub-automaton (the function *Num* must be recomputed).

BUTLA uses a function *compatible* to check whether two states can be merged (see figure 6): The idea is similar to ALERGIA's approach (see section 4), only that we compare in-going and not out-going transitions and that no recursive sub-tree comparisons are needed.

First of all, several additional variables are needed in figure 6: The number of occurrences of an outgoing transition for a specific state and a specific event ($f(a,v)$, step 1), the number of occurrences of ingoing and out-going transitions for a specific state ($f_{in/out}$, step 2-3) and the number of measurement sequences which end in a specific state (f_{end}, step 4). If the f_{end}s for two states (in relation to f_{in}) are too different (see function *compatible* below), they are not merged (step 6). Similarly, if for any event a the corresponding $f(a,*)$s are too different (in relation to f_{in}), the states are also not merged (step 7).

In step 8, it is checked whether two transitions, that might be merged, have too different timing constraint PDFs. This is done to prevent later unnecessary splits; the function *PDF-different* can be implemented using the well-known R^2 test.

Given:
(1) Discrete component C, its function b_C , its events E
(2) Measurements $\mathcal{S} = \{\mathbf{S}_0, \ldots, \mathbf{S}_{n-1}\}$ where $\mathbf{S}_i = (E \times \mathbf{R})^p, p \in \mathbf{N}$ is one sequence of p events over time (i.e. one measurement or one scenario).
Result: C's function b_C defined by an automaton
(1) Build prefix tree $A = (S, S_0, F, \Sigma, T, \delta, Num)$ based on $\mathcal{S}$. A is a timed, probabilistic automaton according to definition 3
(2) **for all** $v, w \in S$ in a bottom-up order **do**
(2.1) **if** compatible(v, w) **then**
(2.1.2) merge(v, w)
end for
(3) **for all** v in a top-down order **do**
(3.1) **for all** out-going transitions e of v **do**
(3.1.1) **if** transition timing $\delta(e)$ is multi-modal **then**
(3.1.1.1) split(e)
end for
end for

Figure 5: Automata learning algorithm *BUTLA*.

Given: $v, w \in S$
Result: decision yes or no
(1) $f(a,v) := \sum_{e=(*,a,v)\in T} Num(e), v \in S, a \in \Sigma$ where $*$ is an arbitrary element
(2) $f_{in}(w) := \sum_{e=(*,*,w)\in T} Num(e), w \in S$
(3) $f_{out}(v) := \sum_{e=(v,*,*)\in T} Num(e), v \in S$
(4) $f_{end}(v) := f_{in}(v) - f_{out}(v), v \in S$
(5) $d(a,v) := \sum_{e=(*,a,v)\in T} \delta(e)$ where the sum denotes the adding of two PDFs
(6) **if** fractions-different$(f_{in}(v), f_{end}(v), f_{in}(w), f_{end}(w))$
(6.1) **then return false**
(7) **for all** $a \in \Sigma$ **do**
(7.1) **if** fractions-different$(f_{in}(v), f(a,v), f_{in}(w), f(a,w))$
(7.1.1) **then return false**
(8.1) **if** PDFs-different$(d(a,v), d(a,w))$ **then**
(8.1.1) **return false**
end for
(10) **return true**

Figure 6: Comparison algorithm *compatible*.

To compare whether two fractions $\frac{f_0}{n_0}$ and $\frac{f_1}{n_1}$ are significantly different (function *fractions-different*), we use the Hoeffding Bound:

$$\text{different}(n_0, f_0, n_1, f_1) := \left| \frac{f_0}{n_0} - \frac{f_1}{n_1} \right| >$$
$$\sqrt{\frac{1}{2} \log \frac{2}{\alpha}} \left(\frac{1}{\sqrt{n_o}} + \frac{1}{\sqrt{n_1}} \right)$$

where $1 - \alpha, \alpha \in \mathbf{R}$ is the probability of the decision.

SCIENCE AND TECHNOLOGY PUBLICATIONS

3 ANOMALY DETECTION

For diagnosis, we use the discrete probabilistic deterministic timed automaton (PDTA) $A = (S, S_0, F, \Sigma, T, \delta, Num)$ as defined in definition 3. Therefore, the behavior can be defined as a path through the automaton as follows.

Definition 4 (Path through Automaton)**.** *Let $A = (S, S_0, F, \Sigma, T, \delta, Num)$ be an automaton. A path P through the automaton is defined as a list of sequential transitions $P \subseteq T^*$.*

Definition 5 (Observation)**.** *An observation of the plant is defined as $o = (a, t)$, where*

- *$a \in \Sigma$ is the trigger event in the plant and*
- *t is a relative time value (relative to the last signal change).*

The learned automaton is now used to detect an unusual behavior (an anomaly) in an automation plant. During runtime we observe the running automation plant and simulate the identified model in parallel. Then we compare the simulation outputs with the observations from the running system. If there arises any difference, an anomaly (error) has occurred.

Figure 7 shows the algorithm for detecting an anomaly. Following types of anomalies can be detected using this procedure:

Functional Errors. In a current state exits no event (i.e. changing signal value) for a certain signal (sensor/actuator). E.g. while filling a bottle the next event should be "bottle full" but for some reason the filling stops (event: "stop filling").

For every observed event it is checked whetherits symbol corresponds to one of the possible outgoing events in the current state (line 2.2a). An error is found when no transition with the observed event exists, i.e. if the observed path is not equal to one possible simulated path in the automaton.

Timing Errors. A timing error occurs when the signal changes correctly, but the timing range doesn't fit. E.g. if the filling of the bottle should take between four and five seconds, an anomaly would be found, when this takes less than four or more than five seconds. Since we often don't have hard time limits, it's useful to work with distribution functions. In this case we can return the probability of the failure.

For every observed event it is checked whetherthe time of the observed event fits into the time range or doesn't differ more than a predefined deviation from the expected value. This is done in line 2.2b.

Probability Errors. Taking the probabilities into consideration more complex and gradual errors can be detected when the probabilities in the observed system diverge from the probabilities in the model. E.g. while checking the filling of a bottle, 95% of the bottles are filled correct and 5% wrong. Here it would be an anomaly, if the observed probability exceeds this usual value.

In line 2.2.2 it is checked whetherthe probability for taking the chosen transition doesn't vary too much. The occurrences of each event in the real plant are counted and the probabilities are recalculated after each occurrence (line 2.2.1). For the check we need α as additional parameter for the allowed tolerance. If the probability exceeds this tolerance, an error is detected.

Given:
(1) Probabilistic Deterministic Timed Automaton (PDTA) $A = (S, S_0, F, \Sigma, T, \delta, Num)$ (according to definition 3)
(2) $O = (o_1, ..., o_k)$, o_i is an observation according to definition 5
(3) α: a predefined value for the probability deviation
(4) $Num' : T \to \mathbf{N}, \forall e \in T : Num'(e) = 0$
Result: localized anomaly (if there exits one) otherwise 'OK'
Algorithm:
(1) s := S_0 // beginning with initial state
(2) **for** $i := 1$ to k **do** // iterate over all observations
(2.1) $o_i = (a, t)$ // observation with symbol and time
(2.2a) **if** exists $e \in T$ with $e = (s, a, s')$ // check symbol
(2.2b) and $t \in \delta(e)$ **then** // check times
(2.2.1) $Num'(e)$++ // update observed occurrences
(2.2.2) **if** $p_{Num'}(e) - p_{Num}(e) > \alpha$ **then**
(2.2.2.1) **return** anomaly
(2.2.3) $s := s'$ // go to next state
(2.3) **else**
(2.3.1) **return** anomaly
(2.4) **end if**
(3) **end for**
(4) **return** OK

Figure 7: Algorithm for anomaly detection.

4 STATE OF THE ART

4.1 Parallelism Structure

From a model learning perspective, a parallelism structure subdivides the overall system into parallely working components. I.e. a parallelism structure defines a (hierarchical) set of interconnected components where components work in parallel (usually asynchronously) and each individual component shows a sequential behavior only. In plants, such sequential components often correspond to one techni-

cal device such as a robot, a conveyor belt, a reactor, or a PLC (programmable logic controller).

Currently there exists no algorithm to identify a parallelism structure by only using observations in an automation system.

4.2 Behavior Model

The Finite State Machine/ Automaton (FSM) is one of the most established modeling formalism. Based on the initial FSM different types for different cases were developed (e.g. non-deterministic, timed, probabilistic, hybrid). An overview to the main formalisms can be found in (Kumar et al., 2010). Petri nets also allow modeling discrete behavior and are used e.g. by (Cabasino et al., 2007).

There exist already several algorithms for learning an automaton by using observations. In general, there is a distinction between online and offline algorithms. Online algorithms allow to ask for new patterns during runtime while offline algorithms have to deal with a given set of examples. The best known and one of the first online algorithm is Angluin's L* (Angluin, 1987).

Offline algorithms use a prefix tree to collect and combine all recorded observations. MDI (Thollard et al., 2000) and ALERGIA (Carrasco and Oncina, 1999) are two offline algorithms which learn a PDFA. They use only positive examples, i.e. no failure measurements. MDI uses a global criterion to check the compatibility of two states. After each merging step the old automaton and the new one are compared. If the similarity measure exceeds a predefined value, the new automaton is kept, otherwise rejected. ALERGIA uses a local criterion to check the compatibility. Before merging two states, the Hoeffding Bound is used to measure the similarity of these states. If these states are similar enough, they are merged.

Verwer already presented different algorithms for identifying timed automata (Verwer, 2010). Some of them use as well negative as positive examples. To include timing information Verwer introduced a splitting operation which splits a transition if the resulting subtrees are different enough.

4.3 Model-based Diagnosis

Model-based diagnosis using discrete automata was firstly introduced by Sampath et al. (Sampath et al., 1994). They use a discrete deterministic (untimed) automaton. This approach was applied for diagnosis e.g. in (Hashtrudi Zad et al., 2003). In some other contributions this approach is extended to the usage of timed automata (e.g. in (Tripakis, 2002)).

Lunze et al. also work intensively with model-based diagnosis based on timed discrete-event systems (e.g. (Lunze et al., 2001) (Supavatanakul et al., 2006)). The main idea is to create a discrete-event model which corresponds to the discrete-event system and afterwards compare their outputs (see figure 8). If a failure occurs in the system, the diagnostic algorithm detects a difference and suggests the failure which occurred in the system.

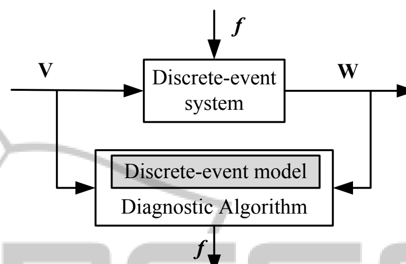

Figure 8: Diagnosis of dynamic systems (Supavatanakul et al., 2006).

SCIENCE AND TECHNOLOGY PUBLICATIONS

5 CASE STUDY

This chapter includes an exemplary use case for the formalisms described below. For this case we use an exemplary plant which is used to transport and produce bulk material e.g. transport corn an produce popcorn. This model factory comprises several modules to store, carry and produce the bulk material. The model factory is controlled by a PLC and the modules are connected using PROFINET.

Using the methodology from chapter 2.1 we discovered the plant topology and detected the two IO-modules. These are used for the parallelism structure. Therefore in the following two automata were learned; one for each module.

A datalogger observes the network traffic on a mirrored port and analyses the profinet frames. For further usage the extracted process data (recorded events and the time stamps) is stored in a database. Using these observations a prefix tree was created for each component. The PTA of the first module contains 26 states, the second one 3611 states.

Then we learned the behavior model as timed automaton (according to definition 3) for each component using the algorithm described in figure 5. The final automaton contains 17 states (8 states in module 1, 9 states in module 2). This corresponds to a compression rate of 99.5%.

Finally, to test the anomaly detection, we caused some failures in the plant. Combining all signal outputs from the components we compared the signal values observed in the running plant with the outputs

of the simulation. Here we used the datalogger in the plant again, but in context of a real-time analysis. The network traffic (profinet frames) are analyzed and after each change of a signal our anomaly detection tool gets a message with the signal, its value and the timestamp.

In some first experiments we inserted 17 different failures. Using the algorithm from figure 7, we were able to detect 88% of the failures correctly. In the remaining 12% we were able to detect the error, but the error cause wasn't identified correctly.

Although we were able to detect most of the errors (at least the failures which were enforced by ourselves), we encountered a problem: Sometimes a correct behavior was recognized as an error. This happens because we are not able to learn the completely correct behavior model. For this we would need an infinite number of recorded test samples. To prevent this, it is possible to enrich the recorded observations e.g. by using a normal distribution and create additional samples. Another possibility is to adapt the model during runtime. For this we would need a supervised learning algorithm which allows the plant operator to add a path to the model. This issue is not yet solved and should be addressed in future work.

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented an efficient method for anomaly detection based on behavior models available as finite state machines/ timed automata. In contrast to usual approaches these models are learned automatically by observing the running plant. We presented an appropriate algorithm for learning this model as timed automaton. Our learning process comprises the learning of the parallelism structure (including the plant topology). Finally we learn the behavior model in the formalism of timed automata for each component.

The overall model is used for anomaly detection. We showed the different types of anomalies which can be detected using this approach and validated the usability of this approach by giving some first experimental results.

During the experiments we encountered the problem, that a model cannot be learned with accuracy of 100%. To reach this, we would need an infinite number of test samples. This means that in practice sometimes a regular behavior is diagnosed as a failure. In future work the learned model should be enriched with empirical data or adapted during runtime.

In further work hybrid automata should be taken into consideration. This will expand the expressiveness and the ability of finding an error reliably. Until now there exists no appropriate learning algorithm for the learning of hybrid automata.

REFERENCES

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Inf. Comp.*, pages 75(2):87–106.

AutomationML (2010). www.automationml.org.

Cabasino, M. P., Giua, A., and Seatzu, C. (2007). Identification of petri nets from knowledge of their language. *Discrete Event Dynamic Systems*, 17:447–474.

Carrasco, R. C. and Oncina, J. (1999). Learning deterministic regular grammars from stochastic samples in polynomial time. In *RAIRO (Theoretical Informatics and Applications)*, page 33(1):120.

Hashtrudi Zad, S., Kwong, R., and Wonham, W. (2003). Fault diagnosis in discrete-event systems: framework and model reduction. *Automatic Control, IEEE Transactions on*, 48(7):1199 – 1212.

Kumar, B., Niggemann, O., and Jasperneite, J. (2010). Statistical models of network traffic. In *International Conference on Computer, Electrical and Systems Science,*. Cape Town, South Africa.

Lunze, J., Schröder, J., and Supavatanakul, P. (2001). Diagnosis of discrete event systems: the method and an example. In *Proceedings of the Workshop on Principles of Diagnosis, DX'01*, pages 111–118, Via Lattea, Italy.

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., and Teneketzis, D. (1994). Diagnosability of discrete event systems. In *11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, volume 199 of *Lecture Notes in Control and Information Sciences*, pages 73–79. Springer Berlin / Heidelberg.

Struss, P. and Ertl, B. (2009). Diagnosis of bottling plants - first success and challenges. In *20th International Workshop on Principles of Diagnosis, Stockholm*, Stockholm, Sweden.

Supavatanakul, P., Lunze, J., Puig, V., and Quevedo, J. (2006). Diagnosis of timed automata: Theory and application to the damadics actuator benchmark problem. *Control Engineering Practice*, 14(6):609–619.

Thollard, F., Dupont, P., and de la Higuera, C. (2000). Probabilistic dfa inference using kullback-leibler divergence and minimality. In *Proc. 17th International Conf. on Machine Learning*, pages 975–982. Morgan Kaufmann.

Tripakis, S. (2002). Fault diagnosis for timed automata. In *FTRTFT*, pages 205–224.

Verwer, S. (2010). *Efficient Identification of Timed Automata: Theory and Practice*. PhD thesis, Delft University of Technology.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS