The concept of residuals for fault localization in discrete event systems

Matthias Roth, Jean-Jacques Lesage, Lothar Litz

The concept of residuals for fault localization in discrete event systems

Matthias Roth [a,b,*], Jean-Jacques Lesage [b], Lothar Litz [a]

[a] Institute of Automatic Control, University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
[b] LURPA - Ecole Normale Supérieur de Cachan, 61, Avenue du Président Wilson, 94235 Cachan Cedex, France

ABSTRACT

In this paper an approach for fault localization in closed-loop Discrete Event Systems is proposed. The presented diagnosis method allows fault localization using a fault-free system model to describe the expected system behavior. Via a systematic comparison of the observed and the expected behavior, a fault can be detected and a set of fault candidates is determined. Inspired by residuals known from diagnosis in continuous systems, different set operations are introduced to generate the fault candidate set. After fault detection and a first fault localization, a procedure is given to render the fault localization more precisely by an analysis of the further observed system behavior. Special emphasis is given to the use of identified models for the fault-free system behavior. The approach is explained using a laboratory manufacturing facility.

Keywords:
Discrete event systems
Fault detection
Fault diagnosis

1. Introduction

Diagnosis in Discrete Event Systems (DES) has been a vital research area in the last 15 years. The class of technical systems that can be modeled as DES includes production, transportation and communication systems (Cassandras & Lafortune, 2006). One of the main purposes of diagnosis methods for technical systems is to increase their availability by helping the system operator to find fault sources as quickly as possible. An increased availability leads to an improved system dependability and has positive effects on economic key issues such as productivity.

An important class of DES diagnosis approaches is model based. The idea behind is to compare the modeled and the observed system behavior in order to detect and to localize faults. Model-based approaches can be divided in two groups. The first group considers models that contain fault-free behavior as well as system behavior for given faults. The second possibility uses models of fault-free system behavior only.

A prominent example for DES-approaches with automaton models including the faulty system behavior is given in Sampath, Sengupta, Lafortune, Sinnamohideen, and Teneketzis (1996). One of the main features of this "diagnoser" approach is the possibility to give guarantees concerning the diagnosability of faults that are considered in the underlying model. If certain conditions hold, faults considered in the model can be precisely localized. Various improvements of this approach have been proposed in the last years mainly dealing with adapting the method for distributed diagnosis e.g. Sayed-Mouchaweh, Philippot, and Carre-Menetrier (2008) or incorporating timing information e.g. Hashtrudi Zad, Kwong, and Wonham (2005).

Apart from using automata as system models for DES, Petri nets are a widely spread modeling formalism. In Fanti and Seatzu (2008) an overview of existing diagnosis approaches using Petri nets is given. In the most recent works it is proposed to perform diagnosis based on integer linear programming, see e.g. Dotoli, Fanti, Mangini, and Ukovich (2009). The idea is to check if an observed behavior is consistent with some modeled faulty behavior. An inherent disadvantage of approaches relying on fault models is that only faults explicitly considered in the system model can be detected and localized.

Diagnosis methods only working with a model of the nominal fault-free system behavior avoid this disadvantage. Faults that lead to an inconsistency between the nominal and the observed behavior can be detected when comparing modeled behavior and measurements without having to be considered explicitly (Cordier et al., 2004; Reiter, 1987). Another advantage of this class of methods is that model building is straightforward since no special knowledge of faulty system behavior is necessary. A drawback of this second class of approaches is that fault localization is more ambitious than in the case of methods relying on fault models, since the models have been built using less knowledge. A second drawback is that diagnosability of given faults can usually not be guaranteed.

In this paper, fault localization is studied based on a fault-free system model. The approach has been developed for a widely used class of closed-loop DES consisting of a plant and a controller. Although the proposed method is not restricted to identified models, special emphasis will be given to the implications of models obtained

E-mail addresses: mroth@eit.uni-kl.de (M. Roth), lesage@lurpa.ens-cachan.fr (J.-J. Lesage), litz@eit.uni-kl.de (L. Litz).

by identification. The paper is structured as follows: In Section 2 an overview of the proposed diagnosis approach is given. Section 3 sketches the identification method that has been used to obtain the fault-free system model. It also defines the semantics of the used model class. Section 4 represents the kernel of the paper: It is explained how to detect faults and to define appropriate residuals for fault localization using the model defined in Section 3. In order to show the relevance of the proposed method for real systems, a case study is treated in Section 5.

2. Overview of the proposed diagnosis approach

According to Isermann and Balle (1997) the term fault diagnosis is defined as follows.

> Diagnosis: Determination of the time of detection, kind, size and location of a fault. Diagnosis follows detection, includes fault isolation and identification.

The definition of Isermann and Balle (1997) is proposed with the background of time continuous systems. Since this paper deals with a different class of systems and different models, not each aspect of this definition applies to the presented approach. Systems that are considered in this paper are DES consisting of a closed-loop of plant and controller (Fig. 1). The controller has a certain number of binary outputs connected with actuators in the plant. Sensors of the plant are connected to binary controller inputs. Possible faults in this scenario are broken actuators and sensors as well as damaged plant hardware. In the presented approach the only information used to perform the diagnosis task are the signals exchanged between controller and plant. The aim is to detect if a sensor or an actuator fails or if some plant hardware gets damaged which leads to an abnormal behavior. Recognizing such an abnormal behavior will be referred to as fault detection. In order to allow systematic repair actions, it is also helpful to give information of where a detected fault is located. This process will be referred to as fault localization.

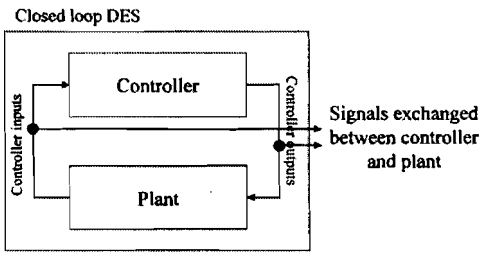

Fig. 1. Closed-loop DES consisting of plant and controller.

The aim of fault localization in the context of this paper is to determine controller I/Os (inputs and outputs) which are related to a faulty component. Based on the *binary* signals exchanged between controller and plant it is not possible to gather any information concerning the "size" of a fault. Hence, in this paper the diagnosis task is considered to be completed after fault localization.

A model-based diagnosis method that is well established for continuous systems is to use residuals. Isermann and Balle (1997) define residuals as follows:

> A residual is a fault indicator, based on a deviation between measurements and model-based computation.

In the upper part of Fig. 2 this principle is depicted for continuous systems. The deviation between a measured signal evolution (dashed line) and the signal evolution computed using a model (continuous line) is quantified. Analyzing this quantification it is possible to detect and localize faults in the observed system. Fig. 2 also shows measurements of a DES and an automaton that models the considered system. Typically, observations of a DES are event sequences as shown in the lower part of the figure. The expected event trace can be derived from the automaton modeling the system. The observed sequence *a*, *b* can be reproduced by the state trajectory 0, 4, 5. When event *c* is observed, the sequence is no longer reproducible with the given automaton and a fault can be detected. The approach presented in this paper focuses on the analysis of the difference between measured and expected DES behavior. Two generic fault symptoms will be considered in Section 4: Faults leading to observed but unexpected events and faults that lead to missed events.

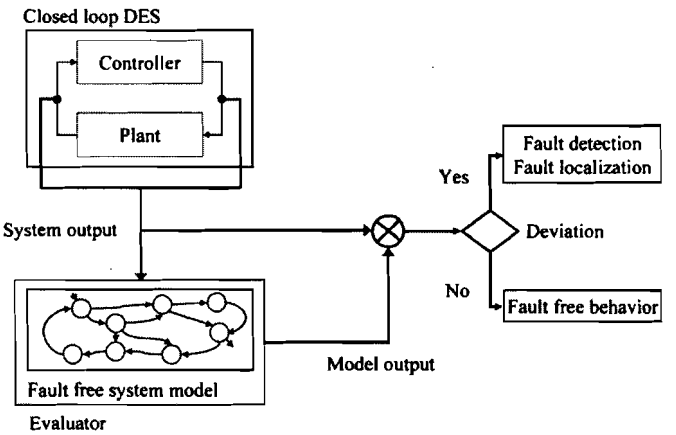

Fig. 3. FDI principle for closed loop DES.

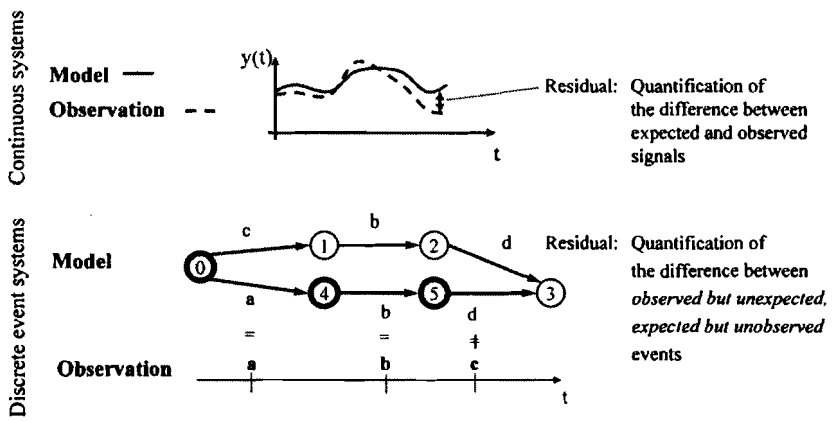

Fig. 2. Residuals in time-continuous and in discrete event systems.

The proposed fault detection and localization principle for closed loop DES is depicted in Fig. 3. The closed-loop DES is observed by collecting the signals exchanged between controller and plant. This measured system output is given to an evaluator which contains an automaton modeling the fault free system behavior. The evaluator tries to reproduce the observed system output using the embedded model. This evaluation creates the model output that is compared to the measured output. If measured and modeled output differ, a fault is detected and fault localization starts in order to determine where the fault is located.

3. Model identification

3.1. Data collection and arrangement

As explained in the former section, model-based diagnosis is driven by the comparison of modeled and observed behavior. The closer the model to the real behavior is, the more accurate results model-based diagnosis can yield. The usual way for manual building of systems consisting of controller and plant (see Fig. 1) is to build models for the plant G_{Plant} and for the controller algorithm $G_{Controller}$ (Sampath et al., 1996). The system model G is obtained by parallel composition of the plant and controller models: $G = G_{Plant} \| G_{Controller}$. Usually, $G_{Controller}$ only models the idealized behavior of the controller algorithm. Hardware specific aspects of the controller are usually not considered. An example for such an aspect are industrial controllers where several controller outputs can change their value within one controller cycle. An example for effects usually not considered during manual model building of the plant model G_{Plant} are chattering sensors. As a consequence, manually built models often represent the specified system behavior rather than the real one. If the real behavior is compared with an idealized model, the results can either be false alerts or non-detectable faults.

The main advantage of working with an identified model is that it represents the real behavior of the closed loop system with respect to the identification data base. As identification data base, it is possible to use behavior observed during normal system operation. This behavior includes consequences implied by the specific controller hardware as well as non-ideal (but not faulty) plant behaviors like chattering sensors.

For model identification and for diagnosis itself it is necessary to collect the signals exchanged between controller and plant when the system is in operation. These signals are available by considering the inputs and outputs of the controller. The controller I/Os are arranged in I/O vectors with several properties summarized in Definition 1.

Definition 1 (*Controller I/O vector*). Given r different controller inputs $I_1, \ldots, I_r$ and s different controller outputs $O_1, \ldots, O_s$, the controller I/O vector $u=(IO_1,\ldots,IO_m)$ with $m=r+s$ is given by $IO_i = I_i \forall i = 1,..,r$ and $IO_{r+i} = O_i \forall i = 1,..,s$. m denotes the dimension of the vector (number of controller I/Os).

The identification of the model is based on sequences of observed I/O vectors. For the identification it is necessary to observe a certain number of system evolutions and to collect the according I/O vectors. The sequence of I/O vectors that is exhibited during a given system evolution is built by I/O vectors in the order of their appearance:

Definition 2 (*I/O vector sequence*). If during the h-th system evolution l_h I/O vectors u_h have been observed, the sequence is denoted as $\sigma(h) = (u_h(1),u_h(2),\ldots,u_h(l_h))$.

Hence l_h denotes the length of the h-th I/O vector sequence. We assume that for two successive I/O vectors $u(t) \neq u(t-1)$ holds. In order to translate the observed I/O vector sequences into a model, a formal definition of the observed system behavior is necessary. We define the set of observed words with length q observed during p different system evolutions:

Definition 3 (*Observed word set and language*). The observed words of length q are denoted as

$$W^q_{Obs} = \bigcup_{i=1}^{p} \left(\bigcup_{j=1}^{l_i-q+1} (u_i(j),u_i(j+1),\ldots,u_i(j+q-1)) \right).$$

With the observed word set we can define the observed language of length n of the system as

$$L^n_{Obs} = \bigcup_{i=1}^{n} W^i_{Obs}.$$

The observed language of length n consists of the I/O vector sequences up to length n that have been observed to get the necessary data base for identification.

3.2. Model definition

Since the observed system language has to be translated into an appropriate DES model, we have to define an automaton that reflects the characteristics of the closed-loop DES. The closed-loop DES from Fig. 1 is an autonomous event generator that exhibits a system output. A well programmed controller can be considered as deterministic, whereas the physical plant must generally be considered as non-deterministic. Hence the coupled system of plant and controller must be considered as non-deterministic. An automaton that reflects these properties is the *Non-Deterministic Autonomous Automaton* with *Output* that is defined by

Definition 4 (*NDAAO*). $NDAAO = (X,\Omega,f,\lambda,x_0)$ with X finite set of states, Ω output alphabet, $f : X \to 2^X$ non-deterministic transition function, $\lambda : X \to \Omega$ output function and x_0 the initial state.

This automaton will be presented as a digraph in the usual manner (Cassandras & Lafortune, 2006). The words and language generated by this automaton are given by

Definition 5 (*Words and Language of the NDAAO*). The set of words of length n generated from a state $x(i)$ is defined as

$$W^{n=1}_{x(i)} = \{w \in \Omega^1 : w = \lambda(x(i))\}$$

and

$$W^{n>1}_{x(i)} = \{w \in \Omega^n : w = (\lambda(x(i)),\lambda(x(i+1)), \ldots,\lambda(x(i+n-1))) : x(j+1) \in f(x(j)) \forall i \leq j \leq i+n-2\}.$$

The language generated by the NDAAO is given by

$$L^n_{Ident} = \bigcup_{i=1}^{n} \bigcup_{x \in X} W^i_x.$$

If the output alphabet Ω consists of I/O vectors, it is possible to reproduce the observed language by performing state trajectories in the automaton.

3.3. Identification and model properties

For identification of DES various methods exist (see e.g. Fanti & Seatzu, 2008 for an overview of identification of Petri nets). An identification algorithm that has especially been designed to deliver appropriate models for fault detection purposes is given in Klein (2005). It works on the basis of words of the parametric length k and yields a non-deterministic autonomous automaton with output (NDAAO) as introduced in Definition 4. Basically, states representing observed words of length k are connected in

the order the words have been observed. The algorithm delivers an automaton that is $k+1$-complete (Moor, Raisch, & Young, 1998). This means that the automaton generates exactly the set of observed words of length $k+1$: $L_{Obs}^{k+1} = L_{Ident}^{k+1}$. This characteristic is advantageous for fault detection: If the observed language has been collected during fault-free system behavior only, it can be made sure that each fault leading to an abnormal word of length $k+1$ can be detected since the model (that is intended to be fault-free) cannot erroneously reproduce it. Faulty I/O vector sequences (words) that are reproduced by the model represent undetectable faults.

A second important model property is shown in Roth, Jean-Jacques, and Litz (2010a): If it is possible to state $L_{Obs}^{k+1} = L_{Orig}^{k+1}$ (with L_{Orig}^{k+1} denoting the fault-free language of the considered system) for an arbitrary k, $L_{Ident}^{n>k} \supseteq L_{Orig}^{n>k}$ holds: If the system language of length $k+1$ (L_{Orig}^{k+1}) has been completely observed, the identified model simulates the complete original system language of length larger than k ($L_{Orig}^{n>k}$). This reduces the number of false alerts when using the identified model for diagnosis purposes: If $L_{Obs}^{k+1} = L_{Orig}^{k+1}$ holds for a given k, the model is able to reproduce *any* fault-free word which can be exhibited by the system. Hence, none of the fault-free words will lead to a false alert. Since both model properties relate the identified and the observed system language, they help validating the model against the observed behavior.

Since for the second property it is necessary to state $L_{Obs}^{k+1} = L_{Orig}^{k+1}$, the most important question for the identification approach is to determine a k for which this assumption holds. Fig. 4 gives a typical evolution of the observed word set for a closed loop system over time. The data has been captured for the case study system that will be described in detail in Section 5. The word sets have been collected during fault-free system evolutions. It can be seen that $|L_{Obs}^1|$ converges to a stable level after about 8 system evolutions. This allows concluding that probably each word of length one (single I/O vectors) that is exhibited during normal system behavior has been observed. It is reasonable to state that this also holds $|L_{Obs}^2|$ and $|L_{Obs}^3|$ (I/O vector sequences of length two and three). If word sets containing longer sequences are considered, it can be seen that the according curves do not perfectly converge. For the data depicted in Fig. 4 it can be concluded that $L_{Obs}^{k+1} = L_{Orig}^{k+1}$ holds for $k=2$ due to the convergence of the according curves. A model identified with $k=2$ will thus not lead to false alerts during online diagnosis. If k is chosen to a larger value, it is likely that the system exhibits new fault-free words of length $k+1$ after the identification data has been collected since the according languages do not converge to a stable level. Since these new words cannot be reproduced by the model, false alerts would be the consequence.

Note that problems similar to the choice of k must be solved if the model is obtained from manual model building. The engineer creating the model has always to decide upon the level of abstraction and necessary model accuracy. It is up to his modeling experience to ensure that the model that is supposed to represent the fault-free system behavior contains any possible fault-free behavior and does not erroneously contain faulty sequences. In general, neither working with identified nor working with manually created models guarantees having a perfect system representation.

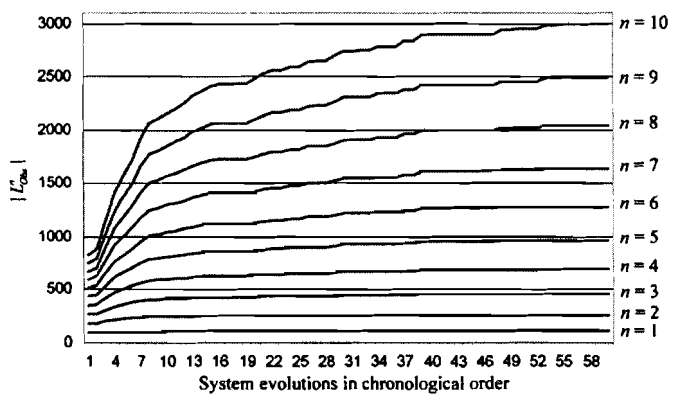

Fig. 4. Evolution of the observed word set for the case study.

4. Fault detection and localization

4.1. Definition of I/O behavior

In Section 3.2 it has been explained that the NDAAO can be used to reproduce I/O vector sequences of the considered system if its output alphabet consists of I/O vectors. In case of a fault it is not sufficient to know that an observed I/O vector sequence cannot be represented by the model (fault detection). It is necessary to determine I/Os which are possibly responsible of the malfunctioning (fault localization). Hence, it should be possible to determine the behavior of single I/Os based on the observed and modeled I/O vectors. If the system produces a new I/O vector, it is the result of at least one I/O changing its value. If a binary I/O changes its value a rising or falling edge can be observed (see Fig. 5):

Definition 6 (*Edges*). For each controller I/O IO_i there exist three edges: IO_i_0 to indicate a change from 1 to 0 (falling edge), IO_i_1 to indicate a change from 0 to 1 (rising edge) and IO_i_ε to indicate no change in value:

$$E = \{IO_i\_0 \cup IO_i\_1 \cup IO_i\_\varepsilon \forall 1 \le i \le m\},$$

with m denoting the number of controller I/Os in the system.

In order to determine edges appearing if two arbitrary I/O vectors are considered, an edge function is defined:

Definition 7 (*Edge function*). Let $IO_i(j)$, $IO_i(k)$ be the i-th controller I/O in the j- and k-th I/O vector.

$$Edge(IO_i(j), IO_i(k)) = \begin{cases} IO_i\_1 & \text{if } IO_i(j) = 0 \text{ and } IO_i(k) = 1, \\ IO_i\_0 & \text{if } IO_i(j) = 1 \text{ and } IO_i(k) = 0, \\ IO_i\_\varepsilon & \text{if } IO_i(j) = IO_i(k) \end{cases}$$

delivers the resulting edge of the i-th I/O from the comparison of its values from the I/O vectors $u(j)$ and $u(k)$.

Instead of $Edge(IO_i(j), IO_i(k))$ it is also written $Edge(u(j)[i], u(k)[i])$. Since more than one I/O can change their value when new I/O vectors are produced, the evolution set that summarizes the edges 'between' two I/O vectors is defined:

Definition 8 (*Evolution set*).

$$ES(u(j), u(k)) = \{Edge(u(j)[i], u(k)[i]) \in E \mid Edge(u(j)[i], u(k)[i]) \neq IO_i\_\varepsilon \forall 1 \le i \le m\}$$

determines the set of rising and falling edges between two I/O vectors $u(j)$ and $u(k)$.

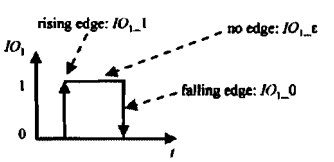

Fig. 5. Example for rising or falling edges of an I/O.

Fig. 6 shows an example for the evolution set resulting from the comparison of two I/O vectors.

4.2. The NDAAO as part of the evaluator

As explained in Section 2, the first step in diagnosis is fault detection. In the presented approach fault detection is performed using the NDAAO as fault-free system model. Fig. 3 shows that the model is used in the evaluator to reproduce the observed system output.

A more detailed view of this procedure is given in Fig. 7. The part with dashed lines will be introduced in the next section. The current I/O vector $u(t)$ that is exhibited by the closed-loop DES is given to the evaluator. Following Algorithm 1, the evaluator tries to determine the current state of the NDAAO. The main idea of this algorithm is to determine a current state estimation following the observed evolution set. $\lambda(x)$ determines the output of an NDAAO state according to Definition 4 and ES is the evolution set according to Definition 8. With $\tilde{X}_{t-1}$ and $\tilde{X}_t$ the state estimations after the occurrence of the t-th and $t-1$-th I/O vector are denoted. Before the algorithm is started, both sets are empty.

Algorithm 1. Evaluator algorithm.

Require New observed I/O vector $u(t)$ and former state estimation $\tilde{X}_{t-1}$

1: if $|\tilde{X}_{t-1}| > 0$: **then**
2: $\tilde{X}_t \leftarrow \{x \in X | \exists x_{pre} \in \tilde{X}_{t-1} : x \in f(x_{pre}) \wedge ES(\lambda(x_{pre}),\lambda(x)) = ES(\lambda(x_{pre}),u(t))\}$
3: **else**
4: $\tilde{X}_t \leftarrow \{x \in X | \lambda(x) = u(t)\}$
5: **end if**
6: $\tilde{X}_{t-1} \leftarrow \tilde{X}_t$
7: **return** $\tilde{X}_t$

The algorithm checks if the former state estimation $\tilde{X}_{t-1}$ contains at least one state. If the observation is being initialized or after a fault has been detected, this set is empty. In case of an empty estimation $|\tilde{X}_{t-1}| = 0$, the evaluator determines each NDAAO state with the observed I/O vector as output as possible current state (see line 4 of Algorithm 1) and adds it to the current state estimation $\tilde{X}_t$. If the former state estimation $\tilde{X}_{t-1}$ was not empty (line 2), the algorithm checks the observed evolution set of I/O edges in order to determine the current state estimation. Each NDAAO state that can be reached by reproducing the observed I/O edges starting in one of the states from the former state estimate $\tilde{X}_{t-1}$ is added to the current state estimation $\tilde{X}_t$. At the end of the algorithm, the former state estimation $\tilde{X}_{t-1}$ for the next run of the algorithm is prepared by copying $\tilde{X}_t$ to it.

The complexity of the evaluator algorithm with respect to the number of I/Os in the system is linear since the number of I/Os defines the number of comparisons carried out in line 2 or line 4. Practical experience with the algorithm showed that it is able to process a newly observed I/O vector even with a large model within a few milliseconds which is sufficient for most industrial applications.

The resulting state estimation $\tilde{X}_t$ of the evaluator as well as the observed I/O vector $u(t)$ are given to the analyzer where the fault detection policy as well as the fault localization operations are implemented. The fault detection policy is given by

$$FD(\tilde{X}_t) = \begin{cases} fault & \text{if } |\tilde{X}_t| \neq 1, \\ OK & \text{if } |\tilde{X}_t| = 1. \end{cases} \quad (1)$$

A fault is detected if the evaluator cannot determine an unambiguous state estimation. Fig. 8 shows an example of the state estimation and fault detection process. In the lower part of the figure, the NDAAO is depicted. In the part "observation" an observed I/O vector sequence is shown. Between NDAAO states and between I/O vectors, the resulting edges are given. It is supposed that the algorithm starts without a defined initial NDAAO state x_0 like it is the case if the start of the diagnosis procedure is not synchronized with the start of the closed-loop DES. After the observation of the first I/O vector, the evaluator algorithm goes to line 4 since $|\tilde{X}_{t-1}| = 0$. The evaluator determines the NDAAO states with $\lambda(x) = u(t)$ and adds them to $\tilde{X}_t$. State x_1 and state x_3 have the first observed I/O vector as output. The fault detection policy returns "fault" since the state estimation is not unambiguous. This changes as soon as the next I/O vector is observed which leads to IO_1_0. Starting from x_1 or x_3 only x_2 can be reached by producing the same evolution set. Hence the state estimation only contains one state (line 2 of Algorithm 1) which makes the fault detection policy declare "OK". The state estimation proceeds with the next observed

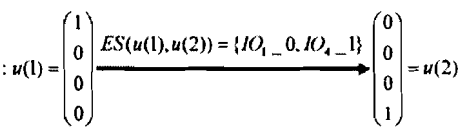

Fig. 6. Example for the evolution set resulting from the two I/O vectors $u(1)$ and $u(2)$.

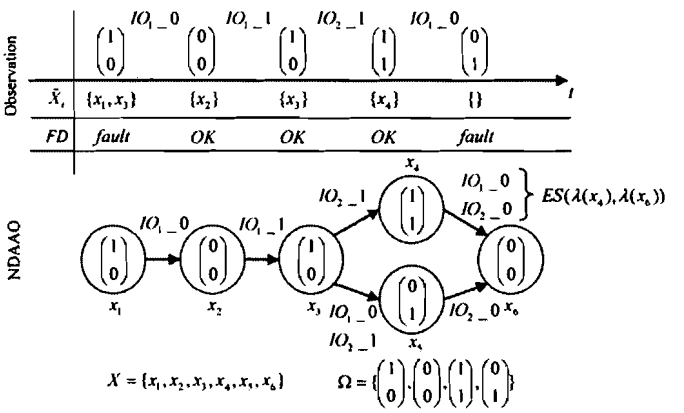

Fig. 8. Example for state estimation and fault detection.

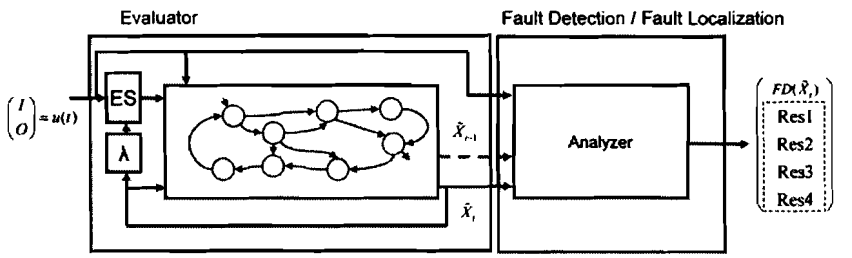

Fig. 7. FDI scheme with the NDAAO.

I/O vectors by determining the state trajectory x_3, x_4. When the last I/O vector is observed, the resulting evolution set $\{IO_1\_0\}$ cannot be produced by leaving state x_4. The only evolution set that can be generated by leaving x_4 is $\{IO_1\_0, IO_2\_0\}$ which differs from the observed one. Hence the state estimation from line 2 in Algorithm 1 results in an empty set which leads to fault detection. During the initial phase of the observation a fault is detected until the state estimation becomes unambiguous. This can be avoided if an initial NDAAO state x_0 is given and the start of the diagnosis process and the closed-loop DES are synchronized.

4.3. Definition of the residuals

After a fault has been detected, the next step is to determine which sensor, actuator or hardware part of the plant is possibly affected. Since sensors and actuators are directly connected to controller I/Os, it is helpful to give a certain number of I/Os that could be related to the fault. In the following two subsections four residuals will be introduced that formalize generic fault symptoms. The idea of the residuals follows the definition of Isermann and Balle (1997) where they are defined as fault indicators based on a deviation between *measurements* and *model-based computation* (see Section 2). In order to calculate the residuals the state estimation before a fault was detected is supposed to be unambiguous $|\tilde{X}_{t-1}| = 1$. Hence, only one NDAAO state was formerly considered as possible current state. If this condition does not hold, the residuals are not calculated.

4.3.1. Unexpected behavior

The first class of residuals has the aim to localize faults that led to an observed behavior that was unexpected in the given context. The current context is defined by the last estimated actual state $\tilde{x}$ in the automaton. The first residual is

$$Res1(\tilde{x},u(t)) = ES(\lambda(\tilde{x}),u(t)) \setminus \bigcup_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x')). \quad (2)$$

With $ES(\lambda(\tilde{x}),u(t))$ the rising and falling edges are determined that are observed when comparing the I/O vector of the last estimated current state and the I/O vector that led to fault detection. This set represents what actually happened when the fault was detected and refers to the notion *measurements* in the residual definition. $\bigcup_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x'))$ represents the union of the sets of rising and falling edges when the last estimated current state and each of its direct successor states are considered. It represents the expected behavior. This refers to the *model-based computation* in the residual definition. The set difference of the observed ($ES(\lambda(\tilde{x})$, $u(t)$)) and the expected ($\bigcup_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x'))$) behavior is built in the residual equation and thus represents the *unexpected* behavior. In *Res*1, the expected behavior is given by the union of each possible following behavior of the last estimated current state. A stricter formulation of the expected behavior is used in the second residual:

$$Res2(\tilde{x},u(t)) = ES(\lambda(\tilde{x}),u(t)) \setminus \bigcap_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x')). \quad (3)$$

Instead of a union over the expected behavior of the possible following states, an intersection is used. The intersection delivers the I/O edges that *must* be observed no matter which following state in the model is taken. It is obvious that $Res1 \subseteq Res2$ since $\bigcup_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x')) \supseteq \bigcap_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x'))$.

Example. Fig. 9 shows an example for unexpected behavior that led to fault detection (instead of the I/O vectors only the I/O edges between two states are given): from the estimated actual NDAAO state x_1 it is not possible to take a transition that has exactly the observed falling edges IO_3_0 and IO_4_0. Hence a fault is detected.

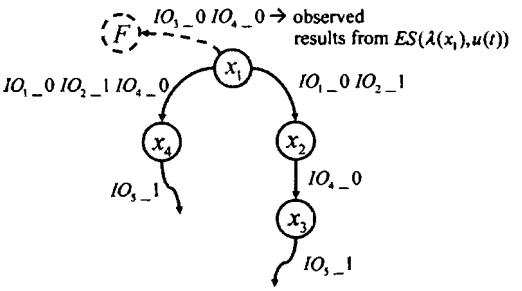

Fig. 9. Example for an unexpected behavior.

The result of *Res*1 (with $\tilde{x} = x_1$) is

$$Res1(\tilde{x},u(t)) = \{IO_3\_0, IO_4\_0\} \setminus (\{IO_1\_0, IO_2\_1, IO_4\_0\} \cup \{IO_1\_0, IO_2\_1\}) = \{IO_3\_0\}.$$

This result means that IO_3_0 was an unexpected event which implies that the system operator should check the sensor or actuator that is connected with IO_3. However, it is possible that the fault cannot be found at this component. If this is the case, *Res*2 should be calculated in order to use a stricter formulation of the expected behavior. In *Res*1 each possible following behavior is subtracted from the observation. Using *Res*2 only the behavior that must occur no matter which regular following behavior is considered:

$$Res2(\tilde{x},u(t)) = \{IO_3\_0, IO_4\_0\} \setminus (\{IO_1\_0, IO_2\_1, IO_4\_0\} \cap \{IO_1\_0, IO_2\_1\}) = \{IO_3\_0, IO_4\_0\}$$

This result implies that the occurrence of a change in value of IO_4 is not always expected and thus another possible fault localization.

4.3.2. Missed behavior

In contrast to an observed but unexpected behavior it is also possible that a fault can be localized by determining a missed event. Set operations that help to localize an expected but unobserved behavior are given by the third and the fourth residual:

$$Res3(\tilde{x},u(t)) = \bigcap_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x')) \setminus ES(\lambda(\tilde{x}),u(t)). \quad (4)$$

*Res*3 is the set difference of the I/O edges that are expected no matter which following state is taken ($\bigcap_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x'))$) and the I/O edges that have been observed ($ES(\lambda(\tilde{x}),u(t))$). Each rising or falling edge that must occur when the estimated actual state is left but has not been observed is part of *Res*3. The expected behavior is represented by the intersection of each possible following behavior. It is also possible to give a less strict formulation of the expected behavior by using the union operation instead of the intersection:

$$Res4(\tilde{x},u(t)) = \bigcup_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x')) \setminus ES(\lambda(\tilde{x}),u(t)). \quad (5)$$

Since $Res3 \subseteq Res4$, the result of *Res*4 is usually less restrictive than *Res*3, it contains more elements.

Example. The situation in Fig. 10 is considered to illustrate *Res*3 and *Res*4. The observation of the edge IO_3_0 leads to fault detection with x_1 as estimated actual NDAAO state. Applying the residuals results in $Res3 = \{IO_1\_0\}$ and $Res4 = \{IO_1\_0, IO_2\_1\}$. The same procedure as explained for *Res*1 and *Res*2 should be started: First, the component connected with IO_1 should be checked. If the system operator does not find a fault at this component, he should proceed with checking IO_2 which is additionally part of *Res*4.

Especially if production systems with many I/Os are considered, the residuals can help to get a relatively small set of I/Os that could be related to the fault. A maintenance operator can

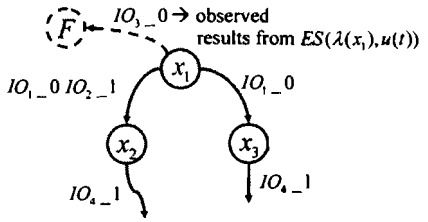

Fig. 10. Example for a missed behavior.

then check the possibly faulty sensors or the related actuators. Actuators are related to a sensor if their activation or deactivation can have an influence on the sensor value.

4.4. Reduction of the residual sets

Once a fault has been detected, the four residuals are available to localize controller I/Os that are related with the fault. It has been explained that the residuals analyze two different generic fault symptoms: missed and unexpected behavior. In general, it is not a priori possible to tell if a fault was caused by missed or by unexpected behavior. Hence, all residuals must be calculated which results in a set of possible fault candidates. The system operator has to check each of the I/Os according to the priority explained before. This operation can be supported by the information if the considered I/O is suspected to have shown a missed or an unexpected behavior. In order to give a more precise estimation of which I/O is possibly affected, it is possible to take further I/O vectors into account that follow the vector that led to fault detection. Of course, this approach is only possible if the system does not immediately have to be stopped after fault detection like for fault tolerant systems. The reduction of the fault candidate set is based on the following heuristic:

> After a fault has been detected it is possible to delete a fault candidate from the candidate list if it shows a following behavior that can reasonably be considered as fault-free.

To decide if the following behavior of a possibly affected I/O can reasonably be considered as fault-free, a special state estimation algorithm is proposed. The idea of this algorithm is to mask out the possibly faulty part of the I/O vector and to perform state estimation only based on the 'healthy' I/Os. This state estimation results in a set of states that represent possible descriptions for the non-faulty part of the current system state. If a possibly affected I/O has the same value in each of these states and in the currently observed I/O vector, it is assumed that this can be considered as the result of a fault-free I/O behavior. The I/O showed a behavior that is completely explicable with fault free system states which have been determined without considering the I/O itself. Hence this I/O can be deleted from the fault candidate list.

The candidate set of all I/O edges that are reported by the residuals is given by

$$Res_{\cup}(\tilde{x},u(t)) = ES(\lambda(\tilde{x}),u(t)) \cup \bigcup_{\forall x' \in f(\tilde{x})} ES(\lambda(\tilde{x}),\lambda(x')). \tag{6}$$

It is obvious that $Res_{\cup} = Res1 \cup Res2 \cup Res3 \cup Res4$. In order to get the controller I/Os that have edges in the candidate set the following function is applied:

$$IOList = \{IO_i | (IO_i\_1 \vee IO_i\_0) \in Res_{\cup}\}. \tag{7}$$

This *IOList* contains the part of the system I/O vector that is possibly influenced by the fault based on the result of the residuals. The possibly faulty I/Os from *IOList* can create the following edges:

$$CandEdges = \{IO_i\_1, IO_i\_0 \in E | IO_i \in IOList\}. \tag{8}$$

Since the state estimation algorithm will work on the basis of the non-affected part of the I/O vector, this part has to be calculated by an I/O vector projection:

Definition 9 (*I/O vector projection*). The I/O vector projection of an I/O vector u to a list of I/Os (*IOList*) that is to be masked out is defined as

$$IOP_{IOList}(u)[i] = \begin{cases} u[i] & \text{if } IO_i \notin IOList, \\ * & \text{if } IO_i \in IOList \end{cases}$$

$\forall i = 1, \ldots, m$ (with m denoting the number of controller I/Os in vector u).

$$IOP_{IOList}(u) = \begin{cases} IOP_{IOList}(u)[1] \\ \vdots \\ IOP_{IOList}(u)[m] \end{cases}$$

applies the function to the entire vector. $IOP_{IOList}(u)$ contains for each I/O that has to be masked out (i.e. an I/O in the *IOList*) the do not care symbol $*$. For each I/O that has to be considered, $IOP_{IOList}(u)$ has the according I/O value in vector u.

Algorithm 2. State estimation and candidate set reduction.

Require *IOList*, *CandEdges*, X

{Initialization}

1: $\tilde{X}_{t-1} \leftarrow X$

{State estimation}

2: Wait for the next observed I/O vector denoted as $u(t)$

3: $\tilde{X}_t \leftarrow \{\tilde{x} \in \tilde{X}_{t-1} \mid IOP_{IOList}(\lambda(\tilde{x})) = IOP_{IOList}(u(t))\}$

4: $\tilde{X}_t \leftarrow \tilde{X}_t \cup \{x' \in X \mid \exists \tilde{x} \in \tilde{X}_{t-1} : x' \in f(\tilde{x}) \wedge IOP_{IOList}(\lambda(x')) = IOP_{IOList}(u(t))\}$

5: $\tilde{X}_{Reach} \leftarrow \{x(i+n) \in X | \exists (x(i), \ldots, x(i+n)) : x(i) \in \tilde{X}_t \wedge \forall i \le j \le i+n : (x(j+1) \in f(x(j)) \wedge ES(\lambda(x(j)), \lambda(x(j+1))) \subseteq CandEdges)\}$

6: $\tilde{X}_t \leftarrow \tilde{X}_t \cup \tilde{X}_{Reach}$

7: $\tilde{X}_{t-1} \leftarrow \tilde{X}_t$

{Candidate set reduction}

8: **for all** $IO_i \in IOList$ **do**

9: **if** $|\tilde{X}_t| > 0 \wedge \forall x \in \tilde{X}_t : \lambda(x)[i] = u(t)[i]$ holds **then**

10: $IOList \leftarrow IOList \backslash IO_i$

11: **end if**

12: **end for**

{recalculate CandEdges}

13: $CandEdges = \{IO_i\_1, IO_i\_0 \in E | IO_i \in IOList\}$

{Analyze result}

14: **if** $|\tilde{X}_t| > 0$ and $|IOList| = 0$ **then**

15: (probably) false alert → end algorithm and start Algorithm 1 with $Res1\text{-}4 = \{\}$ and $\tilde{X}_{t-1} = \tilde{X}_t$

16: **end if**

17: **if** $|\tilde{X}_t| > 1$ and $|IOList| > 0$ **then**

18: go back to line 2

19: **end if**

20: **if** $|\tilde{X}_t| = 0$ **then**

21: no state estimation possible

22: **end if**

With these sets it is possible to perform Algorithm 2 instead of Algorithm 1 in order to estimate the current system state. Since the algorithm starts with considering the first I/O vector after fault detection, the former state estimate is empty and must be reinitialized. Each NDAAO state must be considered as possible estimate (line 1). When the next I/O vector is observed it is checked which NDAAO state of the former estimate is still conform with the 'healthy' part of the new I/O vector (line 3). Possibly

affected I/Os do not contribute any information to determine the new estimate $\tilde{X}_t$. After this, NDAAO states are determined which can be reached starting in the former estimate $\tilde{X}_{t-1}$ by considering the I/O vector projection of the new I/O vector. Each following state of the former estimate $(x' \in f(\tilde{x}) | \tilde{x} \in \tilde{X}_{t-1})$ that has the same 'healthy' part of the I/O vector as output as it has been observed is added to the new estimate (line 4). Since the state estimation should only be restricted with the non-affected part of the I/O vector, the set $\tilde{X}_{Reach}$ is calculated in line 5. It contains each NDAAO state that can be reached starting in the current estimation $\tilde{X}_t$ if during the state trajetory only possibly affected I/O edges occur (*CandEdges*). This set is added to the current estimate. This ensures that the state estimation is not blocked by I/Os that are possibly related to the fault. In line 7, the set $\tilde{X}_{t-1}$ for the analysis of the next I/O vector is prepared.

After the state estimation part is finished, the candidate reduction starts. For each I/O in the *IOList* it is checked if it has the same value in each state of the estimate and in the currently observed I/O vector. If this is true, the I/O gets removed from the *IOList* since the currently observed I/O value is explicable with each estimated NDAAO state. Since the estimated states were determined only using the 'healthy' part of the I/O vector, it is assumed that it is highly probable that the considered I/O is not faulty as it fits well in each fault-free interpretation of the current system state. After the *IOList* has been updated, the *CandEdges* set must also be recalculated. Then, the calculated sets are interpreted. If the *IOList* is empty, the fault detection was probably a false alert since the current observation can perfectly be explained by the estimated fault-free model states. Most often, in this case $|\tilde{X}_t| = 1$ also holds. Hence, the normal observation can go on (Algorithm 1). If at least one I/O is left in the *IOList* and $\tilde{X}_t$ is not empty, the algorithm loops back and waits for the next I/O vector. If it was not possible to determine a state estimation, the algorithm cannot proceed and stops. The operator must check the remaining I/Os. Note that the number of candidates in *IOList* is typically much smaller than the number of I/Os in the system: Before the algorithm is started, *IOList* is calculated according to Eq. (7) as the union of I/Os contained in the residuals. Processing further I/O vectors, *IOList* can only be reduced in the algorithm as can be seen from line 8 to 12.

The algorithm only calculates a new state estimation and an updated candidate set if a new I/O vector is observed. The time to recalculate these two sets depends thus on the time between the observation of the new I/O vectors and cannot be calculated a priori. Since the algorithm has to compare each value of an I/O in an observed vector with the according I/O in the model states, the complexity of the algorithm with respect to the number of I/Os is linear.

5. Case study

5.1. System description

In order to demonstrate the relevance of the proposed method for existing closed-loop DES, a case study is presented in this section. The considered system is a laboratory facility at the Institute of Automatic Control, University of Kaiserslautern. The purpose of the system depicted in Fig. 11 is to treat work pieces that are stored in the feeder (left most station) with three machine tools. The system is controlled using a Siemens S300 PLC (Programmable Logic Controller) with 14 inputs and 15 outputs. The controller inputs and the corresponding sensors can be seen in Fig. 11. Inputs that are written in *italic* are connected to sensors with a specific technology that delivers a logical 0 if they detect something and a logical 1 if they do not detecting anything. All sensors except of those for work piece positions have this technology. The outputs are given in Table 1. If they are set to 1, the according actuator is activated. In order to make the I/O names easier to read they are not labeled IO_i

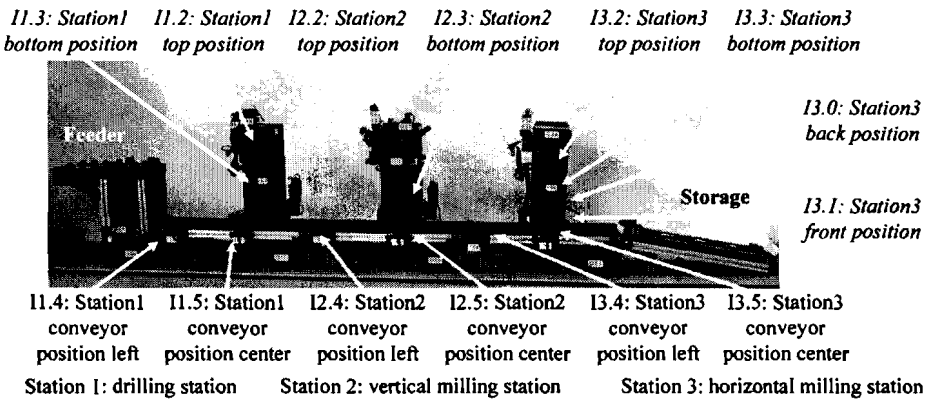

Fig. 11. Lab system.

Table 1
Controller outputs of the case study.

I/O	Description	I/O	Description
O1.2	Drilling machine motor up	O2.2	Vertical milling machine motor up
O1.3	Drilling machine motor down	O2.3	Vertical milling machine motor down
O1.4	Drilling machine drilling motor	O2.4	Vertical milling milling motor
O1.5	Drilling machine conveyor on	O2.5	Vertical milling machine conveyor on
		O2.7	Change milling head
O3.0	Horizontal milling machine motor back	O3.1	Horizontal milling machine motor front
O3.2	Horizontal milling machine motor up	O3.3	Horizontal milling machine motor down
O3.4	Horizontal milling machine milling motor	O3.5	Horizontal milling machine conveyor on

but with I1.2 etc. to indicate the second *input* that belongs to the first machine and O2.4 for the fourth *output* of the second station.

A production cycle (or system evolution in terms of Section 3) consists of the treatment of two work pieces that are stored in the feeder. At the beginning of a system evolution the feeder pushes the first work piece to the conveyor. Then it is transported to the drilling station (station 1). In the drilling station the work piece is stopped and two holes are drilled. After this, the work piece is transported to the vertical milling machine (station 2) and the next piece is taken from the feeder and transported to station 1. The first work piece gets treated by the vertical milling machine (each of its three milling tools is applied to the piece) and the second work piece by the drilling station. After the treatment at the vertical milling machine has finished the according work piece is transported to the horizontal milling station (station 3). The second work piece moves from the drilling station to the vertical milling machine. When the work piece has been treated in the horizontal milling machine, it is stored in the storage at the right side in Fig. 11. This process continues until both work pieces have been treated by each of the three stations.

5.2. Data collection for identification and online diagnosis

As explained in Section 2, the proposed method works on the basis of signals exchanged between controller and plant (see also Fig. 3). The data collection method that was chosen for the case study works with a standard communication processor that is part of the PLC. When controlling the system evolution, the PLC performs the following three steps cyclically: reading the inputs, executing the program in order to determine the new output setting and finally writing the outputs. The transfer of the I/O vector to the diagnosis computer takes place at the end of the second step (program execution). At the same time when the newly determined outputs are transfered to the output card of the PLC, the new I/O vector is sent to a standard PC using the communication processor via a UDP-connection. Since the I/O vectors are captured at the end of the PLC cycle and are thus sampled, it is possible to have different I/Os changing their value between two I/O vectors. This method can be applied to each system with a controller that is able to send data via an Ethernet data link which is a common feature of almost all industrial controllers today.

For the application of the proposed diagnosis method it is necessary to have a model that describes the fault-free behavior of the considered closed-loop system. For the case study system such a model has been identified using the method outlined in Section 3. The data base for the identification consists of 60 fault-free system evolutions (60 treatments of two work pieces) that have been observed using the data link described above. The evolution of the according observed languages (sets of I/O vector sequences of varying length) are shown in Fig. 4 in Section 3.3. It has been explained that with $k=2$ a model can be identified which is not expected to deliver false alerts during online diagnosis. The identified automaton has 121 states and 162 transitions. Fault-free and faulty system behavior can be distinguished since behavior that has not been seen during the model learning phase is considered as a fault symptom. Parts of the identified NDAAO can be seen in Figs. 12 and 13. Since the state output $\lambda(x)$ is too large to be depicted (I/O vector with 29 I/Os), only the evolution set that is obtained when comparing the I/O vectors of two connected states is shown.

5.3. Fault localization examples

The first example is a fault at the sensor connected with controller input I3.3. During normal operation, this controller input changes its value from 1 to 0 when the head of the horizontal milling station reaches the bottom position. The fault that has been introduced artificially is a short circuit causing the sensor to switch its signal from 1 to 0. The fault has been produced when the first work piece was treated by the second machine (vertical milling). At the same time, the second work piece gets drilled in the first station. In this situation no work piece is in front of the horizontal milling station and its milling head is in its home position at the top. Hence, none of the inputs belonging to station 3 should change their value. The short circuit has been introduced when milling with the second of the three tools in station 2 started and the vertical milling head left the top position. This situation is depicted in Fig. 12. After the observation of I2.2_1 indicating that the milling head just left its home position, the evaluator algorithm delivers state x_{18} as single state estimate. Hence, no fault is detected. Then the falling edge I3.3_0 is observed that occurs due to the fault. The evaluator algorithm does not find a following state of x_{18} that can be reached by this edge. Since $\tilde{X}_t$ is empty, a fault is detected due to Eq. (1). Consequently the residuals are applied with $\tilde{x}=x_{18}$ as the former unique estimate:

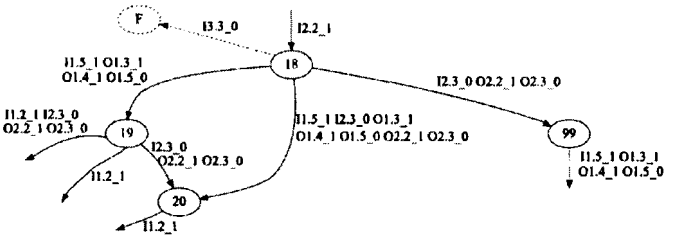

Fig. 12. Example for a fault at I3.3.

$$Res1(\tilde{x},u(t)) = \{I3.3\_0\},$$

$$Res2(\tilde{x},u(t)) = \{I3.3\_0\},$$

$$Res3(\tilde{x},u(t)) = \{\},$$

$$Res4(\tilde{x},u(t)) = \{I1.5\_1,I2.3\_0,O1.3\_1,O1.4\_1,O1.5\_1,O2.2\_1,O2.3\_0\}.$$

Res1 and Res2 both result in the same unexpected edge that is caused by the fault. Res3 results in an empty set and Res4 shows a union of the legal following behavior of state x_{18}. In this case, Res3 and Res4 do not contribute to fault localization. Res1 and Res2 provide the "right" candidate for fault localization.

The second example is a fault at the sensor connected to I2.4. This sensor detects if a work piece is at the left most position of station 2 (I2.4=1). The artificially introduced fault prevents the sensor from switching back to 0 when the work piece has left the left most position. The fault was introduced when the first work piece gets transported from station 2 to station 3 and the second work piece from station 1 to station 2. Fig. 13 shows two parts of the identified NDAAO. The lower part represents the described situation. The evaluation algorithm (Algorithm 1) determines x_{49} as the actual estimate when the second work piece arrives at position I2.4 and produces the rising edge I2.4_1. Now, the fault prevents I2.4 from switching again to 0 when the work piece leaves the position due to the start of the conveyor (O2.5_1). The fault is detected when the rising edge of I3.5 and its corresponding output changes are observed before I2.4 changed back to 0. The residuals are calculated:

$$Res1(\tilde{x},u(t)) = \{I3.5\_1,O3.3\_1,O3.4\_1,O3.5\_0\},$$

$$Res2(\tilde{x},u(t)) = \{I3.5\_1,O3.3\_1,O3.4\_1,O3.5\_0\},$$

$$Res3(\tilde{x},u(t)) = \{I2.4\_0,O1.5\_0\},$$

$$Res4(\tilde{x},u(t)) = \{I2.4\_0,O1.5\_0\}.$$

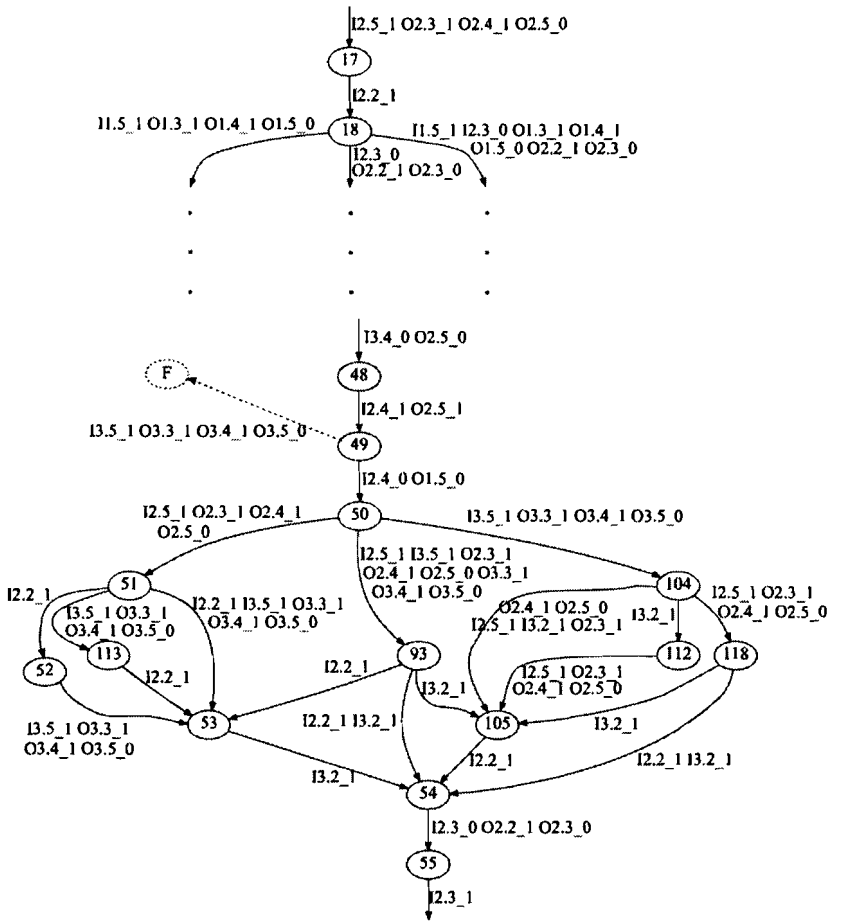

Fig. 13. Example for a fault at I2.4.

In this case, Res1 and Res2 do not contribute to fault localization since they show the observed behavior that is not responsible for the fault. The controller input related to the blocked sensor is localized by both Res3 and Res4 since it is a missed behavior.

A fault that leads to exactly the same symptoms is a defect at the motor of the conveyor in front of the vertical milling machine (*O2.5*). If the conveyor motor does not start due to the motor fault, the symptoms are comparable: The sensor connected to *I2.4* does not change its value since the conveyor does not transport the work piece away from the left most position of station 2. Hence, if Res3 or Res4 report a missing change in value at a sensor, it is reasonable to not only check the according sensor but also the actuators that have a direct influence to this sensor.

In the second example, Res3 and Res4 contained the controller input that was connected to the faulty sensor or the sensor that did not change its value due to a fault at a related actuator. Additionally, the residuals contain controller outputs. These controller outputs are set if a controller input leads to a fulfilled logical condition in the control program and are usually not directly related to the fault. Nevertheless, they can be useful to decide if the fault candidates reported by Res1 and Res2 or the candidates of Res3 and Res4 should be analyzed at first. If controller inputs appear in Res1 or Res2, they are supposed to be unexpected. Indeed, the fault-free system model did not expect their change in value at the time of its observation. Nevertheless, if a change in value of an input together with change in values of controller outputs is observed, it is possible that this was not completely unexpected: The controller program was in an internal state where it waited for the change in value of the input and thus triggered some outputs upon its observation. Given a situation like in the second example, this can help to decide that the observed change in value of I3.5 was not completely unexpected by the controller since it was in an internal state that allowed setting corresponding outputs. Hence, the candidates delivered by Res3 and Res4 should be checked first which delivers the right fault candidate.

Apart from using this strategy to decide which candidate should be analyzed first, the candidate set reduction algorithm presented in Section 4.4 can be applied. As explained before, it must be possible to run the system even if a fault was detected. This is a scenario that is often possible in industrial system like e.g. the system in Supavatanakul, Lunze, Puig, and Quevedo (2006). As an example to illustrate Algorithm 2 we consider the fault of the second example (I2.4 does not switch back to 0). Based on the residuals calculated before, the combined set $Res_{\cup}$ can be determined to

$$Res_{\cup}(\check{x},u(t)) = \{I3.5\_1, I2.4\_0, O3.3\_1, O3.4\_1, O3.5\_0, O1.5\_0\}$$

according to Eq. (6).

Algorithm 2 is started to reduce the candidate set and to estimate the current system state using the next observed I/O vector following fault detection. In line 1, the complete NDAAO state space is assigned to the former estimate. The I/O vector that is observed after fault detection leads to the evolution set {I2.5_1, O2.3_1, O2.4_1, O2.5_0} when compared to the former I/O vector. The I/O vector projection of the current vector and the state outputs of the former estimate (line3) leads to the following estimated states: x_{17} (upper part of Fig. 13), x_{51}, x_{93}, x_{113} and x_{118}. This estimate remains unchanged in line 4. $\check{X}_{Reach}$ representing the

states that can be reached starting in one of the states of the current estimate and only producing edges of possibly affected I/Os is determined to state x_{113}. Starting from x_{51} it is possible to reach x_{113} only producing possibly affected I/Os. Since x_{113} is already part of the current estimate it does not change in line 6. Comparing the observed values of the possibly affected I/Os with the values in each of the estimated states reveals that none of the I/Os can yet be erased from *IOList*. Since $|\tilde{X}_t| > 1$ and $|IOList| > 0$ the algorithm loops back to line 2.

The next observed I/O vector leads to the edge I3.2_1 when compared with its predecessor. Since I3.2 is not part of the possibly affected I/Os, it does not have the observed value 1 in one of the states from the former estimate. Hence, $\tilde{X}_t$ becomes an empty set in line 3. As state x_{105} has the same I/O vector projection as the currently observed I/O vector and $x_{105} \in f(x_{93})$ (x_{93} was part of the former estimate), the current estimate is determined to $\{x_{105}\}$ in line 4. The operation of line 5 results in an empty set. Hence, the current estimate consists only of one state. The candidate set reduction from line 8 to 12 reveals that only the I/Os I2.4 and O1.5 differ in the observed vector and in the estimated state. The other fault candidates can thus be removed from *IOList* since they showed a behavior that can be considered as normal. Since $|\tilde{X}_t| > 1$ and $|IOList| > 0$ the algorithm loops back to line 2 but has reduced the number of fault candidates. Hence the fault must be searched at the sensor that is connected to I2.4 which corresponds to the artificially introduced fault.

6. Conclusions and outlook

In this paper a fault localization technique working on the basis of a fault-free nominal system model has been presented. The concept of residuals known from fault diagnosis in continuous systems has has been introduced to solve the fault localization problem in DES. It has been shown how an identified model can be used for fault detection and fault localization. Results from a case study that represents typical characteristics of industrial production systems show that the residual approach is able to deliver a small subset of controller I/Os possibly related to a detected fault. Using a special candidate set reduction algorithm it is possible to reduce this set of fault candidates. The presented approach was developed for monolithic system models. In large applications, automata networks are often used for diagnosis purposes following a decentralized approach. After the development of an appropriate distributed identification approach in Roth, Jean-Jacques, and Litz (2010b), the fault localization approach will be extended for the application in automata networks in future works.

Many faults are characterized by an altered *timed* behavior. Typical examples are actuator faults often leading to deadlocks. Analyzing the timed behavior for diagnosis purposes necessitates using timed models. Since the determination of timed models is very demanding, current work concentrates on identifying the timed behavior. After the development of appropriate identification algorithms, future works aim at extending the residual approach for timed models.

References

Cassandras, C. G., & Lafortune, S. (2006). *Introduction to discrete event systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Cordier, M., Dague, P., Levy, F., Montmain, J., Staroswiecki, M., & Trave-Massuyes, L. (2004). Conflicts versus analytical redundancy relations: A comparative analysis of the model based diagnosis approach from the artificial intelligence and automatic control perspectives. *IEEE Transactions on Systems, Man and Cybernetics Part B—Cybernetics, 34*(5), 2163–2177.

Dotoli, M., Fanti, M. P., Mangini, A. M., & Ukovich, W. (2009). On-line fault detection in discrete event systems by Petri nets and integer linear programming. *Automatica, 45*(11), 2665–2672.

Fanti, M. P., & Seatzu, C. (2008). Fault diagnosis and identification of discrete event systems using Petri nets. In *Proceedings of the ninth international workshop on discrete event systems* (pp. 432–435), WODES 08, Göteborg, Sweden.

Hashtrudi Zad, S., Kwong, R. H., & Wonham, W. M. (2005). Fault diagnosis in discrete-event systems: Incorporating timing information. *IEEE Transactions on Automatic Control, 50*(7), 1010–1015.

Isermann, R., & Balle, P. (1997). Trends in the application of model based fault detection and diagnosis of technical processes. *Control Engineering Practice, 5*(5), 709–719.

Klein, S. (2005). *Identification of discrete event systems for fault detection purposes*. Shaker Verlag.

Moor, T., Raisch, J., & Young, S. (1998). Supervisory control of hybrid systems via l-complete approximations. In *Proceedings of the IEE fourth workshop on discrete event systems WODES98* (pp. 426–431), Cagliari, Italy.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence, 32*(1), 57–95.

Roth, M., Jean-Jacques, L., & Litz, L. (2010a). Identification of discrete event systems—implementation issues and model completeness. In *Proceedings of the seventh international conference on informatics in control, automation and robotics (ICINCO)*, Funchal, Portugal, June 15–18.

Roth, M., Jean-Jacques, L., & Litz, L. (2010b). Black-box identification of discrete event systems with optimal partitioning of concurrent subsystems. In *Proceedings of the 2010 American control conference (ACC2010)* (pp. 2601–2606), Baltimore, USA, June 30–July 2.

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., & Teneketzis, D. (1996). Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology, 4*(2), 105–124.

Sayed-Mouchaweh, M., Philippot, A., & Carre-Menetrier, V. (2008). Decentralized diagnosis based on Boolean discrete event models: Application on manufacturing systems. *International Journal of Production Research, 46*(19), 5469–5490.

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