# A model-based approach to Fault diagnosis in Service oriented Architectures

Mohammed Alodib and Behzad Bordbar
*School of Computer Science*
*University of Birmingham*
*United Kingdom*
*M.I.Alodib,B.Bordbar@cs.bham.ac.uk*

***Abstract*—This paper aims to present a method of creating architectures which allow monitoring occurrence of failure in Service oriented Architectures (SoA). The presented approach extends Discrete Event Systems techniques to produce a method of automated creation of Diagnoser Service which monitors interaction between the services to identify if a failure has happened and the type of failure. To do so, a formal representation of business processes is introduced, which allows modeling of observable/unobservable events, failure and the type of failure. The paper puts forward a set of algorithms for creating models of Diagnoser Service. Such models are then transformed into new Services implemented in BPEL, which interact with the existing services to identify if a failure has happened and the type of failure. The approach has been applied to an example of diagnosis of Right-first-time failure in Services used in telecommunications.**

## I. Introduction

Service Oriented Architecture (SoA) aims at the implementation of business processes as a composition of interacting services [1]. In a real-world business processes, it is crucial to develop architectures for identifying occurrences of failure, so that suitable courses of actions can be adopted to deal with the failure. One of the common methods of identifying failure is to create Diagnoser, software modules or services, which monitor the interaction between the services to identify if a failure has (or may have) happened [2], [3], [4], [5], [6], [7], [8]. Some of these methods suggested for the design of Diagnoser Service are model-based, extending classical approaches to Diagnosability in Discrete Event Systems (DES) to SoA [9], [10], [11], [12], [13], [14].

Methods suggested for the design of Diagnoser by DES community are mostly reliant on the representations such as automata [15] or Petri net [16], [17]. Users of such methods must transform models of the systems, which are often captured at higher-levels of abstraction, into lower level models in automata and Petri net to utilize DES techniques [8], [3], [6], [7]. This requires a substantial bridging of the gap between different levels of abstractions. In this paper, we adopt Workflow Graphs [18] as a language for specifying models of business processes. Further, we will extend conventional Workflow Graph notation to model observable, unobservable and failure events. Using Workflow Graph formalism, we shall present a set of algorithms for designing Diagnoser Service. The presented formalism directly maps into business process models used in industry which are implemented in popular tools. In our approach the Diagnoser which will be also captured in workflow graphs, will be implemented as a service called a Diagnoser Service. The Diagnoser Service will interact with existing services to identify if a failure has (or may have) happened and the type of failure.

This paper also studies various methods of incorporating the Diagnoser Service into the interacting services. There is a choice to implement the Diagnoser Service as a BPEL service or as dedicated Java Class deployed as Web Service. This amounts to four styles of creating the Diagnoser Service. This paper also reports on tool extending Oracle JDeveloper which produces all four types of Diagnoser and studies them from the performance point of view. The evaluation has been performed with the help of stress testing facilities provided by the Oracle Application Server.

The paper is organized as follows. Section II-A presents a brief review of the business process model, which is further extended in section III to model Observerable/Unobservalbe events, failure and failure types. In Section II-B a running example based on a real-world scenario of failure detection in SoA used in telecommunication systems is presented. The example deals with Right-First-Time failures, in which the Customer Support System fails to complete a task First-Time and is forced to repeat part of the task again. This type of failure is important as occurrence of such failure may cause extra costs and delays in the completion of the tasks, causing a violation of Service Level Agreements. The presented approach has two steps. Firstly, in section IV, an algorithm for creating a graph called Coverability Graphs is presented. Such graphs extend the idea of Petri net Coverability graphs [14]. From this point of view our approach furthers the method suggested by Giua and Seatzu [14], [19] in designing Diagnoser for Petri net models. In section V, a second algorithm is presented which produces the Workflow Graph Diagnoser. Finally in section VI, an outline of a method of transforming the Workflow Graph to an Business Process Execution Language (BPEL) services, which interact with the existing services within the architecture to identify occurrence of the failure, will be presented. The section also describes implementation of the presented approach as a plugin to Oracle JDeveloper. In addition, different styles of incorporating the Diagnoser Service will be presented. It also shows the results

related to the comparison these methods. The paper ends with a discussion and comparison of existing approach with the presented method.

## II. Preliminaries

### A. Business Process Execution Language (BPEL)

There is an ever-increasing pressure on modern enterprisers to adapt to the changes in their environment by evolving to respond to any opportunity or threat [20]. Service oriented Architecture (SoA) provides the foundation for implementing business processes via the composition of existing services. Web services [21] are software systems which make use of well-accepted standards and XML languages to support the creation of SoAs. For example an interaction, a task can be captured in Business Process Execution Language [1]. BPEL can be used to express complex sequential, parallel, iterative and conditional interactions. The type for all messages and variables used in BPEL file are defined via XML Schema Definition (XSD) [22], usually in WSDL file [21].

### B. Example: Right-First-Time failure

Consider a simplified interaction between a customer and a number of services in a typical Telecommunication Company for technical support related to the Broadband connection.

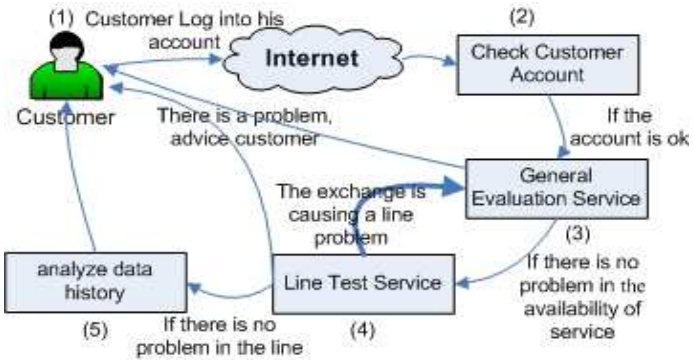


Fig. 1. An interaction between the Customer and System

As depicted in Figure 1, the customer logs [1] onto the company website and enters details such as the account number. Choosing the "Broadband problem" option, he submits his form online. Next, the company's Check Customer Account (CCA) service determines whether the customer account is in a satisfactory condition in order to progress the fault report. If the current status of the account is not satisfactory the customer is advised to phone the call center and the process ends. If the account status is satisfactory, the CCA invokes a request to another service called General Evaluation Services (GES). The GES examines the availability of service at the exchange side and ensures that everything is up and running, in which case the process moves to the next step. If GES identifies any problem with the availability of the services at the exchange side, the customer is informed of the status and a separate process is invoked to deal with this problem (not shown as part of this example). If everything is fine on the exchange side, the Customer Services sends a request to Line Test Service (LTS), which is an automated service to check line status up to the customer premises. However, LTS can also indicate problems on the exchange side which were not detected by the GES. There are three possible outcomes: 1) the line has no problem, move to next step, 2) the line has some problems, advice the customer or 3) There is no problem with the line, although there is a likely problem with the exchange. Option 3 is shown in bold arrow in Figure 1. If the case 3 happens, a failure emerges which means that GES should repeat its course of action violating Right-First-Time. Finally, LTS sends a request to analyze data history in the customer router. If it is possible to carry out analysis then get a decision from the analysis algorithm (either all ok so the customer has to call technical support, or the analysis finds the problem and customer is advised what to do).

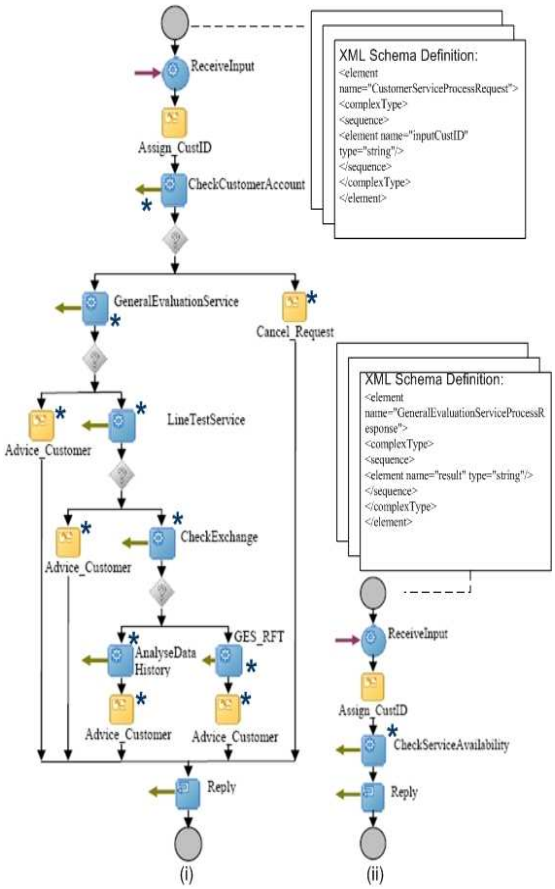


Fig. 2. Customer Service BPEL

Following the lead of Jussi et al. [18] in the next section a formal representation will be explained that can capture the interaction between services as a Workflow Graph

[1]We assume that the Customer can log into the company's website, for example suppose the customer is not happy with the speed of his Broadband connection

### C. Workflow Graphs

One of the most complex aspects of a business process such as the one explained in the above example is the specification of the interaction between the services resulting from the execution of actions and exchange of messages. We aim to rely on a models that can capture the flow of the action and message exchanges similar to models used in tools such as Oracle JDeveloper and Websphere. The aim is not to support all complex and elaborate constructs supported by such tools; however we are interested to capture an essential core of high-level interactions such as sequential, parallel and decision behavior. To do so we adopt a model suggested at [18], which is formalize as follows:

*Definition 1:* A Workflow Graph is a graph $G = (N, E)$, where each node $n \in N$ is one of the following types: Start node, Stop node, Activity, Fork, Join, Decision and Merge. The unique outgoing edge of a Start node is called the *entry edge*, and the unique incoming edge of a Stop node is called the *exist edge*[18]. For each node $n \in N$, the set of incoming/outgoing edges of $n$ are denoted by $In(n)$/$Out(n) \subset E$.

**Semantics of Workflow Graph:** Following the lead of Petri Nets, Workflow Graphs represent a flow of actions which is captured by movement of tokens. The definition of state in Workflow Graph extends the definition of marking in Petri nets [23].

*Definition 2:* A *state* of Workflow Graphs defined as a vector $s = (s_1, \ldots, s_k)$ where $s_i$ is the number of tokens on each edge $e_i \in E$.

For a state $s$, we may write $s_i$, $s(i)$ or $s(e_i)$ to represents the number of tokens in the edge $e_i$, i.e $i$-th is the coordinate of vector $s$. We shall also sometimes refer to the edge $e_i$ as "edge $i$"

An occurrence of an *action*, which is modeled via the edges of the Workflow Graphs denoted by $s \xrightarrow{n} s'$, which means that occurrence of the action $n$ has resulted a change of state from the state $s$ to $s'$. Such change states modifies the number of tokens on the edges in $s$ to obtains $s'$

*Definition 3:* (Change of States) Assume that $s, s'$ are two states of a Workflow Graph such that $s \xrightarrow{n} s'$, then for $1 \leq i \leq k$:

- if $n$ is an activity, fork or join:

$$s'_i = \begin{cases} s_i - 1 & \text{if } e_i \in In(n) \\ s_i + 1 & \text{if } e_i \in Out(n) \\ s_i & \text{otherwise} \end{cases}$$

- if $n$ is a decision and there exists an outgoing edge $e'$ of n:

$$s'_i = \begin{cases} s_i - 1 & \text{if } e_i \in In(n) \\ s_i + 1 & e_i = e'_i \\ s_i & \text{otherwise} \end{cases}$$

- if $n$ is a merge and there exists an incoming edge $e'$ of n:

$$s'_i = \begin{cases} s_i - 1 & e_i = e'_i \\ s_i + 1 & \text{if } e_i \in Out(n) \\ s_i & \text{otherwise} \end{cases}$$

In the above definition, for example if $n$ is a fork one token from each input edge $e \in In(n)$ is removed and one token to each output edge $e \in Out(n)$ is added. The remaining rules can be explained similarly. The flow of tokens in a Workflow Graph starts from an *initial state*.

*Definition 4:* The *initial state* of a Workflow Graph is a state, denoted by $s_{init}$, with a single token on the edge going out of the Start node. The set of all reachable states are defined as $Reach(G) = \{s \mid s_{init} \xrightarrow{*} s\}$, where $\xrightarrow{*}$ is the kleene closure of $\{\xrightarrow{*} \mid n \in N\}$. In other words, $s \in Reach(G)$ if there are nodes $n_1, \ldots, n_r$ and states $s_1, \ldots, s_r$ such that $s_{init} \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r$, where $s_r = s$. In this case, we sometimes write $s_{init} \xrightarrow{\sigma} s$ where $\sigma = n_1 \ldots n_r$ is a node on the alphabet $\{n \mid n \in \mathbb{N}\}$ such nodes result in a language: $L(G) = \{\sigma \mid s_{init} \xrightarrow{\sigma} s, \text{ for some } s \in Reach(G)\}$

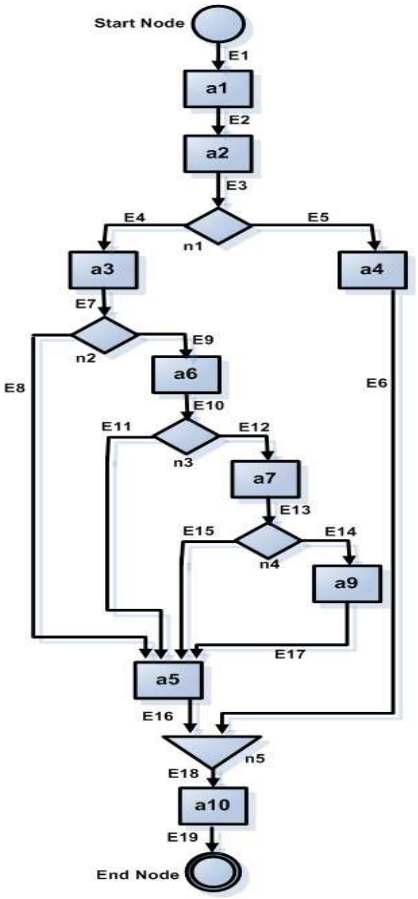


Fig. 3. WorkFlow Graph for the Customer Service

*Example:* The example presented in section II-B can be specified as Workflow Graph as shown in Figure 3, where the workflow's activities are described as follows: a1: receive inputs from client, a2: check customer account, n1: (decision)if-condition to check whether the customer account is ok or not, a3: General Evaluation Service(GES), n2: (decision)if-condition to check the result of GES, a5: advice customer, a6: Line Test Service(LTS), n3: (decision) to check the result of LTS, a7: Check Line Exchange, n4: (decision) to check the

result of exchange result, a9: Right First Time Failure(RFT), n5: (merge) choose one of its incoming edges where at least one token, a10: reply the result to the client.

## III. Observability in Workflow Graphs

In this section we shall extend the model suggested by [18] to include information required for dealing with failure diagnosis. Consider A Workflow Graph model $G = (N, E)$, and a partition of the set of nodes into disjoint subsets of observable $N_{obs}$ and unobservable $N_{uo}$, i.e. $N = N_{obs} \cup N_{uo}$ and $N_{obs} \cap N_{uo} = \phi$. We write $N_{int}$, to represent *internal nodes*, for nodes such as the Start node, Stop node, Fork, Join, Decision and Merge which are executed internally. We consider such actions internal to the system and hence unobservable i.e. $N_{int} \subset N_{uo}$, whereas *activity* nodes which have visible effects such as sending of messages are considered as observable action. Such nodes are for example used to *invoke* another service remotely by sending a message, so have observable effect. Some of the unobservable nodes model occurrence of faults. Since identifying observable faults is trivial, without any loss of generality we shall assume that all failure events are unobservable. Moreover, we assume that occurrence of a faults does not stop the system to a halt. In other words faults are like *caught exceptions* in Java which allow the system to continue gracefully. Further we assume that faults can be classified into $\ell$ disjoint categories, so we write $N_f = N_{f_1} \cup \cdots \cup N_{f_\ell}$ where $N_{f_i}$ denote the nodes representing failure of category $f_i$.

*Example:* In order to explain the above definition, the example presented in section II-B will be used. It has 19 edges $\{e_1, e_2, \ldots, e_{19}\}$, and 22 nodes $N = \{a_1, a_2, n_1, a_3, a_4, n_2, a_5, a_6, n_3, a_7, n_4, a_9, n_5, a_{10}\}$, where $N_{obs} = \{a_4, a_6, a_5, a_9, a_{10}\}$ and $N_{uo} = \{a_1, a_2, n_1, a_3, n_2, n_3, a_7, n_4, n_5\}$, $N_f = \{a_9\}$

From an outsides services point of view only observable event $N_{obs}$ can be recognized. Suppose that the Workflow Graph executes a set of events $\sigma = n_1 \ldots n_r$. To any other service only the observable events are visible.

*Definition 5:* Suppose $P : N \to N_{obs} \cup \{\epsilon\}$ is defined by

$$P(n) = \begin{cases} \epsilon & \text{if } n \in N_{uo} \\ n & otherwise \end{cases}$$

where $\epsilon$ is the identify of the alphabet $N$ ,i.e for $n \in N$, $n\epsilon = \epsilon n = n$. Also assume extending $P : N^* \to (N \cup \{\epsilon\})^*$ by defining for $P(n_1 \ldots n_r) = P(n_1) \ldots P(n_r)$ representing the sequence of observable events in $n_1 \ldots n_r$ (in their right order).

### A. Description of the problem

This paper aims to address the following problem. Suppose $S_1, \ldots, S_n$ are a set of services interacting with each other. The aim is to produce a new service to monitor the behavior of these service to identify the occurrence of the failures.

The Diagnoser Service designed to receives a set of observable events and determines if a failure has happened, or may have happened as shown in Figure 4. This can be formalized as follows:


Fig. 4. An overview of the Diagnosis via a Diagnoser Service

*Definition 6:* considers a Workflow Graph $G = (N, E)$, with $N_{obs}, N_{uo}, N_f = N_{f_1} \bigcup \cdots \bigcup N_{f_\ell}$ of the set of observable, unobservable and failures respectively. Suppose $\sigma = n_1 \ldots n_r \in N^*$ is a is an arbitrary sequence of observable events.

1) we say $\sigma$ ends in a *Normal* state if every sequence of events $\mu_1$ in $N$ which is reachable sequence of $G$ and can be projected to $\sigma$ does not end in a failure event, i. e. $\forall \mu_1 = n'_1 \ldots n'_s \in L(G)(P(\mu_1) = \sigma \Rightarrow \forall i \ n'_i \notin N_f)$
2) we say $\sigma$ *ends in a failure* state of type $F_i$ if every reachable sequence of events $\mu_2$ in $N$, which can be projected to $\sigma$ ends in a failure node of $F_i$ i. e. $\forall \mu = n'_1 \ldots n'_s \in L(G)$ $(P(\mu_2) = \sigma \Rightarrow \forall i < s \ \ n'_i \notin N_f$ and $n'_s \in N_{fi}$
3) we say $\sigma$ *may end in a failure* state of type $F_i$ if there are reachable sequence $\mu_1, \mu_2 \in L(G)$ such that can be both projected to $\sigma$ such that $\mu_1$ ends in a failure node of $N_{fi}$ and $\mu_2$ has no failure nodes, which can be expressed formally similar the cases 1 and 2 above.

To create the Diagnoser Service, in the next section a finite representation that includes language of the Workflow Graph, denoted by $L(G)$ in previous section will be presented.

## IV. Coverability Graph of Workflow

In a Workflow Graph, it is possible to have infinite number of states. This situation happens, when there is possibility of increase of tokens on the edges. Next, we shall present the idea of Coverability Graph for Workflow Graph, or simply Coverability Graph to obtain a finite graph including all permissible reachable traces in Workflow Graph. Coverability Graph in our context is a direct extension of the idea of Coverability Graphs in Petri nets, which have been used by Guia [14], [19] for studying observability.

*Definition 7:* A Coverability Graph of a Workflow Graphs $G = (N, E)$ is a graph $G_{cov} = (N_{cov}, E_{cov})$ where:

I $N_{cov} \subseteq \{0, 1, \omega\}^*$, i.e each edge is marked by a $k$-dimensions vectors of $\{0, 1, \omega\}$, where $k$ is the number of the edges in $G$, $k =\mid E \mid$

II each edge of the Coverability Graph is marked by a node of $G$, i.e. $E_{cov} \subseteq N$ .

III For each reachable set of states $s_{init} \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r$, there is a path $\alpha_0 \xrightarrow{n_1} \alpha_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} \alpha_r$ such that $s_i \leq \alpha_r$ for $1 \leq i \leq r$, where $\leq$ is coordinate ordering of vector in $\mathbb{N} \bigcup \{w\}$ [2].

[2]if $\alpha \in \mathbb{N}^k$ and $\alpha \in \{0, 1, \omega\}$, $s \leq \alpha$ means that for each coordinate $i$, either $s_i = 0$ or $\alpha_i = \omega$ if not $s_i = 1$ and $\alpha_i = 1$. For the reader information with this ordering $\omega$ can be seen $\infty$

Next we shall present an algorithm for constructing the Coverability Graph. This algorithm extends the Coverability Graph of Petri nets by incorporating the semantic of Workflow Graph.

**Algorithm 1**: The computation of WorkFlow coverability Graph

**Result**: compute the Coverability Graph for a Workflow Graph $G$

let $s_{init} = entry\_node$, label the initial node as the root and tag it as "new".;

**while** *"new" Coverability Graph nodes exist* **do**

  Select a new node $\alpha$;

  **if** $\alpha$ *is identical with a node on a path from the root to* $\alpha$ **then**

    tag $\alpha$ "old";

  **else**

    **if** *no nodes are enabled at* $\alpha$ **then**

      tag $(\alpha)$ "dead";

    **else**

      **forall** *events* $n_i$ *enabled at* $\alpha$ **do**

        compute the Marking $\alpha'$ that results from firing $n_i$ at $\alpha$. The firing rules which are described in section II-C must be extended by $\forall n\ n + \omega = \omega + n = \omega$ ; On the path from the root to $\alpha$ if there exists a Coverability Graph node such that $\alpha'' \leq \alpha'$ and $\alpha' \neq \alpha''$ i.e $\alpha''$ is covered by $\alpha'$, then replace $\alpha'(e) = \omega$ for each $e$ such that $\alpha'(e) > \alpha''(e)$;

        Introduce the new $\alpha'$ as node in the Coverability Graph, draw an arc with label $n_i$ from $\alpha$ to $\alpha'$;

        Tag $\alpha'$ "new";

      tag $\alpha$ "old";

*Lemma 1:* Suppose that $G = (N, E)$ is Workflow Graph the algorithm 1 will produce $G_{cov} = (N_{cov}, E_{cov})$ in which $N_{cov} \subseteq \{0, 1, \omega\}^k$

*Proof:* suppose $N_{cov} \nsubseteq \{0, 1, \omega\}^k$ , so there is a node in $N_{cov}$ that is not in $\{0, 1, \omega\}^k$. Because it is possible to have many such nodes, take $\alpha$ to the root is the shortest path. Because $\alpha$ is the shortest, there is a path from the root to $\alpha$. $\alpha_0 \xrightarrow{n_1} \alpha_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} \alpha_r = \alpha$. $\alpha_0$,$\alpha_1$,...,$\alpha_{r-1} \in \{0, 1, \omega\}^k$ but $\alpha_r \notin \{0, 1, \omega\}^k$. Because every execution add only one token $\alpha(j) = 2$. Moreover, because firing of $n_r$ has added one token, $n_r$ can be fork, activity or decision. Suppose $i$ is the input to $n_r$, then $\alpha_r(i) = 0$, as firing of $n_r$ has removed the token from the edge corresponding to $i$, hence $e_r(i) = 0$ , $\alpha_r(j) = 2$. Now, because $\alpha_{r-1} \xrightarrow{n_r} \alpha_r$, we have $\alpha_{r-1}(i) = 1$, $\alpha_{r-1}(j) = 1$, otherwise $n_r$ would has not been enabled under $\alpha_{r-1}$. Since $\alpha_{r-2} \in \{0, 1, \omega\}^k$ and $\alpha_{r-2}(i) \neq \omega$ , $\alpha_{r-2}(j) \neq \omega$. Then, $\alpha_{r-2}(i) \in \{0, 1\}$. Hence, there are four possibility for $\alpha_{r-2}(i)$,$\alpha_{r-2}(j)$

1) $\alpha_{r-2}(i) = 0$ , $\alpha_{r-2}(j) = 1$
2) $\alpha_{r-2}(i) = 1$ , $\alpha_{r-2}(j) = 1$
3) $\alpha_{r-2}(i) = 1$ , $\alpha_{r-2}(j) = 0$
4) $\alpha_{r-2}(i) = 0$ , $\alpha_{r-2}(j) = 0$

Case 1: it is not possible because $\alpha_{r-2} \leq \alpha_{r-1}$ with $\alpha_{r-2}(i) = 0 \leq 1 = \alpha_{r-1}(i)$ which prompts $\alpha_{r-1}(i) = \omega$ from the algorithm 1.

Case 2: Since $n_r$ has a token in its input, it can fire yielding in a new marking $\alpha$ so the $\alpha(j) = 2$. As result there is a path from the root to $\alpha$ in $r - 1$ step which validate minimality of $\alpha_r$.

Case 3: it is not possible as firing of $\alpha_{r-1}$ will not result in $\alpha_{r-1}(i) = 1$.

Case 4: it is not possible as under $\alpha_{r-2}$, $\alpha_{r-1}$ is not enabled to allow $\alpha_{r-2} \xrightarrow{n_{r-1}} \alpha_{r-1}$.

Since all options end in contradiction $N_{cov} \subseteq \{0, 1, \omega\}^k$. ■

*Lemma 2:* For any $G_{cov} = (N_{cov}, E_{cov})$ produced by the algorithm 1 the condition (iii) in definition 7 is satisfied.

*Proof:* Direct extension of the proof of Coverability Graph in [24] with using the semantic of Workflow Graph. ■

*Corollary 1:* Algorithm 1 above produces the Coverability Graph for any given Workflow Graph.

If the Coverability Graph does not have any $\omega$ then every node of it is a reachable state of the Workflow graph. The proof of the following lemma is straightforward and hence omitted.

*Lemma 3:* Suppose that $G$ is a Workflow Graph and $G_{cov}$ is its coverability graph produced by algorithm 1. If none of the nodes in $G_{cov}$ is labeled with a vector containing $\omega$, then $s_{init} \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r$, there is a path of reachable states in $G$, if and only if, there is a path in the Coverability Graph $\alpha_0 \xrightarrow{n_1} \alpha_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} \alpha_r$. As a result, $L(G) = L(G_{cov})$.

## V. WorkFlow Diagnoser Service

A Diagnoser is a graph that approximates the behavior of the system to allow identifying an occurrence of failure. It is used to develop a system (Diagnoser service) that is interacting in an online manner with existing system by receiving a sequence of observable events and providing information about occurrence of failure.

The above definition of the Coverability Graph Diagnoser, firstly draws on the approach suggested by Guia and Seatzu [14], [19] on using Petri nets' Coverability Graphs for diagnosability. Secondly, it follows the method suggested in [16] to include information with regards to the occurrence of failure.

In this section we shall present a diagnoser for a Workflow Graph $G = (N, E)$ based on using Coverability Graph $G_{cov} = (N_{cov}, E_{cov})$. As a result, we shall refer to it as *Diagnoser Coverability Graph.* A Diagnoser Coverability Graph (DCG) is a graph $G_{DCG} = (N_{DCG}, E_{DCG})$ in which edges are marked by only observable events, i.e. $E_{DCG} \subset N_{obs}$. For any observable sequence of events $\sigma = n_1 n_2 \ldots n_r$ in $G$, there is a path in the $G_{DCG}$ starting from the root marked

by $n_1, n_2, \ldots, n_r$. Such path in $G_{DCG}$ ends in a node that provides all the information required for diagnosing failure occurred as a result of firing of any sequence $\mu$ which has $\sigma$ as its observable actions, i.e. $P(\mu) = \sigma$. Next we will explain the nodes of $G_{DCG}$ which include such information which can be used to identify if a failure has happened or may have happened.

Each node of the DCG is marked by a set $\{(\alpha_1, \phi_1), \ldots, (\alpha_r, \phi_r)\}$, where $\alpha_1, \ldots, \alpha_r$ are nodes of the Coverability Graph $G_{cov}$. Each $\phi$ is a vector in $\{0,1\}^\ell$, where $\ell$ is the number of failures categories $F_1, \ldots, F_\ell$. If all coordinates $j$ of vectors $\phi_1, \ldots, \phi_r$ are 0, for any observable path $\sigma$ ending in this node, we can infer that any sequence $\mu$ of actions in $G$ with $P(\mu) = \sigma$ will end up in a state in which no failure of type $F_j$ has occurred. In this case we say, no failure of type $F_j$ has occurred. If none of the coordinates of $\phi_1, \ldots, \phi_r$ is 1, then $\{(\alpha_1, \phi_1), \ldots, (\alpha_r, \phi_r)\}$ represents a normal state, see case 1 of definition 3. Similarly, if all coordinate $j$ of vectors $\phi_1, \ldots, \phi_r$ are 1, then a failure of type $F_j$ has occurred, i.e. case 2 of definition 3. If some of the coordinates $j$ of vectors $\phi_1, \ldots, \phi_r$ are 0 and some are 1, then a failure of type $F_j$ *may* have occurred, case 3 of definition 3.

Suppose that there is a node $\{(\alpha_1, \phi_1), \ldots, (\alpha_r, \phi_r)\}$ of DCG which is connected to another node $\{(\alpha'_1, \phi'_1), \ldots, (\alpha'_q, \phi'_q)\}$ via an edge marked by $n$. Then, there are two nodes $(\alpha_i, \phi_i)$ and $(\alpha'_j, \phi'_j)$, so that $\alpha_i$ is connected to $\alpha'_j$ in the Coverability Graph $G_{cov}$ via an edge marked by $n$. If there is a path in $G_{cov}$ consisting of unobservable events from $\alpha'_j$ to any other node $\alpha$, then $\alpha$ will also appear in $\{\alpha'_1, \ldots, \alpha'_q\}$, i.e. we will have an $(\alpha'_i, \phi'_i)$, with $\alpha'_i = \alpha$. We shall refer to such nodes $\alpha'_i$ as a member of *Unobservable reach* of $\alpha'_j$, as they can be reached from $\alpha'_j$ via unobservable events. Along such path of unobservable events, the vector $\phi_j$ is modified as follows. If an unobservable event in the path is a failure event of type $F_d$, then the $d$-th coordinate of $\phi'_j$ is changed to 1. If the $d$-th coordinate of $\phi'_j$ is already 1 then the coordinate will remain 1. Such changes to the vectors $\phi$ is carried out via a *label propagation function*, described below. Next we shall define the Unobservable reach and label propagation function.

*Definition 8:* Suppose that $G$ is a Workflow Graph and $G_{cov} = (N_{cov}, E_{cov})$ is its Coverability Graph. For each node $\alpha$ of $G_{cov}$, Unobservable Reach of $\alpha$ denoted by $UR(\alpha) = \{\alpha' \mid \alpha \xrightarrow{n_1} \ldots \xrightarrow{n_r} \alpha_r = \alpha', \forall i\ n_i \in N_{uo}\}$.

The label propagation function is used to calculate the vector $\phi$ that must be assigned to each node $\alpha$ of the Diagnoser Coverability Graph.

*Definition 9:* Suppose that $G_{cov} = (N_{cov}, E_{cov})$ is the Coverability Graph of a Workflow Graph $G$. Label Propagation Function is a function $LP : N_{cov} \times \{0,1\}^\ell \times E_{cov} \to \{0,1\}^\ell$. so that $LP(\alpha, \phi, n) = \phi'$ where the $i-$th coordinate of $\phi'$ is defined by

$$\phi'(i) = \begin{cases} 1 & \text{if } n \in F_i \textit{ is an edge of } G_{cov} \textit{ starting at } \alpha \\ \phi(i) & \textit{otherwise} \end{cases}$$

If $\alpha_0 \xrightarrow{n_1} \alpha_1 \ldots \xrightarrow{n_r} \alpha_r$ we will abuse the notation and write $LP(\alpha, \phi, n_1 n_2 \ldots n_r)$ to represent successive application of $LP$ to $n_1, n_2, \ldots, n_r$. Next we shall need a final piece of notation before presenting an algorithm for creating the Diagnoser Coverability Graph.

*Notation:* Suppose that $(\alpha, \phi)$ appears in the labeling of one of the nodes of the DCG. We write $\mathbf{F}(\alpha, \phi)$ to denote the set of all $(\beta, \phi')$ for which there is a sequence of Unobservable events $n_1 n_2 \ldots n_r$ such that $\alpha \xrightarrow{n_1 \ldots n_r} \beta$ and $\phi' = LP(\alpha, \phi, n_1 \ldots n_r)$.

The function $\mathbf{F}$ will be used to calculate the nodes of DCG in the following algorithm.

**Algorithm 2**: Diagnoser coverability Graph

**Result**: compute the Diagnoser Coverability Graph (DCG)

Suppose $s_0 = (entry_node, (0, \ldots, 0))$, where $entry_node$ is the root of the Coverability Graph.;

The first node of DCG is $\{s_0\} \cup \mathbf{s_0}$ and tag it with "new". **while** *Nodes of with tag "new" exist* **do**

- Select a node $S = \{(\alpha_1, \phi_1), \ldots, (\alpha_r, \phi_r)\}$ tagged by "new" ;
- Iterate through the list $(\alpha_i, \phi_i)$ one-by-one;
- **if** *there exists an observable action which is enabled under* $\alpha_i$ *with* $\alpha \xrightarrow{n} \beta$ **then**
  - Let $S' := \{(\beta, LP(\alpha_i, \phi, n))\}$, then write $S' := S' \cup \mathbf{F}(S')$;
- **if** $S'$ *already exists in DCG* **then**
  - discard it;
- **else**
  - create a node marked by $S'$ and tag it as "new";
  - Create and edge from $S$ to $S'$ marked by $n$.;
- Remove the tag "new" from $S$ after finishing the iteration;

Part III of Definition 7 states that each execution trace of $G$ maps into an execution trace of its Coverability Graph. This means the language of G is a subset of the language produced by the Coverability Graph, i.e. $L(G) \subset L(G_{cov})$. However, the converse is not true in general, i.e. it is possible that there are paths in the Coverability Graph starting from the root that don't correspond to any execution path in the Workflow Graph. Such paths may distort the functioning of the diagnoser. However, if the Workflow Graph produces finite behavior the Coverability Workflow Graph will diagnose the system

*Theorem 1:* Suppose that $G$ is a Workflow Graph so that its Coverability Graph $G_{cov}$ does not include any $\omega$. Then the Diagnoser Coverability Graph of Algorithm 2 will diagnose occurrences of the failure.

*Sketch of the proof:* Suppose that $\sigma$ is a sequence of actions observed in the system, i.e. there is an execution path $\mu$ in $G$ so that $P(\mu) = \sigma$. By lemma 3 $L(G) = L(G_{cov})$. So $\mu$ a path in $G_{cov}$ starting at $entry\_node$. By the Algorithm 2 there is a

path in DCG which is marked by $\sigma$ in which failure states of each node is included. For any failure type $F_i$ and any given node $S = \{(\alpha_1, \phi_1), \ldots, (\alpha_r, \phi_r)\}$ if there is no coordinate $i$ marked by 1 in that state no failure of type $Fi$ has happened. If there are some coordinates 1 and some coordinate 0, then a failure of type $Fi$ may have happened. Otherwise, a failure of type $F_i$ must have happened.

*Example:* Applying this algorithm to the example of section II-B, a Diagnoser Service will be generated as shown in Figure 5. Since the running example has 19 edges, the state vector of the Diagnoser Coverability Graph is of length 19. In addition, since it has only one type of failure, the fault vector is of length one. In the first vector, each digit in the row of the state corresponds to the number of tokens in the edge of the workflow , while in the second vector, each digit each row corresponds to a fault type.

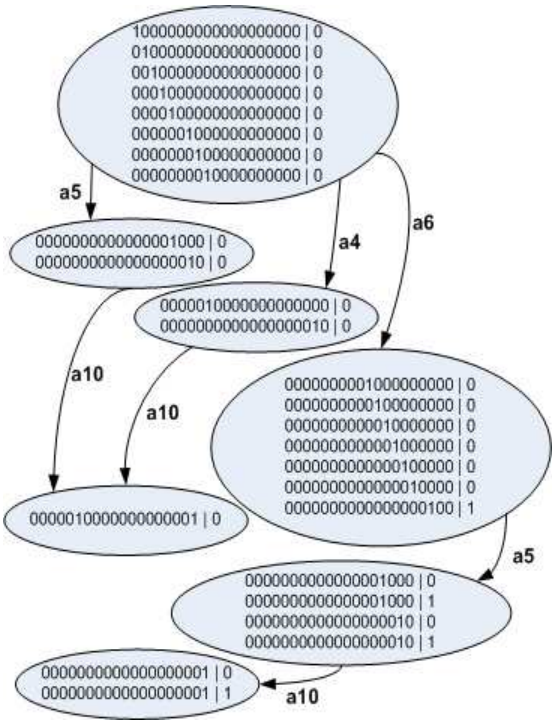


Fig. 5. Diagnoser Coverability Graph Service

## VI. Implementing Workflow Diagnoser Service as BPEL

The presented approach has been implemented as a Plugin for Oracle JDeveloper. Figure 6 is a snapshot of the tool in which path to uploading XSL and BPEL files. The implementation follows the outline of the method as depicted in Figure 7. This method requires passing all BPEL files and their XML Schema Definition (XSD) as inputs. Such information is required to transform BPEL files into their equivalent Workflows. Then, the Algorithm 1, section IV will applied to produce the Coverability Graph. Next, Algorithm 2, section V is applied to generate the Diagnoser Coverability Graph. To create the Diagnoser Service, the Diagnoser Coverability Graph will be firstly implemented and secondly incorporated into existing services so that it can diagnose the occurrence of failure. Next we shall present two methods of implementation of the Diagnoser Coverability Graph and four methods of incorporating it to the exiting services. Further we shall present a comparison between different variations.

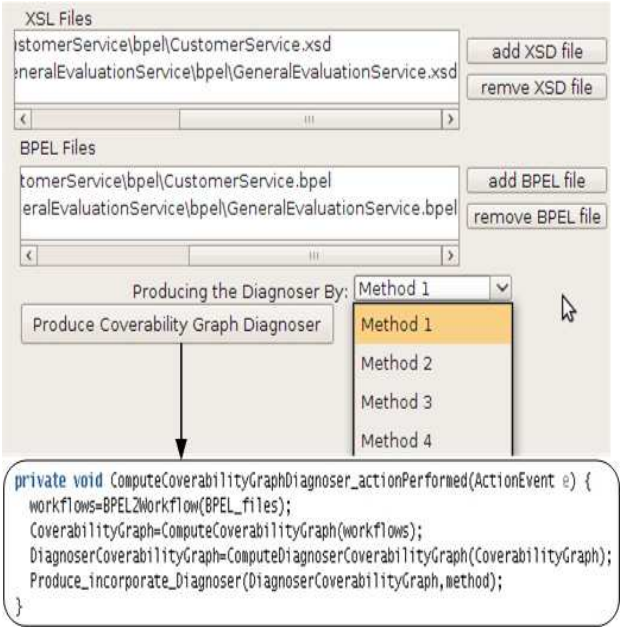


Fig. 6. A snapshot of the implementation as an Oracle JDeveloper plugin

There are various ways to implement the Diagnoser Service. Firstly, Diagnoser Service can be implemented as BPEL file resulting in a service interacting with already existing services in order to diagnose the occurrence of failures. In a nutshell, such BPEL file includes a *Switch* activity involving a number of *Cases* corresponding to the observable events in the Diagnoser model. Each Case is used to evaluate the current status of the services according to the approximation captured in the Diagnoser Coverability Graph (DCG) and returns N for a normal state or the information related to the occurrence of a failure as described in section III. In particular, in case of a failure, the type of failure and the event which is caused the failure will be included in the diagnosing result.

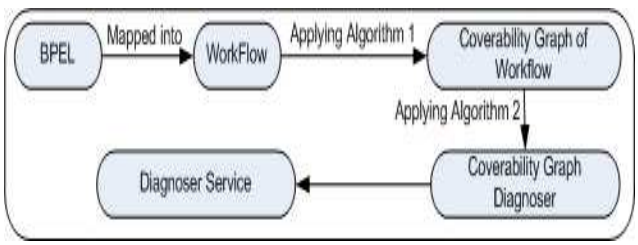


Fig. 7. An outline of the implementation method

It is also possible to implement the Diagnoser Service as

Java Class deployed as Web service that can be invoked by any other services. In this case, similar to the implementation via BPEL, conditional statements in form of *if-then-else* will be used. In both ways, as shown in Figure 4 the Diagnoser Service is incorporated into the system to receive the observable events that has been executed in the service, then it responses with the diagnosing result describing the behavior of the system which is either in normal state or a failure has occurred. Next, we shall discuss various methods of incorporating the Diagnoser Service into the exiting services.

In our Oracle JDeveloper plugin, we have considered four methods can be used to incorporate the Diagnoser Service. These four methods can be explained as following:

**Method 1:** This method is based on implementing the Diagnoser Coverability Graph (DCG) as a BPEL file, which can collaborate with existing services to fulfill the diagnosing task. Each business process should be conducted by including extra *Invoke* activities to execute the Diagnoser Service after each invocation task. Figure 8 represents an example of interaction between the Diagnoser Service and three services, which are *Customer Service*, *GES* and *Line Test Service*. It can be seen that the interaction between services is built as Choreography architecture.

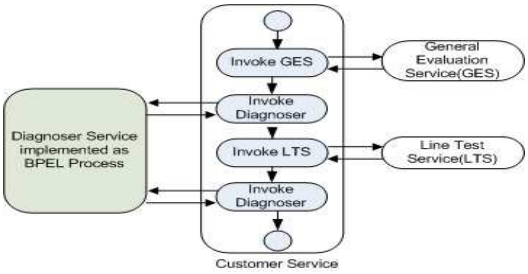


Fig. 8. Example of method 1

**Method 2:** This method produces the Diagnoser Service as BPEL service with a new service called a *Protocol Service*. The protocol service is used as a merge function which is defined to combine the individual Diagnoser Service result and recover the complete diagnosing result that would be obtained after invocation. In contrast to method 1, all the interaction should be performed via the protocol service. For more detail about the protocol service, we refer the reader to our previous approach [7].

**Method 3:** This method automatically produces the Diagnoser Service as a stand-alone Java class deployed as Web service interacting with a group of BPEL services. The generated Diagnoser Service is incorporated in the same manner of Method 1.

**Method 4:** This method based on Method 2 and Method 3, it automatically generates a Diagnoser Service as a Java Class

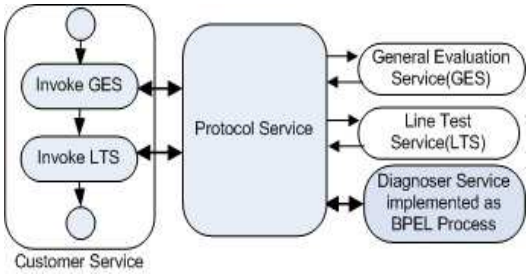


Fig. 9. Example of method 2

deployed as Web service, which interacts with other services through a Protocol Service.

The four presented methods have been tested and evaluated in terms of performance; a common practice of evaluating the performance is applying the *stress testing* which identifies and verifies the stability, capacity and the robustness of services [25], [26]. In order to perform the stress testing, each method has been tested by measuring the time of processing a different number of threads. This test has been carried out with the help of *Oracle Application Server*.

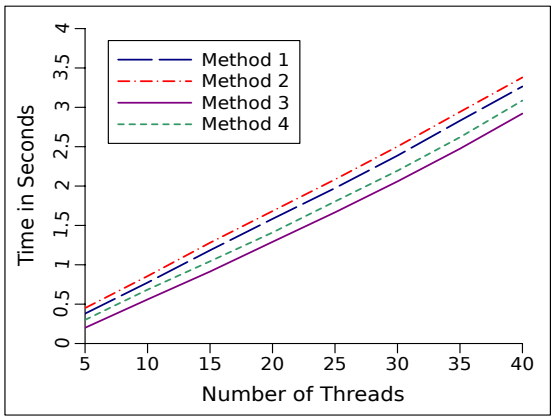


Fig. 10. Stress Testing Result

Figure 10 illustrates the stress testing result of processing different number of threads in seconds. It can be seen that producing the Diagnoser Coverability Graph (DCG) as BPEL file with the Protocol Service in Method 2 is less effective technique. However, producing the Diagnoser Coverability Graph as Java Class deployed as Web service in Method 3 is the fastest way to incorporate the Diagnose Service. Figure 10 shows that the performance of these methods can be seen as linear and parallel. As result, the average of the difference in processing a different number of threads between the fastest method, which is Method 3, and the slowest method, which is Method 2, is approximately 38%.

Producing the Diagnoser Coverability Graph as Java Class deployed as Web service, which is the fastest method in this approach, has been also compared with the fast method of our previous approach which is based in producing the Diagnoser Service by using the Classical diagnosability theory as Java

class Deployed as Web service. The result assets that using the Diagnoser Service based in Coverability Graph are faster than Diagnoser Service based on the Classical diagnosability theory by 13.28% as shown in Figure 11. Therefore, the new proposed approach reduces the cost of the executions time and results in better performance in processing a large number of requests.

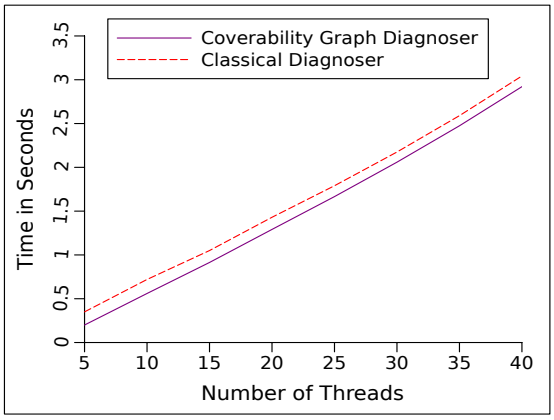


Fig. 11. Classical Diagnoser vs. Diagnoser Coverability Graph

## VII. Discussion

Yan et al. [27], [3] have proposed a method based on computing the business process execution trajectory for monitoring the behavior of BPEL services. The trajectory, which is a path of contiguous states and transitions that begins at an initial state and ends at a final state, can be computed by using the formal model and the observed evolution of the business process. To do so, all BPEL services should be formalized as Discrete Event System (DES). Then, the BPEL engine should record the trajectory of each business process by including information related to the observable events executed by the business process. As a result, if there is any fault occurred during the process, the Diagnoser Service should use the trajectory path record in the Log file to recover from the fault. Our approach differs from [27], [3] in using Coverability Graph and Petri nets diagnosability theories to automatically generate the Diagnoser Service as Workflow Graph which is transformed then to a new service interacting with original existing BPEL representations.

Wang et al. [2] have proposed a method based on discrete control to provide a safe execution in Workflow Graph. The goal of their approach is to ensure that the process reaches a terminal state without entering forbidden states, deadlock and livelock. In that approach, the Workflow Graphs modeled by using a Finite State Machine automaton $G$ which is generated from the original workflow. Then, the control flow of a Workflow Graphs captured by using a Petri net [28]. In order to achieve that goal, there are two phases should be carried out:firstly, an offline control synthesis phase uses the system model $G$ and the specification of the terminal and forbidden state to automatically produce a discrete controller. Secondly, the controllable transitions, which are based on the current execution states, are disabled selectively by the controller during the online dynamic control phase. Disabling transitions is necessary only in order to avoid forbidden states, livelock and deadlock. The proposed solution for that is to build the observer automaton which is also an Finite State Machine (FSM). However, building the Observer has only one complex task in which is that the large number of observer state. Their method can be seen complimentary to ours, they focus on avoiding forbidden state, deadlock and livelock, whereas, in our approach we make use of the diagnosers to identify occurrence of failure, where some might result in arriving in a, for example, forbidden state.

Genc and Lafortune [29] have extended the existing theory of diagnosis of discrete-event systems to satisfy the requirements of diagnosis faults in Petri net. In that approach, the Diagnoser Service is generated as petri net that is associated to every Petri net with the help of a merge function which is defined to combine the individual Diagnoser Service states and recover the complete Diagnoser Service state. Their research results in proposing a distributed fault diagnosis algorithm which allows each module in the distributed system to diagnose its faults independently unless completion of a task requires the use of coupled components.

In our previous approaches [6], [7], a Model Driven Development approach to the design and implementation of Diagnoser Service for a group of interacting services. The method involves transforming BPEL file to Automata and then using DES tools such as [30] to create the diagnoser. This work differs from our previous work substantially. The focus of this research is to develop new algorithms for Workflow Graphs, which are better models for expressing Web services compared to Automata. Comparing the performance of the diagnosers obtained in our previous approach and the new method is an area for further research.

Another area of future research is extending the algorithm to deal with the cases that the coverability graph contains an $\omega$. One candidate for doing so, is to adopt the method developed in [31] for Workflow graphs.

## VIII. Conclusion

This paper presents a method of developing services that can monitor execution of events in a Service oriented Environment and identify occurrence of failure online. The adopted method is model-based; a formal representation of services known as Workflow Graph, which is intended to capture the flow of actions within the system, is used. The conventional Workflow Graph models are extended to model observable, unobservable and failure events. Drawing on existing work on observability in Petri nets, the idea of Petri net Coverability Graph is adopted to produce Diagnoser Coverability Graph (DCG) for Workflow Graphs. The paper also present methods of implementing the DCG as services that can be integrated to a group of services to identify if a failure has happened or may have happened. The presented approach is implemented as an Oracle JDeveloper

plugin, which has been used in a case study involving monitoring of a Customer Service applications to identify Right-first-time failures in telecommunication systems.

## REFERENCES


[1] M. B. Juric, B. Mathew, and P. Sarang, *Business Process Execution Language for Web Services*. Packt Publishing, 2004.

[2] Y. Wang, T. Kelly, and S. Lafortune, "Discrete control for safe execution of it automation workflows," in *EuroSys*, 2007, pp. 305–314.

[3] Y. Yan and P. Dague, "Modeling and diagnosing orchestrated web service processes," in *IEEE International Conference on Web Services*, vol. 9, 2007, pp. 51 – 59.

[4] H. Lababidi, "Model-based diagnostic expert system for chemical plants," 1992, pp. 4/1–4/3.

[5] W. Hamscher, L. Console, and J. de Kleer, Eds., *Readings in model-based diagnosis*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[6] M. Alodib, B. Bordbar, and B. Majeed, "A model driven approach to the design and implementing of fault tolerant service oriented architectures," in *3rd International Conference on Digital Information Management (ICDIM)*, 2008.

[7] M. Alodib and B. Bordbar, "A model driven architecture approach to fault tolerance in service oriented architectures, a performance study," in *3rd International Workshop on Modeling, Design, and Analysis for Service-oriented Architectures (MDA4SOA)*, 2008.

[8] Y. Yan, Y. Pencole, M.-O. Cordier, and A. Grastien, "Monitoring web service networks in a model-based approach," in *ECOWS05*, Sweden, 2005.

[9] R. K. Boel and J. H. van Schuppen, "Decentralized failure diagnosis for discrete-event systems with costly communication between diagnosers," in *WODES '02: Proceedings of the Sixth International Workshop on Discrete Event Systems (WODES'02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 175.

[10] R. Debouk, S. Lafortune, and D. Teneketzis, "Coordinated decentralized protocols for failure diagnosisof discrete event systems," *Discrete Event Dynamic Systems*, vol. 10, no. 1-2, pp. 33–86, 2000.

[11] S. Jiang and R. Kumar, "Failure diagnosis of discrete-event systems with linear-time temporal logic specifications," *IEEE TRANSACTIONS ON AUTOMATIC CONTROL*, vol. 49, no. 6, pp. 934–945, 2004.

[12] J. Lunze and J. Schroder, "Sensor and actuator fault diagnosis of systems with discrete inputs and outputs," *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICSPART B*, vol. 34, no. 2, pp. 1096 – 1107, 2004.

[13] M. Sampath, S. Lafortune, and D. Teneketzis, "Active diagnosis of discrete event systems," in *PROCEEDINGS OF THE 36TH IEEE CONFERENCE ON DECISION AND CONTROL, VOLS 1-5*, ser. IEEE CONFERENCE ON DECISION AND CONTROL - PROCEEDINGS, 1997, pp. 2976–2983, 36th IEEE Conference on Decision and Control, SAN DIEGO, CA, DEC 10-12, 1997.

[14] A. Giua and C. Seatzu, "Observability of place/transition nets," *IEEE Transactions on Automatic Control*, vol. 47, no. 9, pp. 1424–1437, 2002.

[15] M. Sampath, R. Sengupta, and S. Lafortune, "Diagnosability of discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 40, pp. 1555–75, Sept. 1995.

[16] S. Genc and S. Lafortune, "Distributed diagnosis of discrete-event systems using petri nets," in *International Conf. of Application and Theory of Petri Nets*, ser. IEEE CONFERENCE ON DECISION AND CONTROL - PROCEEDINGS, 2003, pp. 23–27.

[17] G. Jiroveanu, R. B. and, and B. Bordbar, "On-line monitoring of large petri net models under partial observation," *Discrete Event Dynamic Systems*, 2008.

[18] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through sese decomposition," in *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*. Springer-Verlag, 2007, pp. 43–55.

[19] A. Giua and C. Seatzu, "Fault detection for discrete event systems using petri nets with unobservable transitions," in *44th IEEE Conference on Decision and Control*, 2005, pp. 6323– 6328.

[20] A. Arsanjani, "Empowering the business analyst for on demand computing," *IBM Systems Journal*, vol. 44, no. 1, pp. 67–80, 2005.

[21] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services*. Springer Berlin, 2004.

[22] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "Xml schema part 1: Structures," 2004.

[23] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[24] W. Reisig, *Petri nets: an introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1985.

[25] M. B. Juric, B. Mathew, and P. Sarang, *Business Process Execution Language for Web Services*. Packt Publishing, 2004.

[26] U. o. M. Department: Software Verification, "Stress test strategy."

[27] Y. Yan, Y. Pencole, M.-O. Cordier, and A. Grastien, "Monitoring web service networks in a model-based approach," in *ECOWS05*, Sweden, 2005.

[28] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[29] S. Genc and S. Lafortune, "Distributed diagnosis of discrete-event systems using petri nets." in *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003),*. Springer-Verlag, 2003, pp. 316–336.

[30] L. Ricker, S. Lafortune, and S. Genc, "Desuma: A tool integrating giddes and umdes," in *WODES*, 2006.

[31] G. Jiroveanu, R. Boel, and B. Bordbar, "On-line monitoring of large petri net models under partial obervation," *Submitted to Journal of Discrete Event Dynamic Systems*, 2006.