A modelling Approach to Service oriented Architecture for On-line Diagnosis

Mohammed Alodib and Behzad Bordbar, *Member, IEEE*

Abstract—Failure diagnosis is one of the key challenges of Service oriented Architectures. One of the methods of identifying occurrences of failure is to use Diagnosers; software modules or services which are deployed with the system to monitor the interaction between the services for identifying if a failure has happened or may have happened. This paper aims to present a suitable modelling framework to allow automated creation of Diagnosers based on Discrete Event System (DES) theory. Coming up with an appropriate modelling language framework is a prerequisite to applying DES techniques. Modelling languages popular in DES, such as Petri nets and Automata, despite being sufficiently adequate for modelling, are not well adopted by the SoA community. Inspired by Petri nets and Workflow Graph, the modelling suggested in this paper closely follows BPEL which widely used by the community. In particular, our language includes constructs which are supported by major tool vendors. To demonstrate that the suggested formal language is a suitable basis for the application of DES theory, we have extended one of existing DES methods for the creation of centralised Diagnoser. Two algorithms for creating Diagnosers are put forward. These algorithms are applied into the models which are abstracted from the BPEL representation of the involving services. As a proof of concept, an implementation of the suggested approach is created as an Oracle JDeveloper plugin that automatically produces new Diagnosing services and integrates them to work with existing services. The paper ends with a series of empirical results on the performance-related aspects of the proposed method.

Index Terms—Service oriented Architecture, Failure, diagnosis, Workflow Graph, BPEL.

1 INTRODUCTION

SERVICE oriented Architecture (SoA) provides a layered architecture for organising software resources as services, so that they can be deployed, discovered and combined to produce new services [1]. SoAs can be involved in performing crucial functions that the users are heavily dependent upon. As a result, it is urgent to identify occurrences of failure, so that suitable remedial actions can be adopted. In order to identify the occurrence of failures a common method is to create *Diagnosers* i.e. 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]. This paper deals with methods of automated creation of Diagnosers for SoAs, which draw on Discrete Event System (DES) theory [9], [10], [11], [12], [13].

To capture business process models, we adopt Workflow Graphs as suggested by Vanhatalo et al. [14]. This modelling language is suitable for our purpose for the following reasons. Firstly, Workflow Graph is a rich modelling language. It includes necessary constructs, such as Fork, Join, Merge and Decision, which are commonly used in the modelling of business processes. Secondly, Workflow Graphs have strong semantics based on Petri nets which allows a formal approaches to formulating failure diagnosis. Thirdly, to the best of our knowledge, the formalism of [14] is the closest to the standards such as Business Process Execution Language (BPEL) which are widely adopted by the industry and tool vendors. An alternative approach for creating Diagnosers would be to directly use conventional Petri nets [15], [16], and develop diagnosis for Petri net [10], [11]. This would require the developers to adopt Petri nets as their modelling language. Petri nets, despite being sufficiently adequate, are not well adopted. Currently, developers and commercial SoA tool vendors prefer BPEL [17] and BPMN [18]. Another alternative would be to automatically transform BPEL to Petri nets or Automata. In our earlier work [5], [6], [7], we made use of Automata which meant that we relied on the transformation of business process models to automata. Using Workflow Graphs eliminates any need to transform a business process, for example in BPEL, into a secondary language such as Automata or Petri net, which may require bridging the gap between modelling languages by proving the correctness of the transformation. Indeed all exiting methods for using Automata and Petri net are valid only if the correctness of the transformation is proved. Proving the correctness of model transformations, which is similar to proving correctness of compilers, is a formidable task.

This paper makes three main contributions. Firstly, it extends the Workflow language suggested in [14] by providing support for *While* loop as specified in the BPEL standard. Indeed, Vanhatalo's Workflow Graphs of [14] can express repetitive behaviour by creating loops. However, creating loops by making cycles in the Workflow Graph may lead to *unstructured cycles* which have more than one entry or exit points [19], [20]. Tools such as IBM WebSphere and Oracle JDeveloper disallow creation of such unstructured loops. Instead, they introduce high-level constructs in form of "While loops" [21], [22], [1]. The formalism presented by Vanhatalo et al. [14], although closely follows the standard, does not support such While loop constructs. Our extension of Workflow Graph called *Extended Workflow Graph (EWFG)* results in a tree of Workflow Graphs where a While activity

• *M. Alodib and B. Bordbar is with the School of Computer Science, University of Birmingham, Birmingham, B15 2TT, UK.*
E-mail: m.i.alodib@cs.bham.ac.uk, b.bordbar@cs.bham.ac.uk

is a node with a (unique) child, which is also an Extended Workflow Graph, representing the behaviour that should be repeated when the While occurs. We also present a formalism for supporting BPEL *Invocation*, which are widely used in business process. The formalism is used to prove that the language produced by EWFG is a Regular language. In other words, the language underlying models supported by these tools,BPEL, is a regular language. Secondly, two algorithms are presented for the automated generation of Diagnosers. This shows that our language is a suitable basis for developing Diagnosers via DES techniques. The third contribution is a report on an implementation of the method as an Oracle JDeveloper Plugin, as a proof of concept. In SoA, due to the distributed nature of services, it is crucial to present suitable infrastructure and code for integrating the created Diagnosers to the exiting services. There are various options for integration of the Diagnoser, we have presented four in this paper. The created tool has been used for the evaluation of the performance of different methods of integrating the created Diagnoser to the existing services.

The paper is organised as follows. Section 2.1 presents a brief review of the business process modelling. Section 2.2 reviews the Workflow Graph model [14]. In Section 2.3 a running example based on a scenario of failure detection in an e-shopping system is described. Unstructured loops are explained in Section 3. Extended Workflow Graphs and their semantic are introduced in Section 4. Automated creation of Diagnosers involves two steps. Firstly, in Section 5, an algorithm for creating a graph called Coverability Graphs, which extends the idea of Petri net Coverability Graph [13], is presented. From this point of view our approach furthers the method suggested by Giua and Seatzu [13], [23] and Genc and Lafortune [10] in designing Diagnoser for Petri net models. In Section 6, a second algorithm is put forward which produces the Diagnoser. Section 7 presents an outline of the implementation method which automatically produces a Diagnoser service based on the two algorithms, and a sketch of implementation as an Oracle JDeveloper's plugin. The Section ends by proposing and comparing four styles of interaction between the Diagnoser service and existing systems.

2 PRELIMINARIES

2.1 Business Process Execution Language (BPEL)

There is an ever-increasing pressure on modern enterprises to adapt to the changes in their environment by evolving to respond to any opportunities or threats [24]. Service oriented Architecture (SoA) provides a foundation for implementing business processes via the composition of existing services. Web services are software systems which make use of well-accepted standards to support the creation of SoAs. The interaction between services is facilitated via Web service Description Language (WSDL). WSDL is an XML language aimed at defining message formats, data-types and transport protocols [25]. Web services interaction can be either orchestration or choreography [26], [27]. Orchestration relies on a central Service coordinating and controlling the invocation between a set of Web services. In contrast, the choreography structure does not rely on any central entity [26], [27].

Interaction between services can be represented with the help of Business Process Execution Language (BPEL) [1]. BPEL is used to express complex behaviour such as sequential, parallel, iterative and conditional interactions. BPEL also has model elements for specifying Reply, Receive, Invoke and Terminate [1], [28] which are used in business process. For general information on Web service we refer the reader to [1].

2.2 Workflow Graph

Workflow models are now widely used for specifying business processes [15], [16], [29], [30]. Focusing on the analysis of the systems, Van der Aalst et al. [15], [16] present a Workflow modelling language, in which models are constructed from blocks of Petri net models representing common workflow constructs. The blocks of Petri nets are assembled together to create new models of the overall business process. These new models are used in conducting analysis to identify, among other things, existence of deadlocks. Petri nets, despite being a power representation, are not adapted by the SoA community, which is more keen of modelling via BPEL [17] and BPMN [18]. Such languages are also supported by tools vendors. We find the modelling language suggested by Vanhatalo et al. [14], which is also based on Petri net, the closest to the style adopted by the major tools such as IBM WebSphere and Oracle JDeveloper. As a result, our work builds on the formalism suggested in [14]. For the rest of this section, we will briefly review [14].

Definition 1: [14] A Workflow Graph is a graph $G = (N, E)$ where N is the set of nodes and E is the set of edges. Each node $n \in N$ represents an action such as Start, Stop, Activity, Fork, Join, Decision and Merge. Each edge of the Workflow Graph connects two nodes to each other i.e. $E \subseteq N \times N$.

Notation 1: Every Workflow Graph has a unique Start and Stop node. For a Workflow Graph G we shall denote them with *G.Start* and *G.Stop*.

Fig. 2 is a graphical depiction of a Workflow Graph. Each Workflow Graph has a single *Start* and *Stop*. These are denoted by two circles in Fig. 2. Start Node has no input edge, whereas Stop node has a single output edge.

Notation 2: For each node n, the set of Input/Output edges of n are denoted by $In(n)$ and $Out(n)$.

Fork and Decision nodes have only one input edge, while Join and Merge have only one output edge. Fork and Decision edges are generally expected to have more than one output edges, whereas Join and Merge nodes have more than one input edges.

Semantic of Workflow Graph According to Vanhatalo et al. [14] a *state* of a Workflow Graph is represented by assignment of tokens to the Workflow Graph edges. An edge with a token indicates that the action following the edge can be potentially executed. This is very similar to assignment of tokens into the places in Petri nets [31]. Also similar to the Petri net execution of an event in a Workflow Graph results in the movement of the tokens between the edges to capture the flow of actions.

Notation 3: Suppose that $G = (N, E)$ is a Workflow Graph. A *state* in G is a mapping $s : E \to \mathbb{N}$ which assigns a number of tokens to each edge $e \in E$ i.e. $s(e) = k$ means edge e carries out k tokens, where $\mathbb{N} = \{0, 1, 2, \ldots, \}$. An execution of *node* $n \in N$ results in changing the state from s to s' denoted by $s \xrightarrow{n} s'$.

Definition 2: (Semantics of change of States) [14] Assume that s, s' are two states of a Workflow Graph and $s \xrightarrow{n} s'$.

- if n is a Start node then:

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

- if n Stop node then:

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

- if n is an Activity, Fork or Join:

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

- if n is a Decision and e' is one of the output edges of n:

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

  In other words, Vanhatalo et al. [14] assume that an execution of a Decision node n results in removing tokens from an input edges of n and depositing it in one of the output edges of n non-deterministically.
- if n is a Merge and e' is one of the incoming edges of n with $s(e') > 0$:

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

In the next section we will introduce our running example.

2.3 Example: E-Shopping System

This section gives an outline of a running example which is used in the rest of the paper. To describe our approach, we will make use of a modified version of the example given by Guillou et al. [32], [33]. This example is based on a typical on-line e-shopping system consisting of three main services: Shop, Supplier and Warehouse.

As depicted in Fig. 1, the customer accesses the Shop Web site to search for items. Then, he adds his items to the Shopping Cart which is then passed to the Supplier service by the Shop Service. For each item in the list, the Supplier service sends a request to the Warehouse to check if the item is available. If the item is available the Warehouse service sends an acknowledgement to the Supplier to complete processing the order. Next, the Supplier Service send back the list of available items to the Shop service. Finally, the list is forwarded to the customer who then confirms his order.

Fig. 2 depicts the Workflow Graph for the Shop service. It can be seen that the Shop service receives the placed order which is stored in its database. Shop will send the list of items

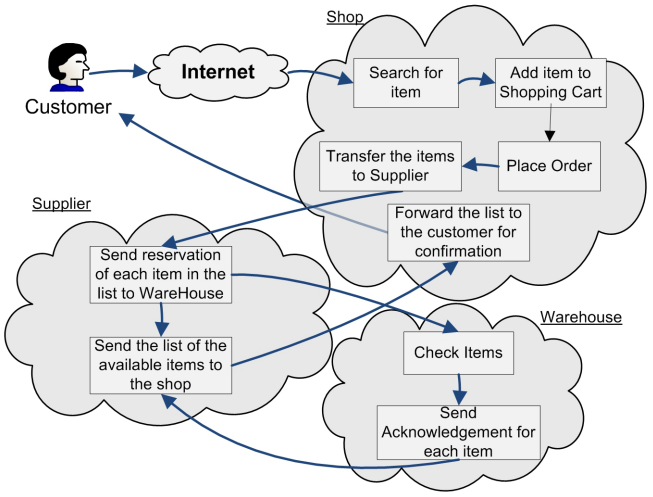

Fig. 1. E-shopping Scenario

to the Supplier to check their availability. After receiving the list of available of items from the Supplier, the payment items is calculated and a summary of the order is be sent to the customer. Then, the customer has the option to either confirm or cancel his order.

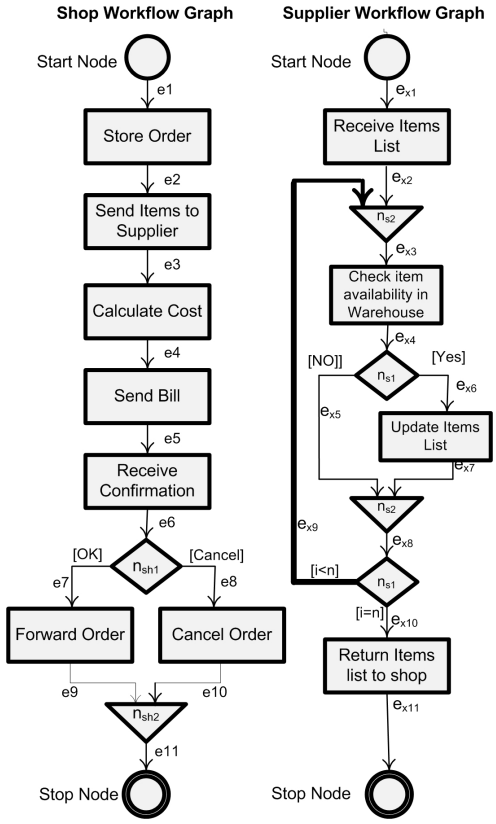

Fig. 2. Workflow Graph for the Shop and Supplier Services

Fig. 2 also depicts the behaviour of the Supplier service which receives the list of items sent by the Shop service. Then, a request for each item in the list is sent to the Warehouse

service. If the item is available at Warehouse, the list of items is updated. After iterating through the processed items, the final updated list is returned back to the Shop Service.

Fig. 3 depicts the Workflow Graph for the Warehouse service which receives a request sent by the Supplier for an item. If the item is available, it is reserved and subsequently the list of item in the stock is updated. If not, the Supplier is informed that the item is not available.

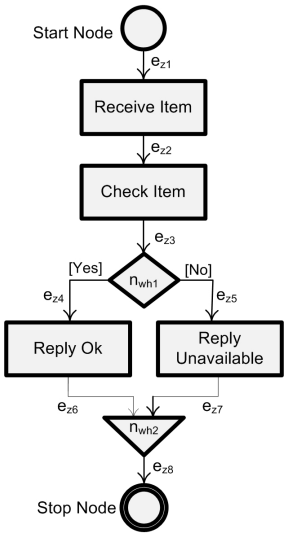

Fig. 3. Workflow Graph of the Warehouse Service

3 UNSTRUCTURED LOOP AND BPEL

In the Workflow Graph of Vanhatalo et al. [14], a repetitive behaviour can be represented by a cycle in the graph created by combining a Merge and Decision to link a node to an earlier node on the path from the Start node. For example, the iterative scenario to check the availability of each item in the list of customer's order can be captured as depicted in Fig. 2. This style of representing the repetitive behaviour, which is known as *unstructured loop*, is very similar to using *goto* command and allows loop with multiple entry and exit.

In programming languages the use of *goto* command is discouraged [19], [20]. Creation of Workflow Graph models using unstructured loop may result in an elimination of the parallel behaviour producing inefficient code [34]. Eliminating unstructured loops does not reduce the expressive power of the Workflow Graph. There are various algorithms available which allow elimination of the unstructured loops and replacing them involving equivalent loops with a single input and output node [35], [34].

Leading tool vendors do not support creation of unstructured loop in a model. For example, it is not possible to produce a Workflow Graph similar to Fig. 2 in Oracle JDeveloper. Instead such tools adopt a hierarchical style similar to While loop in conventional programming languages. A While loop is designed to repeat a set of tasks as long as the loop *condition* is valid. It also worth mentioning that BPEL does not support For-Loops or Do-While Loops because they would prevent the process from being exported [21], [22]. As a result, a repetitive behaviour such as the Supplier behaviour captured in Fig. 2 must be converted to a While node as depicted in Fig. 4 which includes internal repetitive behaviour as a separate Workflow graph ($B : While\ Block$).

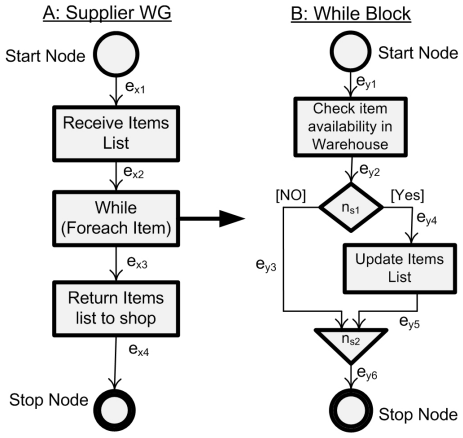

Fig. 4. Workflow Graph of the Supplier Service with While

4 EXTENDED WORKFLOW GRAPH MODEL

In this section, we shall extend the formalism presented by Vanhatalo et al. [14] to support While loop of BPEL. We also model Invocation nodes which are used to perform remote invocation of another Services [1]. An Invocation activity in BPEL can be either a *one-way operation* (i.e. it has only an input message and does not expect a result to be returned from the remote service) or a *two-way operation* which has an input message and returns a *result* synchronously [1].

Extension of Vanhatalo et al. [14] is achieved in two steps. Firstly, in definition 3 we shall present an enhancement of Workflow Graph of [14], which we call a *Workflow Graph with Invocation and While nodes (WFGIW)*. A WFGIW is an extension of Definition 1 to include Invocation and While nodes. For example, the Workflow Graph at the left-hand-side of Fig. 4 is WFGIW. Any conventional Workflow Graph, as defined in [14], is also a WFGIW. Consequently, WFGIW extends the conventional Workflow Graph.

WFGIW is used at the nodes of the tree that represents, for example, the relation between a While loop and its repetitive behaviour. Secondly, in definition 4 the tree representation is enhanced to introduce the notation of *Extended Workflow Graph*.

Definition 3: A Workflow Graph with Invocation and While nodes (WFGIW) is a graph $G = (N, E)$ where N is a set of nodes representing one of the following: Start node, Stop node, Activity, Fork, Join, Decision, Merge, Invocation and While. The set of all Invocation nodes of G is denoted by $\mathscr{I}(G)$. The set of all While nodes of G is denoted by $\mathscr{W}(G)$. E is the set of edges $E \subseteq N \times N$ where each edge $e \in E$ connects two nodes with each other. Invocation and While nodes have a single input and single output edge which are not identical where: $\forall n \in \mathscr{I}(G) \bigcup \mathscr{W}(G)\ |In(n)| = |Out(n)| = 1$ and $In(n) \bigcap Out(n) = \phi$.

Fig. 4 depicts the Supplier WFGIW in (A: Supplier WG). The internal behaviour associated to the execution of the While node is depicted as a second WFGIW (B: While Block). Such an internal behaviour for the While node can be executed as long as the condition attached to the While node is true[1]. Similarly, there is an internal behaviour which itself can be a WFGIW for each Invocation node. As a result, our proposed models, called Extended Workflow Graph, are in form of tree with a WFGIW on each node of the tree.

Definition 4: An Extended Workflow Graph (EWFG) is a tree of form $T = (\mathscr{V}, \Sigma)$ where $\mathscr{V} = \{G_1, \ldots, G_n\}$ is the set of Workflow Graphs with Invocation and While nodes (WFGIW). Each edge of the tree maps two Workflow Graphs together $e = (n, G_j) \in \beta$ where β is a function that maps Invocation or While nodes to their corresponding internal behaviour i.e.

$$\beta : \bigcup_{i=1}^{n} \mathscr{I}(G_i) \cup \mathscr{W}(G_i) \longrightarrow \mathscr{V}$$

For each $G_i \in \mathscr{V}$, except the root node, $\beta^{-1}(G_i)$ is the unique Invocation or While node associated to G_i.

The following properties can be inferred from the above definition:

- There are no Invocations or While nodes in the final Workflow Graph of an EWFG because T is a finite tree.
- If there are more than one Invocation or While nodes in a Workflow Graph, there are more than one edges out of that Workflow Graph. To be precise, for each node G_i the number of Invocation and While nodes is exactly the same as the number of edges starting at G_i. This ensures assignment of a unique WFGIW to each Invocation or While node as their internal behaviour.
- Invocation and While nodes can not call a WFGIW which is their father node, because T is a tree.

Clearly, Workflow Graphs are special cases of Workflow Graphs with Invocation and While nodes (WFGIW). Sometimes, when there is no chance of ambiguity, we shall abuse the notation and refer to a WFGIW as simply a "Workflow Graph". However, we will always use the phrase Extended Workflow Graph for the tree which has WFGIWs as it nodes.

4.1 States of Extended Workflow Graph

Vanhatalo et al. [14] define the state of a Workflow Graph G as a function $s : E \longrightarrow \mathbb{N}$, where E is the set of edges in G. This definition can be extended for Extended Workflow Graph which has multiple set of edges $E_1, \ldots, E_n$. Let us write $E = E_1 \cup \cdots \cup E_n$ and define a function $s^E : E \longrightarrow \mathbb{N}$ for capturing the number of tokens on each of the edges of an Extended Workflow Graph $T = (\mathscr{V}, \Sigma)$ where $\mathscr{V} = \{G_1, \ldots, G_n\}$ and $E_1, \ldots, E_n$ are edges of $G_1, \ldots, G_n$. However, this will not be adequate for capturing the states of an Extended Workflow Graph. We shall explain this with the help of the Workflow Graph of Fig. 4. If one of the edges in block B is marked with a token, we can infer that the internal Workflow Graph related to the While loop is executing. We need a mechanism to capture this information in the Workflow Graph represented in block A. To achieve this, we shall assign a token to each While node, which its internal Workflow Graph is executing. Similarly, we shall assign a token to each Invocation node, which its internal Workflow Graph is executing. As a result, state of Extended Workflow Graph consists of a trio of functions (s^E, s^I, s^w) where s^E is the extend state as explained above, s^I and s^w are functions to keep the number of tokens in Invocation and While nodes respectively. This can be formalised as follows.

Definition 5: Suppose $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph EWFG where $\mathscr{V} = \{G_1, \ldots, G_n\}$. For each i, E_i, $\mathscr{I}(G_i)$ and $\mathscr{W}(G_i)$ represent the set of edges, Invocation and While nodes of G_i. Each state s of T is a trio of functions $s = (s^E, s^I, s^w)$ so that:

- $s^E : E_1 \cup \cdots \cup E_n \longrightarrow \mathbb{N}$ where $E = E_1 \cup \cdots \cup E_n$ and $s^E(e) = n$ means that there are n tokens on the edge e
- $s^I : \bigcup_{i=1}^{n} \mathscr{I}(G_i) \longrightarrow \{0, 1\}$, where $s^I(n) = 1$ if and only if the Workflow Graph representing the internal behaviour of the Invocation node n is executing.
- $s^w : \bigcup_{i=1}^{n} \mathscr{W}(G_i) \longrightarrow \{0, 1\}$, where $s^w(n) = 1$ if and only if the Workflow Graph representing the internal behaviour of the While node n is executing.

Notation 4: We will write each trio of functions $s = (s^E, s^I, s^w)$ as the concatenation of the three parts involving coordinates which are mapped to edges $E_1 \cup \cdots \cup E_n$, Invocation nodes $\bigcup_{i=1}^{n} \mathscr{I}(G_i)$ and While nodes $\bigcup_{i=1}^{n} \mathscr{W}(G_i)$.

We shall use the example of e-shopping System to illustrate the state of the Extended Workflow Graph with the help of initial state. Fig. 5 depicts the initial state of the system which is considered with having only one token in the output edge of the Start node of the Shop service.

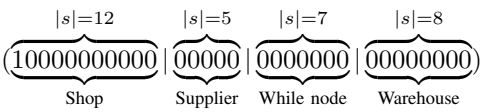

Fig. 5. Initial State of e-shopping system

4.2 Semantics of Extended Workflow Graph

Suppose that s_i and s_{i+1} are two states of an Extended Workflow Graph. We write $s_i \xrightarrow{n} s_{i+1}$ to denote firing a node n alters the state of s_i to s_{i+1}. In Section 2.2 the firing rules of Merge, Activity, Decision, Fork, and Join have been defined. Also for Start and Stop nodes of the root Workflow Graph the firing rules have been explained. The same set of firing rules can be applied to these nodes in EWFG. When these nodes fire, obviously the value of coordinates corresponding to internal actions of Invocations and While nodes remain unchanged i.e. $s^I_{i+1} = s^I_i$ and $s^w_{i+1} = s^w_i$.

1. Modelling of such condition requires modelling of data. The focus of this paper is on diagnosing of failure related to the flow of actions. Modelling of data remains a topic for future research. In the absence of such a condition, we will assume that an internal behaviour can occur any arbitrary number of times.

In the rest of this section we shall discuss the firing rules of Invocation and While nodes. In such nodes, the tokens must move into the children node to execute the internal behaviour of the parent node. This is achieved by firing of the Start nodes of the children node. Similarly, when the execution of the internal Workflow Graph, i.e. the children node, is terminated, the token must be removed. For example, in case of a While node, either the internal behaviour repeats itself, which means a new execution of the Start node, or the execution of the While is terminated and it results in a removal of token from the child Workflow. Consequently, as a part of describing the firing rules for the execution of Invocation and While nodes, we shall describe the execution for their Start and Stop node of the children nodes, which is different from the execution rules for Start and Stop node of the root of the EWFG.

Definition 6: (Change of States for Invocation and While) Assume that $s_i = (s_i^E, s_i^I, s_i^w)$ and $s_{i+1} = (s_{i+1}^E, s_{i+1}^I, s_{i+1}^w)$ with $s_i \xrightarrow{n} s_{i+1}$. Suppose n **is a While node** such that $(n, G) \in \beta$ i.e. G is the internal behaviour associated to the node n. The While node n is enabled when it has one token in its input edge as shown in Fig. 6a. Firing n removes one token from its input edge and add one token to the Start Node of its associated Workflow Graph $\beta(n).Start$; the While node n is marked by "1" token to indicate that the Workflow Graph associated to n is executing i.e. $s_i^w(n) = 1$ as shown in Fig. 6b.

- n is enabled if $s_i(e) > 0$ for $e \in In(n)$ then
  $$s_{i+1}^E(e) = \begin{cases} s_i^E(e) - 1 & \text{if } e \in In(n) \\ s_i^E(e) + 1 & \text{if } e \in Out(\beta(n).Start) \\ s_i^E & \text{otherwise.} \end{cases}$$
  For the coordinates in the Invocation part $s_{i+1}^I = s_i^I$. For the coordinates in the While part:
  $$s_{i+1}^w(m) = \begin{cases} 1 & \text{if } m = n \\ s_i^w & \text{if } m \neq n. \end{cases}$$
- Firing rule for the Stop node of a child node of a While node as shown in Fig. 6c: if m is $\beta(n).stop$ where n is an While node, m is enabled if $s_i(e) > 0$ for $e \in In(m)$. If While should be repeated, firing m removes one token from its input edge and adds one token to the Start Node i.e. $\beta(n).Start$. Otherwise, firing of m removes one token from its input edge and adds one token to the output edge of While node n; the token held on the While node n node is removed and the coordinate corresponding to n in the vector of state is marked by "0" to indicate that the process on the Workflow Graph associated to n has been completed as shown in Fig. 6d.
  $$s_{i+1}^E = \begin{cases} s_i^E(e) + 1 & \text{Repeat } \& \ e \in Out(\beta(n).Start) \\ s_i^E(e) + 1 & \text{No Repeat } \& \ e \in Out(n) \\ s_i^E(e) - 1 & \text{if } e \in In(m) \\ s_i^E & \text{otherwise} \end{cases}$$
  For Invocation part $s_{i+1}^I = s_i^I$.
  For While part:
  $$s_{i+1}^w(m) = \begin{cases} 0 & \text{if } m = n \\ s_i^w & \text{if } m \neq n \end{cases}$$

Suppose n is an Invocation node such that $(n, G) \in \beta$ where G represents the internal behaviour that should be performed as n executes.

If n **is an Invocation node with a *one-way operation***, the firing of n removes one token from its input edge and adds one token to its output edge and one token to the Start Node of its associated Workflow Graph $\beta(n).Start$. This means that the parent Workflow Graph continues executing, and at the same time, child Workflow Graph also start executing.

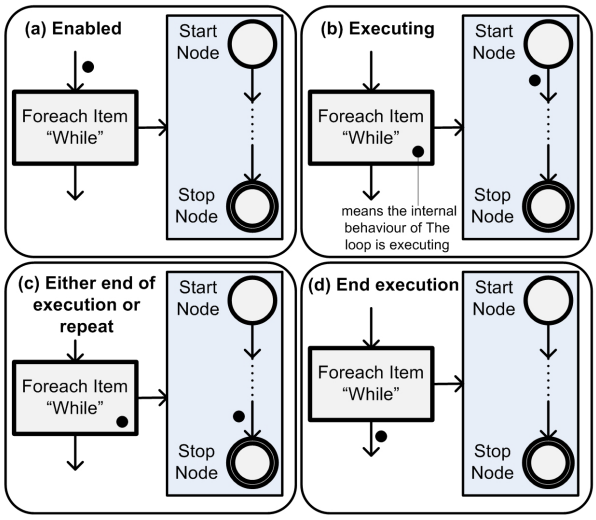

Fig. 6. While loop Structure and Semantics

- n is enabled if $s_i(e) > 0$ for $e \in In(n)$ then
  $$s_{i+1}^E(e) = \begin{cases} s_i^E(e) - 1 & \text{if } e \in In(n) \\ s_i^E(e) + 1 & \text{if } e \in Out(\beta(n).Start) \\ s_i^E(e) + 1 & \text{if } e \in Out(n) \\ s_i^E & \text{otherwise} \end{cases}$$
  In *one-way operation*, Invocation and While parts are not changed and their value are still the same i.e. $s_{i+1}^I = s_i^I$ and $s_{i+1}^w = s_i^w$.
- Firing rule for the Stop node of a child node of one-way operation: if $m = \beta(n).stop$ where n is an Invocation node. Firing m removes one token from its input edge. If there is only one token on the input edge, m is terminated. m is enabled if $s_i(e) > 0$ for $e \in In(m)$ then
  $$s_{i+1}^E(e) = \begin{cases} s_i^E(e) - 1 & \text{if } e \in In(m) \\ s_i^E & \text{otherwise.} \end{cases}$$

In this case the value of Invocations and While coordinates remains unchanged i.e. $s_{i+1}^I = s_i^I$ and $s_{i+1}^w = s_i^w$.

If n **is Invocation with *two-way operation***, firing of n removes one token from its input edge and add one token to the Start Node of Workflow Graph associated to n i.e. $\beta(n).Start$. In other words, in the two way operations the execution of the parent Workflow Graph is blocked while the execution of the internal behaviour, i.e. the child node, terminates. The Invocation node n is marked by "1" token to indicate that the Workflow Graph associated to n is currently executing i.e. $s_i^I(n) = 1$.

- n is enabled if $s_i(e) > 0$ for $e \in In(n)$ then
  $$s_{i+1}^E(e) = \begin{cases} s_i^E(e) - 1 & \text{if } e \in In(n) \\ s_i^E(e) + 1 & \text{if } e \in Out(\beta(n).Start) \\ s_i^E & \text{otherwise.} \end{cases}$$
  For Invocation part s_{i+1}^I:
  $$s_{i+1}^I(m) = \begin{cases} 1 & \text{if } m = n \\ s_i^I & \text{if } m \neq n. \end{cases}$$

However, the coordinates for the While part remain unchanged, i.e. $s^w_{i+1} = s^w_i$.

- Firing rule for the Stop node of a child node of a two-way operations: If $m = \beta(n).stop$ where n is an Invocation node, firing of m removes one token from its input edge and adds one token to the output edge of the Invocation node n. The token in the Invocation n node is removed and n is marked by "0" to indicate that the process of executing the Workflow Graph associated to n has been completed. m is enabled if $s_i(e) > 0$ for $e \in In(m)$ then
$$s^E_{i+1}(e) = \begin{cases} s^E_i(e) - 1 & \text{if } e \in In(m) \\ s^E_i(e) + 1 & \text{if } e \in Out(n) \\ s^E_i & \text{otherwise} \end{cases}$$
For the Invocation part:
$$s^I_{i+1}(m) = \begin{cases} 0 & \text{if } m = n \\ s^I_i & \text{if } m \neq n \end{cases}$$
However, the While part remains unchanged$s^w_{i+1} = s^w_i$.

Remark 1: Vanhatalo et al. [14] assign tokens to *only* the edges of a Workflow Graph. In this paper, tokens are also assigned to the nodes to indicate that the associated Workflow Graph for Invocation and While node is executing. This seems to be inevitable to have extra functions s^I and s^w as a part of states, as we require a mechanism to capture the scenarios that an internal Workflow Graph of an Invocation or While nodes is executing. A naive approach would be to suggest that if an internal Workflow Graph i.e. a child of a node n had a token on its input edges, it would be sufficient to show that n is executing. However, this may cause ambiguity if a new token arrives at the input edge of n. In this case a new execution of the internal Workflow Graph may be repeated. This would result in a wrong execution of the Workflow Graph such as execution of a Stop node, which implies execution has terminated, while there are still tokens in the Workflow Graph.

We shall end this section by extending the definition of execution traces from Petri net to EWFG.

Definition 7: Suppose that s_0 is a state with $s_0(e) = 1$ if $e \in Out(root.start)$ i.e. e is an edge out of the Start node of the root and $s_0(e') = 0$ for all other coordinates. We refer to s_0 as an *initial state*. For example, Fig. 5 represent the initial node for the example in Section 2.3. Any sequence $s_0 \xrightarrow{n_1} s_1 \ldots \xrightarrow{n_k} s_k$ is called an *Execution Sequence* of the Extended Workflow Graph. Sometime, if there is no chance of ambiguity, we write $s_0 \xrightarrow{n_1 \ldots n_k} s_k$ to denote the Execution Sequence. The set of all Reachable state of T is defined as $Reach(T) = \{s_k | \exists n_1 \ldots n_k$ so that $s_0 \xrightarrow{n_1 \ldots n_k} s_k\}$. We also define the *language* of an Extended Workflow Graph T as $L(T) = \{n_1 \ldots n_k | \exists s_k \in Reach(T)$ so that $s_0 \xrightarrow{n_1 \ldots n_k} s_k\}$.

4.3 Modelling Observability and failure in of Extended Workflow Graph

Assume that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph EWFG. For each $G_i \in \mathscr{V}$, $G_i = (N_i, E_i)$ where N_i is the set of nodes in G_i and E_i is the set of edges of G_i. Suppose that $N = \bigcup N_i$ is the set of all nodes of the EWFG. N can be partitioned into disjoint subsets of *observable nodes* N_{obs} (i.e. their occurrence can be observed) and *unobservable nodes* N_{uo} i.e. $N = N_{obs} \cup N_{uo}$ and $N_{obs} \cap N_{uo} = \phi$. For example, we assume that Send Item to the Supplier node in the Shop Workflow Graph is an observable action since it can be seen by the Supplier service, whereas Calculate the Cost is considered as unobservable node as it is considered an internal action of the system.

Some of the events such as Fork and Join are used across all Workflow Graph and system. We shall distinguish them from the events which are directly represent the system events.

Definition 8: (Internal Actions [7]) N_{int} represents *internal action* such as the Start node, Stop node, Fork, Join, Decision, Merge and While which their execution is performed internally and hence unobservable i.e. $N_{int} \subset N_{uo}$.

In our models some of the nodes represent occurrences of failure. The set of failure nodes is denoted by N_f. For example, a violation of a quality constraint is considered as a failure. If a node which represents an occurrence of failure is observable, then its execution can be seen and it can be diagnosed trivially. So without any loss of generality we shall assume that all failure nodes are unobservable $N_f \subseteq N_{uo}$. A typical example of failure nodes which are observable are faults in the software which are modelled as *Caught Exception* which allow the system to continue with the execution of the program. We are not interested in such failure. This is because occurrences of such events is observable and can be detected trivially. A system may have different types of failure. We shall write $N_f = N_{f_1} \cup \cdots \cup N_{f_\ell}$ to classify the the set of failure into different categories.

From an outside services point of view only observable event N_{obs} can be recognised. Suppose that the Extended Workflow Graph executes a sequence of events $\sigma = n_1 \ldots n_r$. A *Projection* map is often used to erase unobservable actions from σ to create the set of observable actions [36].

Definition 9: 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 ϵ 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.

The given example can be used to explain the observability concepts of a Workflow Graph. The set of the observable and unobservable nodes of the Workflow Graph of Supplier service of Fig. 4 is defined as follow:
$N =$*{Start Node, Receive Items List, While, Return Items List to Shop, Stop Node}*, where N_{obs}=*{Receive Items List, Return Items List to Shop}*, and N_{uo}=*{Start Node, While, Stop Node}*

Some of the scenarios of the execution in the given example can result in a failure action. In this paper, we have selected two failures specified in [32], [33] to demonstrate our approach:

- A failure N_{f_1} occurs when the customer makes a mistake while placing his order. This failure is caused by a data acquisition which may result in either billing of wrong items or billing of wrong number of items. So, the Receive Confirmation node in the Shop service is a failure of type N_{f_1}
- A failure N_{f_2} represents the case that the Supplier service does not return back the same list of items to to the Shop

service i.e. one or more items are missing from the list. In other words, N_{f_2} is a failure, which may occur after the execution of *Return Items List to Shop* node in Supplier service.

4.4 Description of the problem

Diagnosing failure during the interaction between a set of services is crucial and challenging task. The aim of our approach is to produce Diagnoser service. As shown in Fig. 7, the Diagnoser service is designed to receive a sequence of observable events to determine if a failure has happened, or may have happened. This can be formalised as follows [7]:

Definition 10: Consider an Extended Workflow Graph G with a set of nodes N. Also assume the set of failures are divided into categories $N_{f_1}, N_{f_2}, \ldots, N_{f_\ell}$. Suppose that $\sigma = n_1 \ldots n_r \in N^*_{obs}$ is an arbitrary sequence of observable actions. We say:

1) σ ends in a *Normal* state if every execution sequence of G with an observable sequence σ does not have any failure events i.e.
   $\forall \mu_1 = n'_1 \ldots n'_s \in L(G) \;\; P(\mu_1) = \sigma \Rightarrow \forall i \;\; n'_i \notin N_{f_i}$
2) σ ends in *a failure state of type* N_{f_i} if every reachable sequence of events μ_2 in N, which can be projected to σ ends in a failure node of N_{f_i} i.e.
   $\forall \mu = n'_1 \ldots n'_s \in L(G) \; P(\mu_2) = \sigma \Rightarrow \forall i < s \; n'_i \notin N_{f_i}$ and $n'_s \in N_{f_i}$
3) σ may end in *a failure state of type* N_{f_i}, if some of the execution sequence of G which map to σ end in failure N_{f_i} and some have no failure of type N_{f_i} i.e.
   $\exists \mu_1 = n'_1 \ldots n'_s, \mu_2 = n''_1 \ldots n''_q \in L(G)$ so that $P(\mu_1) = P(\mu_2) = \sigma$ and $\forall i < s \;\; n'_i \notin N_{f_i}, n'_s \notin N_{f_i}$ and $\forall i < q \; n''_i \notin N_{f_i}$

Fig. 7. An overview of the Diagnoser Service

The definition of diagnosability presented here is a weaker version of the definition by Cabasino et al. [37] for Petri nets. Cabasino et al. [37] requires an infinite trace to follow occurrence of failure to model delay in discovering the failure. We have not included that requirement in our definition. However, adopting definition similar to [37] will not require any changes to the rest of the paper, as infinite traces following a failure map into circles of unobservable events, which can be detected in the Coverability Graph. Cabasino et al. [37] also shows that for regular languages the presented definition is equivalent with a notion of diagnosability known as k-diagnosability. In Section 5, we will demonstrate that the languages of our models are regular. As a result, classical methods of creating Diagnoser can be applied in this context.

In the rest of the paper we shall demonstrate that the formalism presented in this paper is suitable for extending the Diagnosability theory [9]. To do so, we shall present a method of creating Diagnoser and integrate it into the system.

5 Coverability Graph of Extended Workflow Graph

In Petri nets, Coverability Graph is used to analyse unbounded nets [38]. In our earlier paper [7] we extended the idea Coverability Graph from Petri nets to Workflow Graph [14]. In this paper, we shall further extend the Coverability Graph to Workflow Graph with Invocation and While nodes. We also prove that for Extended Workflow Graph consisting Workflow Graph with Invocation and While nodes without unstructured loop the Coverability Graph is the same as the Reachability Graph. In other words we show that in such cases there are only finite numbers of reachable states, which can produce infinite repetitive behaviour by creating loops. The following definition of Coverability graph is a direct extension of a similar definition in Petri nets.

Definition 11: A Coverability Graph of an Extended Workflow Graphs $T = (\mathscr{V}, \Sigma)$ is a graph $G_{cov} = (N_{cov}, E_{cov})$ where :

I Each node of the Coverability Graph is marked by a $k-$dimension vector of coordinates $\mathbb{N} \bigcup \{\omega\}$, where k is the number of coordinates in the state of an EWFG as defined in Definition 4.1.

II Each edge of the Coverability Graph is marked by a node of T, i.e. $E_{cov} \subseteq N$ where N is the set of all nodes of T.

III For each reachable set of states $s_0 \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r$, 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 $\mathbb{N} \bigcup \{\omega\}$, in which $\forall n \in \mathbb{N}, n \leq \omega$.[2]

Algorithm 1 is a direct extension of Algorithm [31], [38] in Petri net. It is adapted to produce the Coverability Graph of an Extended Workflow Graph.

Lemma 1: Algorithm 1 produces the Coverability Graph for any given Extended Workflow Graph.

Proof: Similar to the proof of corresponding result in Petri net [38] with replacing semantics of Petri net with the semantics of Extended Workflow Graph. □

The idea of Coverability Graph is to represent the behaviour involving infinite number of states in a finite graph. Clearly if a Coverability Graph for a Workflow Graph has no *Structured loop*, then there is no chance of repetitive behaviour. In this case, the set of reachable states is finite, which means that the Coverability Graph is the same as the Reachability Graph.

In the representation of Workflow Graph introduced in Section 4, repetitive behaviour is modelled in a *controlled* manner using While loops. In this section we show that Extended Workflow Graph, although include infinite behaviour, the set of reachable states for them is finite. Intuitively, We will show that the set of Reachable states of the Extended Workflow Graph is a subset of the Cartesian product of the set of all Reachable states of the Workflow Graph produced from "Stripping" each WFGIW at the node of the tree from any possible Invocation or While nodes. First we shall start by

2. Readers who are not familiar with the idea of Coverability Graph in Petri net are invited to think of ω as a symbol that represents possibility of having infinite token on an edge.

Algorithm 1 The computation of Workflow Coverability Graph

Output $G_{cov} = (N_{cov}, E_{cov})$
Create an initial node $(1 \dots 0| \dots |0 \dots 0)$
Label the initial node as the root and tag it as "new".
while a node marked by "new" in the Coverability Graph exists **do**
  Select a node marked by "new" α
  if α is identical with a node on a path from the initial node to α **then**
    tag α as "old"
  else
    if no activities are enabled at α **then**
      tag α as "dead"
    else
      for all activities n_i enabled at α **do**
        compute the Marking α' that results from firing n_i at α. The firing rules which are described in Section 2.2 must be extended by $\forall n \; n + \omega = \omega + n = \omega$
        On the path from the root to α if there exists a Coverability Graph node such that $\alpha'' \leq \alpha'$ and $\alpha' \neq \alpha''$ i.e. α'' is covered by α', then replace $\alpha'(e) = \omega$ for each e such that $\alpha'(e) > \alpha''(e)$
        Introduce the new α' as Coverability Graph node
        Draw an arc with label n_i from α to α'
        Tag α' "new"
      end for
      Tag α "old"
    end if
  end if
end while

"stripping" each node of the tree from all Invocation and While node to create a conventional Workflow Graph without any *structured loops*.

Notation 5: Suppose that $G = (N, E)$ is a Workflow Graph with Invocation and While nodes. Let us denote by $\widehat{G} = (\widehat{N}, \widehat{E})$ a Workflow Graph created by replacing each Invocation and While nodes with an Activity node with the same name.

Lemma 2: Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG) where $\mathscr{V} = \{G_1, G_2 \dots, G_n\}$. Then for each i, $\pi_i(Reach(T)) \subseteq Reach(\widehat{G_i}) \bigcup \{\overrightarrow{0}\}$ where π_i is the projection of the state vector of T to edge of G_i, $\overrightarrow{0} = (0, \dots, 0)$ is zero vector of dimension of $\mid E_i \mid$, where E_i is the set of the edges of G_i.

Proof: The proof is by induction on r where $\sigma = s_0 \xrightarrow{n_1} s_1 \dots \xrightarrow{n_r} s_r$ is an execution sequence of T and s_r is the r-th reachable state. The first step of induction is trivial as if G_i is the root of the Graph $\pi_i(s_0)$ would be $(1, 0, \dots, 0)$, otherwise $\pi_i(s_0) = \overrightarrow{0}$. Suppose that for $0 \leq j \leq r$ $\pi_i(s_0), \dots, \pi_i(s_{r-1}) \in Reach(\widehat{G_i}) \bigcup \{\overrightarrow{0}\}$, we must show that $\pi_i(s_r) \in Reach(\widehat{G_i}) \bigcup \{\overrightarrow{0}\}$. In $s_{r-1} \xrightarrow{n_r} s_r$ if n_r is not an edge of G_i then $\pi_i(s_r) = \pi_i(s_{r-1})$ and there is nothing to prove. Similarly if n_r is not a While or Invocation node, the change of state of the overall system and change of states of $\widehat{G}$ are identified. So the nontrivial cases are when n_r is a While or an Invocation node. The idea of the proof is that execution of a While or Invocation node results in executing of a child Workflow Graph which makes no changes to the state of $\widehat{G_i}$. □

Lemma 3: Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG) where $\mathscr{V} = \{G_1, G_2 \dots, G_n\}$. Suppose that none of the G_i have Structured loops, then the set of reachable states of T is a finite set.

Proof: From lemma 2 we can show that $Reach(T) \subseteq (Reach(\widehat{G_1}) \bigcup \{\overrightarrow{0}\}) \times \dots \times (Reach(\widehat{G_n}) \bigcup \{\overrightarrow{0}\})$ as each coordinate of a reachable state belongs to a coordinate of one of $Reach(\widehat{G_1}) \dots Reach(\widehat{G_n})$. If G_i has no Structured loop, then $\widehat{G_i}$ has no Structured loop. Creating $\widehat{G_i}$ does not modify the topology of G, while replacing Invocation and While nodes with Activity nodes. Since $\widehat{G_i}$ has no structured loops, $Reach(\widehat{G_i})$ is finite. Hence $Reach(\widehat{G_1}) \dots Reach(\widehat{G_n})$ are all finite. As a result, $Reach(T)$ is finite □

Theorem 1: Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG) where $\mathscr{V} = \{G_1, G_2 \dots, G_n\}$. Suppose that none of the G_i have unstructured loops, then the language of T is a regular language.

Proof: Since the set of all Reachable states is finite, the Coverability Graph captures all possible Reachable states as an Automata. So, the language of T is regular. □

Sampath et al. [9] present a necessary and sufficient conditions for the diagnosability in DES of the regular languages.

Corollary 1: An Extended Workflow Graph language L is diagnosable with respect to the projection P and the failure partition N_f if and only if it does not include a cycle of unobservable events.

6 The Diagnoser of Extended Workflow Graphs

Fig. 8 captures an overall picture of the behaviour of the Extended Workflow Graph in terms of involving states and evaluation from one state to another state when a node gets executed. However, many of the nodes in the graph are not observable. In this section, we shall introduce the idea of *Diagnoser Coverability Graph (DCG)* which serves two purposes. Firstly, a DCG is an approximation of the behaviour of the EWFG to include *only* the behaviour which is observable. To be precise, for any observable sequence of actions σ, there is a path in the DCG marked by σ starting from the root of the DCG to a node of the DCG which contains the *set of all states* s so that there is a sequence μ with $s_0 \xrightarrow{\mu} s$ with $P(\mu) = \sigma$. This would indeed provide an approximation of σ, as the node of the DCG that contains s also included all states of the DCG which are reached from s via firing of unobservable events. In [36], [9], [11], this is refereed to as the *Unobservable Reach* of s.

Definition 12: Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph and s is a Reachable state of T. We shall define the Unobservable Reach of s as follows: $UR(s) = \{s_r | s \xrightarrow{n_1} s_1 \dots \xrightarrow{n_r} s_r, \forall i \; n_i \in N_{uo}\}$.

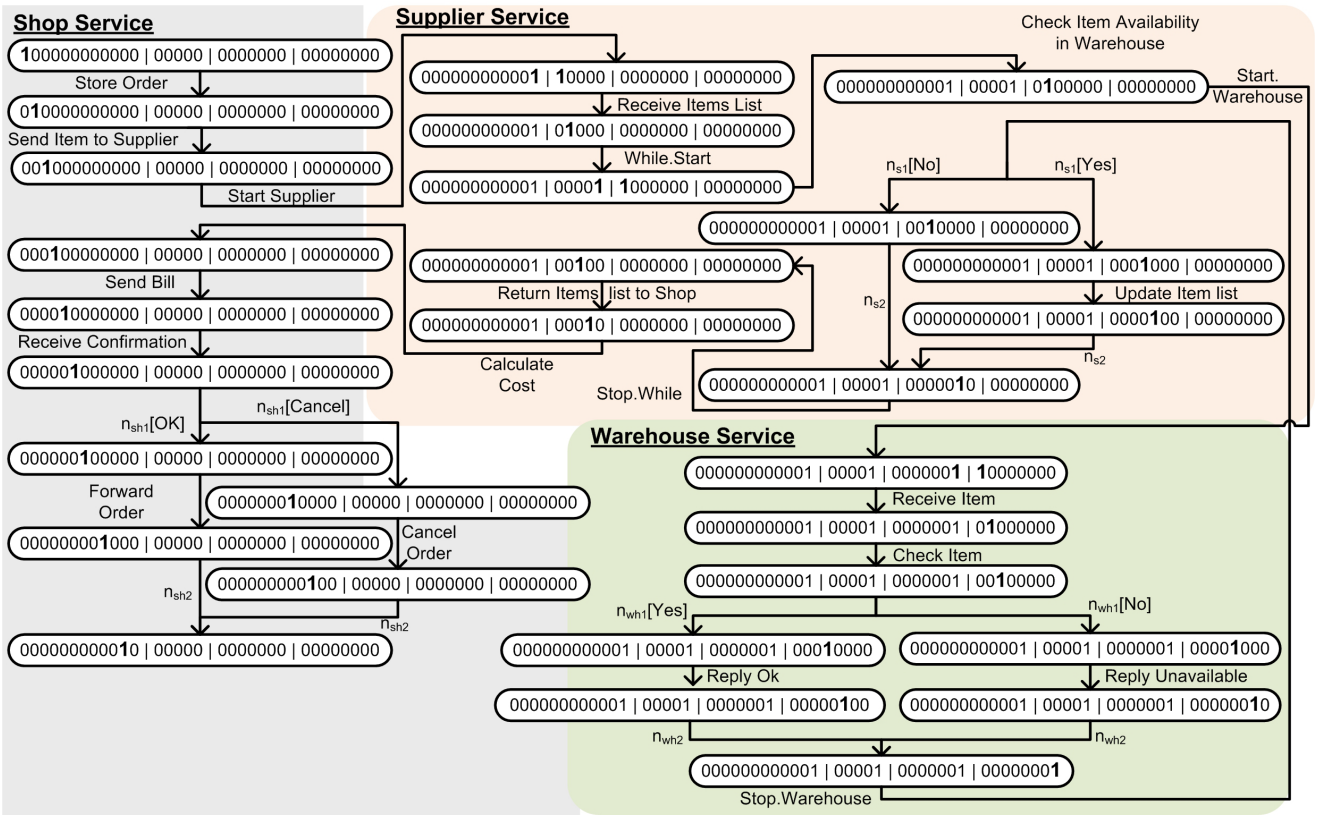

Fig. 8. The Coverability Graph of the E-shopping Example

The second purpose of a DCG is to encode information about the failure state of the system into the approximated states. The encoded information represents the type of the failure that can occur when the system arrives at a given state.

Definition 13: Suppose that T is an Extended Workflow Graph. Each *State of a Diagnoser* of Extended Workflow Graphs is denoted by (α, ϕ) where α is a state of the EWFG T and $\phi \in \{0,1\}^{\ell}$ in which ℓ is the number of categories of failures. Intuitively speaking, if the coordinate i of ϕ is equal to 1, we infer that "under the state α, a failure of type N_{f_i} has happened".

We wish to warn the reader that the word "*state*" is used both for referring to the State of an EWFG and also the States of a Diagnoser. If there is no chance of confusion, we shall use the phrase "state" for both of them.

Notation 6: We shall sometimes denote a State (α, ϕ) of the Diagnoser

$$(\underbrace{\underbrace{s_1^1 \dots s_{n_1}^1 | s_1^2 \dots s_{n_2}^2 | \dots | s_1^k \dots s_{n_k}^k}_{\text{State of the EWFG}}}_{\alpha} \mid \underbrace{\underbrace{\phi(1) \dots \phi(l)}_{\text{encoding of failure}}}_{\phi})$$

where $(s_1^1 \dots s_{n_1}^1 | s_1^2 \dots s_{n_2}^2 | \dots | s_1^k \dots s_{n_k}^k)$ is already defined in Section 4.1 and $\phi = (\phi(1) \dots \phi(l))$ is the encoding of failure in which l is the number of failure specified for the system, where $\phi(i) = 1$ if a failure of type N_{f_i} has occurred.

Each node of the DCG is marked by a set $\{(\alpha_1, \phi_1), \dots, (\alpha_r, \phi_r)\}$, where $\alpha_1, \dots, \alpha_r$ are Reachable states of the Extended Workflow Graph. As explained earlier, $\alpha_1, \dots, \alpha_r$ contains all states of the EWFG obtained from firing of unobservable transitions, i.e. $\{\alpha_1, \dots, \alpha_r\} = UR(\alpha_1, \dots, \alpha_r)$. To obtain the failure vector coordinates $\phi_1, \dots, \phi_r$, a new function called Label Propagation Function is required [9]. The function modifies the failure state, in case of occurrence of a failure and also, as the name suggests, propagates the information about the failure in one state to a subsequent state.

Definition 14: Suppose that T is an Extended Workflow Graph with the set of Reachable States $Reach(T)$ and the set of nodes N. A Label Propagation Function is a function $LP : Reach(T) \times \{0,1\}^{\ell} \times N \to \{0,1\}^{\ell}$, so that $LP(\alpha, \phi, n) = \phi'$ where the $i-$th coordinate of ϕ' is defined by

$$\phi'(i) = \begin{cases} 1 & \text{if } n \in N_{f_i} \text{ executes at state } \alpha \\ \phi(i) & otherwise. \end{cases}$$

ϕ' is the encoding of failure for the state which results from the firing of n under the state α. If $\alpha_0 \xrightarrow{n_1} \alpha_1 \dots \xrightarrow{n_r} \alpha_r$ we will abuse the notation and write $LP(\alpha, \phi, n_1 n_2 \dots n_r)$ to represent successive application of LP to $n_1, n_2, \dots, n_r$. Next we shall need a final piece of notation before presenting an algorithm for creating the Diagnoser Coverability Graph of an Extended Workflow Graphs (DCG).

Notation 7: Suppose that (α_i, ϕ_i) appears in the labelling of one of the nodes of the DCG. We write $\mathbf{F}(\alpha, \phi)$ to denote the set of all (β, ϕ') for which there is a sequence of Unobservable events $n_1 n_2 \dots n_r$ such that $\alpha \xrightarrow{n_1 \dots n_r} \beta$ and $\phi' = LP(\alpha, \phi, n_1 \dots n_r)$.

The function $\mathbf{F}$ will be used to calculate the nodes of the Diagnoser Coverability Graph.

Algorithm 2 Creating Diagnoser Coverability Graph (DCG)

INPUT: Reachability Graph of an EWFG T
OUTPUT: Diagnoser Reachability Graph $G_{DCG} = (N_{DCG}, E_{DCG})$
Create a first node of DCG, mark it "new", and include in it a vector $(\alpha, \mathbf{0})$, where α is the initial node of the Reachability Graph and $\mathbf{0}$ is a vector with coordinates 0 of dimension l, the number of failure categories
while there exist a node of the DCG tagged "new" **do**
  Select a State of the DCG $S = \{(\alpha_1, \phi_1), \ldots, (\alpha_r, \phi_r)\}$ which is tagged by "new"
  Iterate through the list (α_i, ϕ_i) one-by-one
  for all observable actions which are enabled under α_i with $\alpha_i \xrightarrow{n} \lambda$ **do**
    Let $S' := \{(\lambda, LP(\alpha_i, \phi, n))\}$
    $S' := S' \cup \mathbf{F}(S')$
    if S'_i already exists in DCG **then**
      discard it
    else
      Create a State node marked by S' and tag it as "new"
      draw an arc from S to S' marked by n
    end if
    Remove the tag "new" from S after finishing the iteration
  end for
end while

Example: Applying the Diagnoser Coverability Graph algorithm to the example of Section 2.3, has resulted in Fig. 10. It can be seen that the states of of the Diagnose are similar to the states of the Coverability except there are new part add to the vector to encode the failure information. The vector for failure part has a length 2 since we assume that there are two type of failures for this example N_{f_1} and N_{f_2} as explained in Section 4.3. For example, Fig. 9 depicts the system state which has a failure of type N_{f_1}. This state may be reached after executing the node called Return Items to Shop in the Supplier Workflow Graph as shown in Fig. 4. In the Shop part, a token is assigned to the last coordinate which is associated to the Invocation node to indicate that the Supplier Workflow Graph is executing as an internal action of the Invocation node. As explained in Section 4.1, a token is also assigned to the 4-*th* coordinate of Supplier part corresponding to the output edge of the node called Return Items list to Shop. The failure part shows that after this execution a failure of type N_{f_1} has occurred.

$$(\underbrace{00000000001}_{\text{Shop}}|\underbrace{00010}_{\text{Supplier}}|\underbrace{0000000}_{\text{While node}}|\underbrace{0000000}_{\text{Warehouse}}|\underbrace{10}_{\text{Failures}})$$
($|s^E|$=12, $|s^E|$=5, $|s^E|$=7, $|s^E|$=8, $|N_{f_\ell}|$=2)

Fig. 9. An example of Diagnoser state

7 Implementation and Evaluation

The presented approach has been implemented as a Plugin for the Oracle JDeveloper. An outline of the stages involved in the creation of the Diagnoser Service is depicted in Fig. 11. Firstly, the user produces a model of the System in Oracle JDeveloper or upload existing model into the tool. Then, our tool extracts an equivalent Extended Workflow Graph from the BPEL files and their XML Schema Definition (XSD). Then, the tool applies the Algorithm 1 explained in Section 5 to produce the Coverability Graph. We showed in Section 5 that the Coverability Graph is the same as the Reachability Graph. Next, the Algorithm 2, which was explained in Section 6, is applied to produce Reachability Graph to generate the Diagnoser Coverability Graph (DCG). Finally, the DCG is implemented as a working Diagnosing Service which is deployed with the rest of the system. The remaining of this section discusses the method of transforming the DCG to produce a Diagnosing Service.

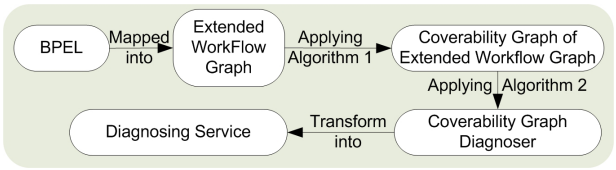

Fig. 11. An outline of the implementation method

One of the key tasks of the Diagnoser is to ensure that any given execution sequence respects the language of the system. This task is achieved by adopting the idea of the lexical analysis used in Compilers theory [39] as a parse function to check whether the grammatical structure of a given text respects a specific regular language. Theorem 1 shows that the Extended Workflow Graph presented in this approach is considered as a regular language. After ensuring the correctness of the execution trace, the Diagnoser applies the diagnosability theory presented in Section 6 to infer whether the execution results in a failure or a normal state.

7.1 Implementing the Diagnoser

There are various ways to implement the Diagnoser. We shall next present two possible methods of Implementation.

Creation of Diagnoser as a BPEL Service: The produced Diagnoser can be implemented as BPEL service interacting with already existing services. In a nutshell, such a BPEL service includes a *Switch* activity involving a number of *Cases* corresponding to the observable events in the Diagnoser model which capture all possible observable traces. 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 4.3. In particular, in case of a failure, the type of failure and the event which has caused the failure will be included in the diagnosis result.

Creation of Diagnoser as Web Service: The produced Diagnoser can be implemented as a stand-alone Java class

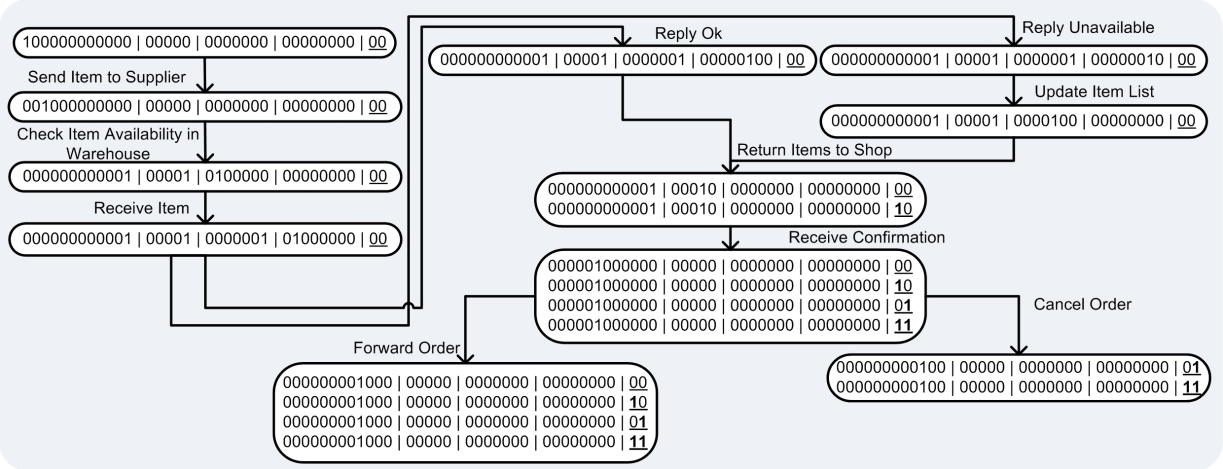

Fig. 10. The Diagnoser Coverability Graphs of Extended Workflow Graphs of the E-shopping Example

deployed Web service. In this case, similar to the previous implementation, conditional statements in form of *if-then-else* will be used.

In both ways, as shown in Fig. 7 the Diagnoser Service is incorporated into the system to receive the observable events that has been executed, then it responses with the diagnosing result describing the behaviour of the system which is either in normal state or a failure has occurred.

7.2 Integrating the Diagnoser into the system

After producing the Diagnoser, there are various ways to integrated the created Diagnoser into the system to interact with a bunch of exiting services. We shall present two of such methods.

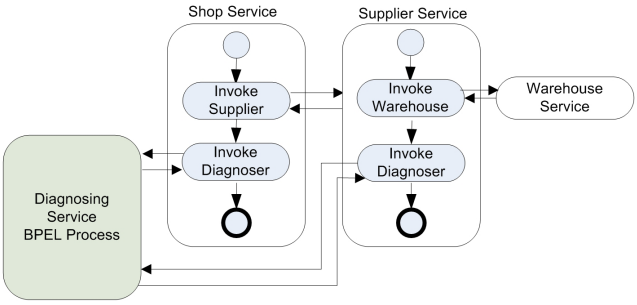

Fig. 12. Example of method 1

Integrating the Diagnoser by adding extra Invocations: To fulfil the diagnosing task in this way each business process is automatically annotated by including extra *Invoke* activities to execute the Diagnoser Service after each invocation task. Fig. 12 represents the interaction between the Diagnoser Service and three services, which are *Shop*, *Supplier* and *Warehouse*, which are explained in Section 2.3. This interaction is a Choreography since the interaction is is achieved by a collaboration between a collection of services.

Integrating the Diagnoser by adding a Protocol Service: In this method the Diagnoser is produced with a new service called a *Protocol Service*. The *Protocol Service* is generated automatically to accomplish the interaction between the Diagnoser Service and the existing services. The *Protocol Service* should interact with the generated Diagnosing Service after each invocation. Then, the Diagnosing Service will return the overall result of its diagnosing to the *Protocol Service*. The diagnosing result includes information about the behaviour of the system after the invocation, which either in normal behaviour or an indication that a failure has occurred. In case that a failure has occurred, the type of the failure is presented. As a result, in this method there is no need to conduct any activity inside the BPEL Service for interacting with Diagnoser.

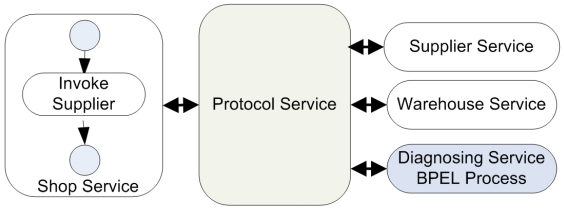

Fig. 13. Example of method 2

7.3 Evaluation

Evaluating the amount of resources which are required to implement the Diagnoser is a requisite task. For this reason the evaluation of our approach is focused on the performance. Our tool supports four different methods of implementations to incorporate the the produced Diagnoser into the system as described Table 1. For example, in Method 1, the Diagnoser is created as BPEL file which is integrated into the system by adding invocations.

Experimental Results

The presented four methods have been tested in terms of performance. A common practise of evaluating of services in term performance is *Stress Testing* which identifies and verifies the stability, capacity and the robustness of services

[17], [40]. The idea of *Stress Testing* is based on measuring the execution time of a number of threads executing services. The detail of the experiments test can be explain similarly to our experiments on the classical Diagnoser [6].

Fig. 14 illustrates the result of the *Stress Testing* of processing different number of threads in second. The result of the experiments shows that integrating the Diagnoser with a help of the *Protocol Service* results in slightly better performance in term of maintainability and modularity point. However, it can be argued that using the *Protocol Service* may result in bottlenecks affecting the performance of the system. Various distributed diagnosing scheme are proposed to enhance and address this issue [41], [42], [43], which will be used as future research. Data related to experiments is available at [44].

8 Related Works

Recently, several fault diagnosis methods for SoA based on Discrete Event Systems theory have been proposed. Yan et al. [8], [3] formalise BPEL model with synchronised automata. The focus of this approach is on failures caused by exceptions such as mismatching data. So, to diagnosis the failure the execution traces of events leading to the exception are used. Our methods differs from them in a number of ways. Firstly, they do not distinguished between observable and unobservable events. Secondly, a "While" activity in [8] results in a cycle which is not compatible with BPEL [21], [22] as discussed in Section 3. Thirdly, the method suggested here results in automated creation of Diagnoser. Finally, they do not study the integration of the produced Diagnoser.

Baresi et al. [45] propose a monitoring language called Service Centric Monitoring Language (SECMOL) which is an extension of WS-Agreements [46]. SECMOL is a specification language based on run-time quality assessment for monitoring data. In our approach, we do not focus on monitoring data, as we aim to monitor systems which can fail because of failure of the underlying services, or failures which can be through erroneous interaction between services resulting in undesirable scenarios. A failure caused by a breakdown of each service is often dealt with exception handling, whereas detection of failure caused by wrong execution of a business process often requires provision of additional infrastructure to monitor interactions of the services.

Giua and Seatzu [13], [23] and Genc and Lafortune [10] deal with the diagnosis of failure in Petri net models. In common with their approach, we are using Coverability Graph. Coverability Graph of Petri net can include ω. In this paper, we proved that the Coverability Graph of an Extended Workflow Graph does not include any ω. This does not mean that an EWFG can not include infinite behaviour. It is indeed possible to include a repetitive behaviour in BPEL file. In Genc and Lafortune [10], the presented algorithm results in distributed fault diagnosis. Indeed in this paper, producing the Diagnoser as a centralised service is used as proof of concept. It can be argued that using distributed architecture would be more realistic than the centralised one which may result in bottlenecks affecting the performance. As future works, We shall extend our theory to implement a Decentralised Diagnosers.

TABLE 1
Different methods of implementations

Integration via \ Diagnoser as	BPEL	Web Service
adding extra Invocations	Method 1	Method 3
Protocol Service	Method 2	Method 4

In our earlier papers [5], [6], a Model Driven Development approach to the design and implementation of Diagnoser Service for a group of interacting services is proposed. The method involves transforming BPEL files to automata automatically. Then, DESUMA [47] framework is used to create the Diagnoser. A second transformation maps the produced Diagnoser back-into a BPEL service and deploys it to interact with the existing services. Similarly, Li et al. [48] transform BPEL into coloured Petri nets. It can be argued that such approaches, including [5], [6], require proving the correctness of the transformation which is a formidable task.

Failures caused by a violation of constraint have been also studied in the past few years [49], [48]. For example, Baresi et al. [49] propose a method to tackle the dependability of Business processes modelled using Business Process Modelling Notation (BPMN). This method is based on introducing a concept of Supervision Rule as a set of defined constraints and reaction strategies. The constraints are used to monitor how the business process evolves, while reaction strategies specify actions that must executed when constraints are violated. In contrast to such approaches, we are interested in the failures related to the underlying logic for the Business Process. A main advantage of such research is the use of Data Flow Models. Incorporating data flow into our approach remains an issue for future work.

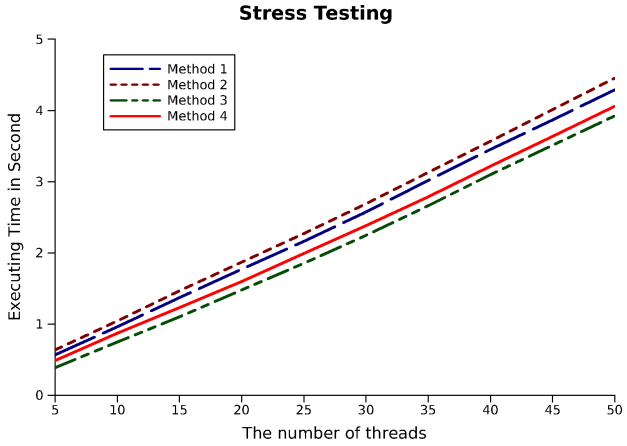

Fig. 14. Stress Testing Result

9 Conclusion

A main prerequisite for the creation of a model-based fault tolerance approach is a suitable modelling language. This paper extends an existing modelling language called Workflow Graph by including new constructs based on the well-accepted stander BPEL. We demonstrated that adding the

constructs in the present form avoids unstructured loops which are neither supported by the standers nor adopted by tool vendors. Building on the formalism underlying Workflow Graphs, which itself has its roots in Petri nets, we have formalised our modelling language. This is used to prove that models created from this widely used subset of BPEL produce regular languages. This result to the best of our knowledge is original. To demonstrate that the presented formalism is suitable for the model-based diagnosis, a method of generation of Diagnosers based on Discrete Event System (DES) used as a proof of concept. We have extended and adopted exiting algorithms form DES for the new modelling language. The approach is implemented in a tool, as a plugin to Oracle JDeveloper, which allows multiple BPEL models of the system to be uploaded. The tool automatically produce the Diagnoser and the infrastructure for its integration into the system. The production of the Diagnoser and integration mechanism is automated. In total four types of integration mechanism are presented and compared.

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 Conf. on Web Services*, 2007, pp. 51–59.

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

[5] M. Alodib, B. Bordbar, and B. Majeed, "A model driven approach to the design and implementing of fault tolerant service oriented architectures," in *IEEE Conf. on Digital Information Management*, 2008, pp. 464–469.

[6] M. Alodib and B. Bordbar, "A model driven architecture approach to fault tolerance in service oriented architectures, a performance study," in *Proceedings of the 12th Enterprise Distributed Object Computing Conference Workshops*, 2008, pp. 293–300.

[7] ——, "A model-based approach to fault diagnosis in service oriented architectures," in *In Proceeding of the IEEE European Conference on Web Services (ECOWS)*, Netherlands, 2009, pp. 129–138.

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

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

[10] S. Genc and S. Lafortune, "Distributed diagnosis of discrete-event systems using petri nets," in *International Conference on Applications and Theory of Petri Nets*, 2003, pp. 316–336.

[11] G. Jiroveanu, R. K. Boel, and B. Bordbar, "On-line monitoring of large petri net models under partial observation," *Discrete Event Dynamic Systems*, vol. 18, no. 3, pp. 323–354, 2008.

[12] 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.

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

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

[15] W. M. P. v. d. Aalst, A. Hirnschall, and H. M. W. E. Verbeek, "An alternative way to analyze workflow graphs," in *Proceedings of the International Conference on Advanced Information Systems Engineering*, London, 2002, pp. 535–552.

[16] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *The Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

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

[18] T. Allweyer, *BPMN 2.0*. BoD, 2010.

[19] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, 3rd ed. London: Academic Press, 1972.

[20] E. Dijkstra, "Go to statement considered harmful," pp. 27–33, 1979.

[21] M. Fasbinder, "Using loops in websphere business modeler v6 to improve simulations and xport to bpel," WebSphere Software Technical Sales, IBM, Tech. Rep., 2007.

[22] J. Koehler and J. Vanhatalo, "Process anti-patterns: How to avoid the common traps of business process modeling, part 1," IBM WebSphere Developer Technical Journal, Tech. Rep., 2007.

[23] 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.

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

[25] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0," 2006.

[26] BEA, IBM, Microsoft, A. SAP, and S. Systems, "Business process execution language for web services. version 1.1," 2003.

[27] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. T. Nagy, I. Trickovic, and S. Zimek, "Web service choreography interface (wsci) 1.0," World Wide Web Consortium,, Tech. Rep., 2002.

[28] M. Friess, E. Fussi, D. Konig, G. Pfau, S. Ruttinger, F. Schwenkreis, and C. Zentner, "Websphere process server v6 - business process choreographer: Concepts and architecture," *IBM Group*, 2006.

[29] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Conceptual modeling of workflows." Springer-Verlag, 1995, pp. 341–354.

[30] W. Sadiq and M. E. Orlowska, "Applying graph reduction techniques for identifying structural conflicts in process models," in *Proceedings of CAiSE*, 1999, pp. 195–209.

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

[32] X. Le Guillou, M.-O. Cordier, S. Robin, and L. Rozé, "Chronicles for On-line Diagnosis of Distributed Systems," Research Report, 2008.

[33] X. Le Guillou, M.-O. Cordier, S. Robin, and L. Rozé, "Chronicles for on-line diagnosis of distributed systems," in *Proceeding of ECAI*, The Netherlands, 2008, pp. 194–198.

[34] Z. Ammarguellat, "A control-flow normalization algorithm and its complexity," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 237–251, 1992.

[35] J. Koehler and R. Hauser, "Untangling unstructured cyclic flows - a solution based on continuations," 2004, pp. 121–138.

[36] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2007.

[37] M. Cabasino, A. Giua, S. Lafortune, and C. Seatzu, "Diagnosability analysis of unbounded petri nets," in *CDC09: Proceedings of the 48th IEEE Conference on Decision and Control*, Shanghai, China, 2009.

[38] W. Reisig, *Petri nets: an introduction*. New York: Springer-Verlag, 1985.

[39] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. USA: Addison-Wesley, 1986.

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

[41] Y. Wang, T.-S. Yoo, and S. Lafortune, "Diagnosis of discrete event systems using decentralized architectures," *Discrete Event Dynamic Systems*, vol. 17, no. 2, pp. 233–263, 2007.

[42] R. Debouk, S. Lafortune, and D. Teneketzis, "A coordinated decentralized protocols for failure diagnosis of discrete event systems," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 10, no. 1, pp. 33–86, 2000.

[43] A. Aghasaryan, E. Fabre, A. Benveniste, R. Boubour, and C. Jard, "Fault detection and diagnosis in distributed systems: An approach by partially stochastic petri nets," *Discrete Event Dynamic Systems*, vol. 8, no. 2, pp. 203–231, 1998.

[44] www.cs.bham.ac.uk/~bxb/public/mxa/data2010.pdf.

[45] L. Baresi, S. Guinea, O. Nano, and G. Spanoudakis, "Comprehensive monitoring of bpel processes," *IEEE Internet Computing*, vol. 14, no. 3, pp. 50–57, 2010.

[46] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web Services Agreement Specification (WS-Agreement)."

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

[48] Y. Li, L. Ye, P. Dague, and T. Melliti, "A decentralized model-based diagnosis for bpel services," in *Proceedings of the Inte. Conference on Tools with Artificial Intelligence*, USA, 2009, pp. 609–616.

[49] L. Baresi, S. Guinea, and M. Plebani, "Business process monitoring for dependability," pp. 337–361, 2007.