A model driven approach to the design and implementing of fault tolerant Service oriented Architectures

Mohammed Alodib[1], Behzad Bordbar[1] and Basim Majeed[2]

[1] School of computer Science, University of Birmingham, UK
M.I.Alodib,B.Bordbar@cs.bham.ac.uk
[2] British Telecom, Adastral Park, Ipswich, UK
Basim.Majeed@bt.com

Abstract

One of the key stages of the development of a fault tolerant Service oriented Architecture is the creation of Diagnosers, which monitors the system's behaviour to identify the occurrence of failure. This paper presents a Model Driven Development (MDD) approach to the automated creation of the Diagnosing Services and integrating them into the system. The outline of the method is as follows. BPEL models of the services are transformed to Deterministic Automaton with Unobservable Event representations using MDD transformations. Then, relying on Discrete Event System techniques a Diagnoser Automaton for the Deterministic Automata are created automatically. Finally, the Diagnoser Automaton is transformed into a new BPEL representation, which is integrated into the original architecture.

1. Introduction

One of the crucial steps in building fault tolerant Service oriented Architectures (SoA) is to diagnose the occurrence of the failure automatically. This is often achieved by the creation of the *Diagnoser* which allows monitoring of a group of services and their interactions to identify an occurrence of a failure [1, 2]. Although diagnosability is a new area of research in SoA, researchers in Discrete Event System (DES) Community have been dealing with similar challenges for the past two decades [3]. The DES community mostly uses representations such as automata [3] or Petri net [4] for the modelling of the systems and Diagnosers. On the other hand, SoA makes use of languages such as BPML and BPEL [5] for modelling the services and business processes. There is a clear scope for adopting methods used in DES and applying them in SoA.

The method presented in this paper aims to harness the capability of Model Driven Development (MDD) [6] to automatically generate a *Diagnosing Services* using DES methods. A *Diagnosing Service* can be implemented to interact with the existing services. The presented approach is implemented as a tool which makes use of a sequence of model transformations to create the *Diagnosing Service* for the system. Firstly, BPEL representations of the system are transformed into a variant of automata called Deterministic Automata. Then, applying DES techniques *Observer Automaton* is produced to generate the *Diagnosing Service*. The approach is applied to a case study involving Right-First-Time failures, in which a Customer Support System fails to complete a task First-Time and is forced to repeat a part of the task again, causing violations of Service Level Agreements (SLA).

The paper is organized as follows. Section 2 briefly reviews the preliminary material used in the rest of the paper. Section 3 presents an outline of a running example, which will be used in the rest of the paper. The approach adopted in the paper is explained in section 4. Section 5 discusses the related work and the conclusions are given in section 6.

2. Preliminaries

Diagnosability of Discrete-Event Systems: A Discrete Event System (DES) is a *discrete-state*, *event-driven* system whose state depends on the occurrence of asynchronous discrete events over time [7]. There are a variety of languages used for capturing DES models such as variants of automata and Petri net [7]. Although the approach presented in this paper is independent of the language adopted, a variant of Deterministic Automaton known as Deterministic Automaton with Unobservable Events [3] will be used to describe the approach.

A Deterministic Automaton with Unobservable Events, or simply a Deterministic Automaton is a four tuple G:=(X , Σ , δ , x_0), where X is a finite set of *states*, Σ denotes a set of *events*, $\delta \subseteq X \times \Sigma \times X$ represents the transition between the states. Here, $x_0 \in X$ is called the *initial state*. Some of the events in a

DES are *observable*, for example the events specified at the interfaces of the Web services. An event which is not observable is called an *unobservable* event. Internal action of service and events which represent a failure are example of unobservable events. Without any loss of generality, it can be assumed that all failure events are unobservable.

The purpose of the diagnosis is to use a model of the system to identify the occurrence of failure. Since a failure is assumed to be unobservable, it can not be detected at the time of its occurrence. As a result, the model of the system is used to monitor its behaviour in order to reduce the uncertainty [7]. To achieve this, from a Deterministic Automaton, a new model called an *Observer Automaton*, or *Observer* for short, is created. The Observer of the system describes the current state of the system after the occurrence of observable events [3]. From the Observer a new automaton, called the Diagnoser Automaton is created which is used to achieve the diagnosis when it observes the behaviour of the system. A *Diagnoser Automaton* is modelled as $G_d = (Q_d , \Sigma_o , \delta_d , q_0)$ where Q_d is the subset of the observable state which includes all the states which can be reached from the initial state under a specific transition δ_d [8]. Each state in Q_d is described by its name and a set of Labels which describe the type of failure that has occurred. As result, a Label either, represents a *normal* status, denoted by *N*, or a failure state which can be identified by a subset of failure types (F_1, F_2, ….F_m) to clarify what type of failure has happened [3, 9]. Hence a Diagnoser is produced to server two main purposes: firstly online detection and isolation of failure ("Did a fault happen or not?", "What type of fault happened?"). Secondly offline verification of diagnosability properties of the system [7]. For further information about DES and algorithms for creating the Diagnosers automaton we refer the reader to [3, 9].

Model Driven Development: The method adopted in this paper relies on Model Driven Development (MDD) [6] techniques for defining and implementing the chain of transformations resulting in the creation of the Diagnoser model. Each Model is based on a specific *metamodel*, which defines the elements of a language, which can be used to represent a model of the language. In the MDD a model transformation is defined by mapping the *meta-elements*, constructs of the metamodel, of a *source language* into *meta-elements* of the *destination language*. Then every model, which is an *instance of* the source metamodel, can be automatically transformed to an instance of the destination metamodel with the help of a model transformation framework such as OpenArchitectureWare [10] and SiTra [11]. For future information the MDD, we refer the reader to [6] or www.omg.com/mda.

SoA and Web services: Service Oriented Architecture (SoA) provides the foundation for implementing business processes via the composition of existing services. Web services [5] are software systems which make use of well-accepted standards and XML languages to support the creation of SoA. The interaction between services in this paper is captured via Business Process Execution Language (BPEL) [12]. 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) [13], usually in WSDL file [5]. For further information about Web services, we referee the reader to [5].

3. Example: Right-First-Time failure

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

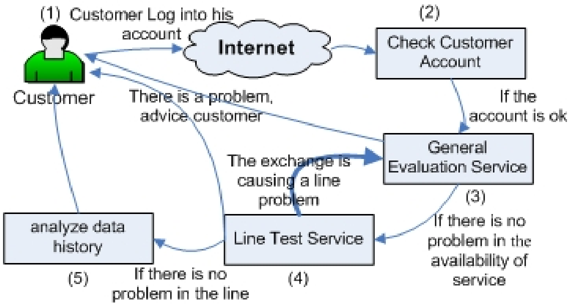

(1) Customer Log into his account
Customer
Internet
(2)
Check Customer Account
If the account is ok
There is a problem, advice customer
General Evaluation Service
The exchange is causing a line problem
(3)
If there is no problem in the availability of service
analyze data history
Line Test Service
(5)
If there is no problem in the line
(4)

Figure. 1. An Overview of the Interaction.

As depicted in Figure1, 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 centre 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

[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.

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).

4. An MDD approach to the design of Diagnosing Service in SOA

Consider a number of services which interact with each other. The behaviour of these services and their interaction is captured by a number of BPEL files. The outline of our method is depicted in Figure 2. First, BPEL representations are annotated to identify the observable and unobservable events. Then, a model transformation (BPEL2FSM) is used to transform the annotated BPEL models automatically to a Deterministic Automaton. Next, applying classical theories of diagnosability [3] a Diagnoser is computed and created, this is denoted by the arrow marked as *Generate Diagnoser* in Figure 2. Then the second model transformation (Diag2BPEL) produces a new BPEL process which represents the *Diagnosing Service* for the original BPEL models. The *Diagnosing Service* is designed to receive the current state of the system as input. Then, it responses with *diagnosing result* which describes the system behaviour whether it is *normal* or a failure has occurred. If the system status had a failure, the Diagnoser specifies which event caused this failure.

The presented approach is implemented as a Plug-in for Oracle JDevloper. First, each BPEL file and its XSD are combined together and are transformed into a Deterministic Automaton via BPEL2FSM. Then UMDES-LIB [14] is used to produce a Diagnoser automaton. Finally, Diag2BPEL method transforms the Diagnoser Automaton into the *Diagnosing Service*. Both BPEL2FSM and Diag2BPEL are implemented with the help of SiTra [11]. The details of the case study are available at [15].

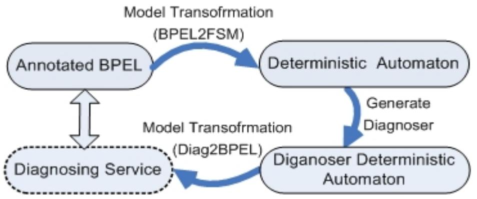


Figure 2. Applying MDD to Produce Diagnosers

BPEL model for the example of section 3: Due to space restriction the scenario described in Section 3 is modelled with the help of only two services: *Customer Service* and *General Evaluation Service*. Figure 3(i) shows the Customer Service BPEL modelled in Oracle JDeveloper. The scenario described in section 3 consists of eight main activities which are marked by (*). The flow of activities depicted in BPEL file describes the actions captured in Figure 1. For example, after checking the customer account (*CheckCustomerAccount*) there is a switch depicted (◆) which result into alternating cases either *GeneralEvaluationService* activity or cancellation of the request (*Cancel_Request*). The variables and data used in BPEL file are declared at the XML Schema Definition (XSD). For example, *CustomerServiceProcessRequest* which represents input variable used to input the customer ID (*InputCustID*). This is captured as XSD file in Figure 3. Figure 3(ii) represents the General Evaluation Service BPEL which can be explained similarly. The BPEL files and related XSD are available from [15].

4.1. Annotating BPEL

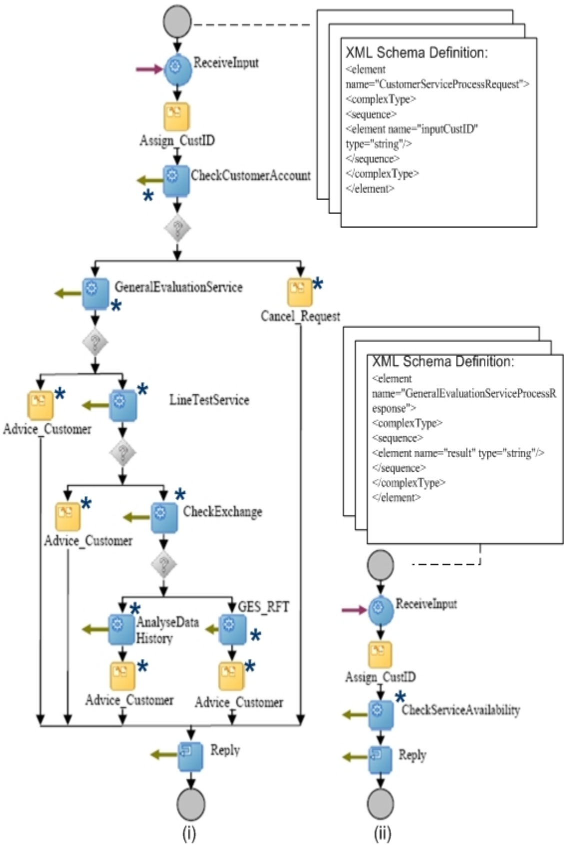
In order to apply DES techniques, BPEL models representing the services must be transformed into their equivalent Deterministic Automaton with Unobservable Event. To do so, the BPEL representations must be augmented to allow identifying, for example which events are observable or which events represent the failure action. Such information is not included in a BPEL file; a common practice is to annotate the BPEL file to include such information [2].

Three main types of annotations are conducted: annotation to include information with regards to *states*, *actions* and *failures*. Next, we will explain the annotation of the state with the help of an example. Annotation related to the actions and failures can be explained similarly.

Annotation to include States: In contrast with DES, web services tend to adopt a process oriented approach, focusing on the activities and their execution. BPEL files do not include any inherent notion of States. As a result, we will annotate BPEL file by including new attributes tags representing the states. Following the lead of Yan et al. [2] a new BPEL attribute *State* will be declared. This new variable is added to the XML Schema Definition

(XSD) part of the BPEL file, where the input and output variables are declared. For example, the following snippet of XML represents the input variables of states in *General Evaluation Service*. It can be seen that there are total of three states named as *GES1*, *GES2* and *GES3*. Moreover, the state *GES1* is an initial state.

```
<element name="states">
<complexType><sequence>
<element name="GES1" type="string"
xml:marked="0"                              (1)
xml:initialstate="yes"/>
 <element name="GES2"
type="string" xml:marked="0"/>
</sequence></complexType>
</element>
```



```
XML Schema Definition:
<element
name="CustomerServiceProcessRequest">
<complexType>
<sequence>
<element name="inputCustID"
type="string"/>
</sequence>
</complexType>
</element>
```

```
XML Schema Definition:
<element
name="GeneralEvaluationServiceProcessR
esponse">
<complexType>
<sequence>
<element name="result" type="string"/>
</sequence>
</complexType>
</element>
```

Figure. 3. BPEL for Customer Support System.

4.2. Transformation from BPEL to Deterministic Automaton

To define the transformation three items are required: metamodel for the annotated BPEL, metamodel of Deterministic Automaton and the transformation rules from the annotated BPEL to the Deterministic Automaton. Figure 4 depicts a part of the BPEL metamodel [16], which also includes the *meta-elements* related to the annotations. For example, it can be seen that *Invoke, Reply, Receive* and *Assign* activities models have new attributes which are used to annotate the BPEL file as described in section 4.1. These new set of attributes are *controllability*, *observability*, *current state*, *next state*, *isFailure* and *typeFailure*.

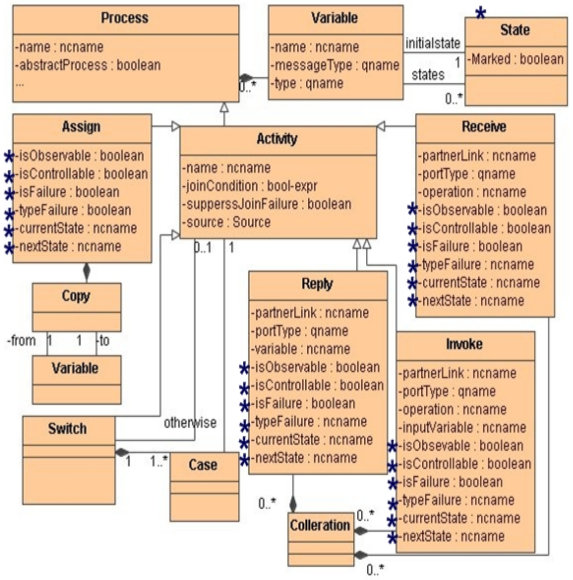


Figure. 4. A fragment of BPEL metamodel with added elements marked by (*)

Figure 5 represents a metamodel for Deterministic Automaton with Unobservable events, which is based on [17]. It can be seen that a number of *states* are connected to each other with the help of *Transitions*. Each *Transition* between two *States* is *Trigger*ed by an *Event*, which has further attributes to define the *observability*, *controllability* and whether this *Event* is a failure or not. If the *Event* were defined as a failure, the type of the failure should be specified.

The transformation rules specify the mapping from the annotated BPEL metamodel of Figure 4 to the model elements of Figure 5. The *State* model element of BPEL is mapped into the *State* in Deterministic Automaton model. Activities such as *Invoke*, *Receive*, *Reply* and *Assign* are mapped into a combination of Deterministic Automaton *Transition* and *Event*. For example consider an *Invoke* activity, the transformation make use of the current state (Invoke.currentState) and the next state (Invoke.nextState) of the *Invoke* activity to create the source (Transition.source) and the target (Transition.target) of a created transition.

As denoted in Figure 5 the *Transition* may be *Trigger*ed by an *Event*. At the destination, such an event must be created. Then, the attributes *isObservable* and *isControllable* must be assigned to the correct value. For example, in case of *Invoke* these attributes can be set according to the values of *Invoke.isObservable* and *Invoke.IsControllable*. If a BPEL activity is consider as a failure, the failure type attribute (typeFailure) is transformed to a *FailureType* associated to the corresponding *Event*.

Samples of the transformation specification are included in [15].

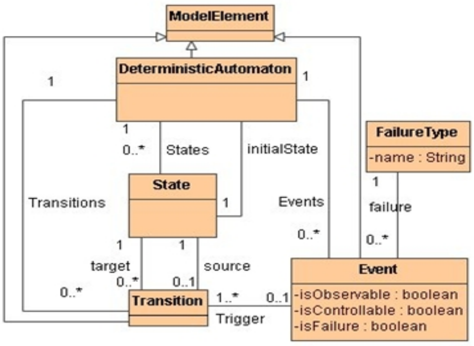


Figure. 5. Metamodels of Deterministic Automaton

Example of Transformation from BPEL to Deterministic Automaton: Figure 6 represents the Deterministic Automata created as result of applying our transformation approach on the annotated BPEL model of the Customer Technical Support example shown in Figure 3. Consider *CheckCustomerAccount* which is an *Invoke* activity. It can be seen that *currentState* of this Invoke activity is "CUS1" and its *nextState* is "CUS2". As a result, in Figure 6 the model transformation has created a transition from CUS1 to CUS2 marked by *CheckCustomerAccount*.

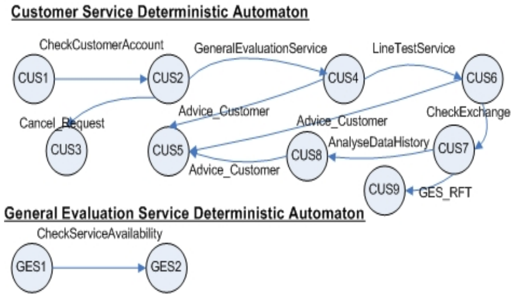


Figure. 6. Created Deterministic Automata.

4.3. Transformation of Diagnoser Automaton to Diagnosing Service (Diag2BPEL)

After performing the model transformation on a BPEL model a Deterministic Automaton is produced. Because the system may be express in more than one BPEL model, as for example in our running example, the transformation produces more than one Deterministic Automaton, see Figure 6. The overall behaviour of the system is captured by the parallel composition of created Automaton. For information one parallel composition see [7]. From a parallel composition of the Deterministic Automata with Unobservable Events, it is possible to create a single automaton with equivalent behaviour [7]. The second transformation (Diag2BPEL) maps the automaton into a BPEL model *Diagnosing Service*.

Figure 7 depicts the Diagnosing Automaton automatically created from the Deterministic Automata of Figure 6 via the UMDES tool [14]. The Diagnoser Automaton represents all the possible states which can be reached after the execution of an event. For example, (CUS7,GES2 N, CUS9,GES2 F1) represents two states which may be created as a result of the execution of *CheckServiceAvailability*. Firstly, the service *Customer Service* is at state "CUS7" and the service *General Evaluation Service (GES)* is at state "GES2" see 4.1(1). This is a *normal* state marked by N. Secondly, the service *Customer Service* is at state "CUS9" and the service *General Evaluation Service (GES)* is at state "GES2" which is a failure of type 1.

The transformation from Diagnoser Automaton to BPEL is very similar to the mapping described in section 4.2. Due to space restriction, we have included a fragment of the *Diagnosing Service* in Figure 8. Element of Figure 7 marked with (*) are transformed and included in Figure 8. Full Diagnoser Automaton is available at [15].

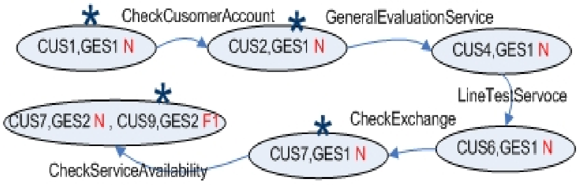


Figure. 7. A Fragment of Diagnoser Automaton.

5. Discussion and related work

Yan et al. [2] formalize BPEL Web service model as Discrete Event System (DES). In [18], Yan and Dague propose a Model-Based approach to diagnosing of behaviour of Web services by extracting synchronized automata from the BPEL. The synchronized automata are used to identify the dependency between the variables and to identify the trajectories following the detection of the exception. Our approach differs from [18] in various ways. Firstly, we make use of MDD to automatically generate the Diagnoser. Secondly, using MDD allows us to reuse existing results in DES [3] and UMDES tool [14] which reduces the cost of implementation. Our approach can deal with a wide range of failure including the type of failure which is discussed in [18]. It seems that the approach presented in [2] can not handle failure such as Right-First-Time. Finally, our approach fundamentally differs from the above as our Diagnoser is modelled in Web services languages.

In this paper, variants of automata are used to represent Discrete Event Systems. Our approach is independent of such reorientation. Petri nets are another formalism used in diagnosability [1, 4]. Considering the wide adoption of Petri nets for workflow modelling, there is a large scope for using Petri net as formalism in this context. This is a direction for future research.

A centralized Diagnoser may result in bottlenecks affecting the performance. Various decentralized diagnosing scheme have been proposed to address this issue [19, 20]. A decentralized diagnosing method generates one Diagnoser per each module of the system. Applying the method represented in this paper along with decentralized diagnosing approach result in a *Diagnosing Service* for each service which is expected to result in better performance. These *Diagnosing Service*s can collaborate with each other to fulfil the task of centralized Diagnoser. We are currently extending our tool set to implement a Decentralized approach.

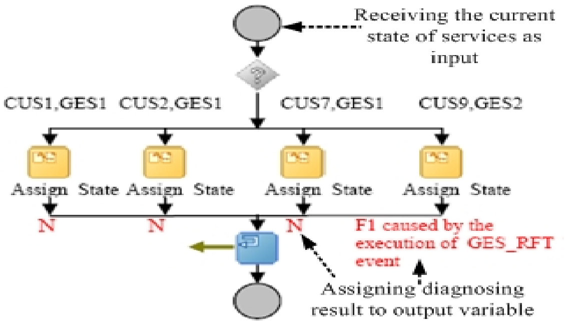

Receiving the current state of services as input
CUS1,GES1
CUS2,GES1
CUS7,GES1
CUS9,GES2
Assign State
Assign State
Assign State
Assign State
N
N
N
F1 caused by the execution of GES_RFT event
Assigning diagnosing result to output variable

Figure. 8. A Fragment of the Diagnoser Service.

6. Conclusion

This paper presents a Model Driven Development approach to the design and implementation of Diagnosers for a group of interacting services. The underlying idea is to apply Discrete Event System techniques to produce a *Diagnosing Service*, which will monitor the services. MDD is used to transform models of Services, captured in BPEL, into Deterministic Automata with Unobservable Events. Using DES algorithms, a Diagnoser Automaton for the Deterministic Automaton is created. MDD model transformations map the Diagnoser Automaton to produce the *Diagnosing Service*. The presented approach is implemented as an Oracle JDeveloper plug-in and has been applied to a case study involving the monitoring of a Customer Service application to identify Right-first-time failures.

8. References

[1] Y. Wang, T. Kelly, and S. Lafortune, "Discrete control for safe execution of IT automation workflows," in *EuroSys*, 2007.

[2] Y. Yan, Y. Pencole, M.-O. Cordier, and A. Grastien, "Monitoring Web Service Networks in a Model-based Approach," in *ECOWS05*, Sweden, 2005.

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

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

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

[6] T. Stahl and M. Volter, *Model Driven Software Development; technology engineering management*: Wiley, 2006.

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

[8] F. Lin, "Diagnosability of discrete event systems and its applications " *Discrete Event Dynamic Systems,* vol. 4, 1994.

[9] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. C. Teneketzis, "Failure diagnosis using discrete-event models," *IEEE Trans. on Control Systems Technology,* vol. 4, pp. 105-124, 1996.

[10] http://www.openarchitectureware.org

[11] D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier, "SiTra: Simple Transformations in Java," in *MoDELS*, 2006, pp. 351-364.

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

[13] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "XML Schema Part 1: Structures," W3C 2004.

[14] L. Ricker, S. Lafortune, and S. Genc, "DESUMA: A Tool Integrating GIDDES and UMDES," in *WODES*, 2006.

[15] www.cs.bham.ac.uk/~bxb/Alodib/RFTC.html

[16] B. Bordbar and A. Staikopoulos, "On Behavioural Model Transformation in Web Services," in *eCOMO*, China, 2004.

[17] UML2.0, "UML 2.0 Superstructure Specification, www.omg.com," 2004.

[18] Y. Yan and P. Dague, "Modelling and Diagnosing OrchestratedWeb Service Processes," in *ICWS*, 2007.

[19] Y. Wang, T.-S. Yoo, and S. Lafortune, "Diagnosis of Discrete Event Systems Using Decentralized Architectures" *Discrete Event Dynamic Systems,* vol. 17, 2007.

[20] S. Genc and S. Lafortune, "Distributed Diagnosis of Place-Bordered Petri Nets," *IEEE Transactions on Automation Science and Engineering* vol. 4, pp. 206-219, 2007.