On automated generation of Diagnosers in Fault tolerant Service oriented Architectures

Mohammed Alodib¹, Behzad Bordbar¹, Basim Majeed²
¹School of computer Science, University of Birmingham, UK
(M.I.Alodib,B.Bordbar)@cs.bham.ac.uk
²British 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 the Diagnoser, which monitors the system’s behavior 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 the MDD transformations. Then, relying on Discrete Event System techniques Diagnosers Automation for the Deterministic Automaton representations are created automatically. Finally, the Diagnosers Automaton is transformed into a new BPEL representation, which is integrated into the original architecture. The proposed approach is implemented as an Oracle JDeveloper plugin. To evaluate the presented method, a case study involving a Right-First-Time failure scenario motivated by telecom application is used.

Subject Categories and Descriptors
C.1.4 [Parallel Architectures]; C.2.4 [Performance and Reliability]; Reliability, Testing, and Fault-Tolerance

General Terms
Service Oriented Architecture, Fault tolerant systems, Discrete event systems

Keywords: Oracle JDeveloper, Web Services, Business Process Execution Language

Received on 12 January 2009; Revised 19 March 2009; Accepted 10 May 2009

1. Introduction

One of the crucial steps in building fault tolerant Service oriented Architectures (SoA) is to diagnose the occurrence of failure automatically. This is often achieved by the creation of the Diagnoser which allows monitoring of services and interactions between them to identify an occurrence of 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]. DES community mostly uses representations such as automata [3] or Petri net [4] for the modeling of the system and the Diagnoser. On the other hand, SOA makes use of languages such as BPML and BPEL [5] for the modeling of the system. There is a clear need for adopting methods used in DES and applying them to the SOA.

Model Driven Development (MDD) [6] promotes the role of modeling and automated modeled generation to bridge the gap between technical spaces [7]. This paper harnesses the capability of MDD to automatically generate Diagnosing Services using DES methods. A Diagnosing Service can be implemented as BPEL representation and interacts with the existing services within the architecture to identify occurrence of the failure. The paper also presents an outline of a tool developed, as an Oracle JDeveloper plugin, 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 Automaton. Then, applying DES techniques produces an Observer Automaton, which is used to generate the Diagnosing Service. The approach is applied to a case study, which is based on a scenario involving a Service-based Customer Support System for telecommunication applications. The aim is to design a monitor to identify Right-First-Time failures, in which the Customer Support System fails to complete a task First-Time and is forced to repeat part of the task again. This type of failure may cause extra costs and delays in the completion of the tasks, causing a violation 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 illustrates the architecture of the tool developed on the basis of the presented approach. Section 6 discusses the related work and section 7 includes the concluding remarks.

2. Preliminaries

This section describes introductory notions used in this paper. Firstly, a brief description of diagnosability of Discrete Event Systems (DES) will be explained. Secondly, Model Driven Architecture will be discussed. Finally, a brief review of Web services and Business Process Execution Language (BPEL) will be presented.

2.1 Diagnosability of Discrete-Event System

A Discrete Event System (DES) is a discrete-state, event-driven system whose state depends on the occurrence of asynchronous discrete events over time [8]. As result, DES embodies a wide range of application domain including Web services [1, 2]. There are a variety of languages used for capturing DES models such as variants of automata and Petri net [8]. 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 will be used in this approach [3]. A Deterministic Automaton with Unobservable Events is a four tuple \(G=(X, Z, \delta, x_0)\), where \(X\) is a finite set
of states, \(\Sigma\) denotes a set of events, \(\delta \subset X \times \Sigma \times X\) represents the transition between the states and \(x_i \in X\) is called the initial state. Some of the events in a DES are observable, for example output of sensor or 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. The set of observable/unobservable events is denoted by \(\Sigma_o/\Sigma_u\) respectively. As result, \(\Sigma = \Sigma_o \cup \Sigma_u\). The set of events which represent the occurrence of failure is denoted by \(\Sigma_f\). Since a failure is unobservable, i.e. \(\Sigma_f \subset \Sigma_u\). For the purpose of brevity we will sometimes write "Deterministic Automaton" instead of "Deterministic Automaton with Unobservable Events".

The purpose of the diagnosis is to use a model of the system, which is for example captured in Deterministic Automaton, to identify the occurrence of failure. Since a failure is 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 behavior in order to reduce the uncertainty [8]. 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, 9]. From the Observer a new Finite State Machine, called the Diagnoser Automaton is created which is used to achieve the diagnosis when it observes the behavior of the system. A Diagnoser Automaton is modeled as \(G_f = (Q_f, \Sigma_f, \delta_f, q_0)\) where \(Q_f\) is the subset of the observable state which includes all the states which can be reached from the initial state under a specific transition \(\delta_f\) [10]. Each state in \(Q_f\) 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 \(F_1, F_2, \ldots, F_f\) to clarify what type of failure has happened. For example, the initial state is declared to be \(\{(x_0,\{N\})\}\) which means that the behavior of the system is normal in state \(x_0\) but for example, \(\{(x_i,\{F_1\})\}\) means that the system is in state \(x_i\) and a failure of type "1" has occurred [3, 11]. 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 [8]. For further information about DES and algorithms for creating the Diagnosers automaton we refer the reader to [3, 12, 13].

2.2 Model Driven Architecture MDA

The method adopted in this paper relies on Model Driven Architecture (MDA) [14] 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 [15]. In the MDA 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 Kermeta [16], OpenArchitectureWare [17] and SiTra [18].

2.3 Service Oriented Architecture and Web services

There is an ever-increasing pressure on modern enterprises to adapt to the changes in their environment by evolving to respond to any opportunity or threat [19]. 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 SOAs. The interaction between services in this paper is captured via Business Process Execution Language (BPEL) [20]. 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) [21], usually in WSDL file [5]. For further information about Web services, we refereee the reader to [5].

3 Example: Diagnosing Right-First-Time failure in services

This section is the outline of a running example which will be used in the rest of the paper. The example is based on a scenario involving a simplified interaction between a customer and a number of services provided by a typical Telecommunication Company. The services aim at providing technical support for the customers' Broadband connection.

As depicted in Figure 1, the customer logs 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 problem and a separate process is invoked to deal with this problem (not shown as part of this example). If everything is fine on the exchange side, the Customer Services sends a request to Line Test Service (LTS). This is an automated service to check line status up to the customer premises, but can also indicate problems on the exchange side which were not detected by the GES. As a result the outcome to the check is one of the three possible cases 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, which is shown in bold arrow in Figure 1, is reached only if the LTS has the ability of checking if its exchange functioning correctly. Notice, the exchange is carried out independently from the GES. As a result if the case

1This is an imaginary example, real life scenarios and processes can differ substantially.
2We assume that the problem the Customer can log into the company’s website, for example suppose the customer is not happy with the speed of his Broadband connection.
In our approach, as depicted in Fig. 2, BPEL representations should be annotated by identifying the observable and unobservable events. A similar approach is adopted by Yan et al. [2] for annotating BPEL files. Then, a model transformation (BEPL2FSM) will be used to transform the annotated BPEL models automatically to a Deterministic Automaton. Next, applying classical theories of diagnosability [3] a Diagnoser will be computed and created, this is denoted by the arrow marked as Generating 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 responds with diagnosing result which describes the system behavior whether it is normal or a failure has occurred. If the system status had a failure, the Diagnoser should specify which event caused this failure.

BPEL model for the example of section 3: A real world Customer Support system may make use of a large number of services. Due to space restriction the scenario described in Section 3 consists of eight main activities which are marked by (*). The flow of activity 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 captured as XML Schema Definition (XSD). For example, CustomerServiceProcessRequest which represents input variable used to input the customerID (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 [22].

In the remaining of this section, the annotating BPEL representations will be explained. Then, the first model transformation which represents the transformation from annotating BPEL to a Deterministic Automaton will be described. Finally, the second model transformation process from the Diagnoser Automaton to a new BPEL representation will be illustrated.

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 occurrence of failure. Such information is not included in a BPEL file; a common practice is to annotate the BPEL file to include such information [2]. The rest of this section, present outline of our method of annotation of BPEL files.

Including information related to the 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.

Figure 2. Applying MDA to the design of Diagnoser

Figure 3. BPEL representations of Customer Support system example
Annotating BPEL to include information about the actions: BPEL activities such as Invoke, Reply, Receive and Assign change the state of the systems. To identify if such activities are observable or controllable the BPEL file is annotated. Seven of the main activities captured in Figure 3(i) are observable and controllable: Check Customer Account, Cancel Request, General Evaluation Service, Line Test Service, Check Line Exchange, Analyses Data History, and Advice Customer. For example, Check Customer Account (CheckCustomerAccount) is annotated in following snippet of code. It can be seen that "c" is used to indicate that the action CheckCustomerAccount is controllable and "o" to indicate that it is observable.

In the above XML code, CheckCustomerAccount is interacting with Customer Process service. This is denoted by partnerLinks. The portType which represents the interface of the web service is also declared. The interaction involves execution of an operation of checkCustomerAccount of the service.

Identifying the activities corresponding to failures: Since a failure is uncontrollable and unobservable, annotating such an activity is similar to annotating an observable event. Except in the case of failures, two further attributes used to declare that the activity is indeed a failure and also to represent the type of the failure is required. In the BPEL model of Figure 3(i), General Evaluation Service Right First Time (GES_RFT) is a failure. This type of failure is a Right-First-Time failure, i.e. it occurs only when the Line Test Service checks the Line Exchange, and if it is not ok, the general evaluation service must be performed again, as described in section 3. The following snippet of XML code represents the annotation of the part of XSD of GES_RFT. It can be seen that the activity is uncontrollable (uc) and unobservable (uo). The type of failure is declared as type 1.

4.2 Transformation from BPEL to Deterministic Automaton

After annotating the BPEL model, the transformation from BPEL into the Deterministic Automaton can be automated. 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. To include the meta-elements related to the annotations, the metamodel of [23] is extended. The added elements, which correspond to the annotations, are marked with (*) in Figure 4. In the rest of this section, samples of the meta-elements represented in Figure 4 are explained briefly. The main attributes related to a BPEL model are included in Process metamodel-element. Partner Links identify how two parties such as web services can interact with each other and what processes are offered by each party relying on their role and their port type. A port type represents the interface of web services in the WSDL file [24]. Switch model-elements support conditional behavior of the process. Receive starts the BPEL process and sometimes is used to perform callbacks to other services. A Reply returns the response to a synchronous BPEL process. Invoke executes an operation on another service on behalf of a given service. Executions are either synchronous request/response or an asynchronous one-way operation. Assign allocates the value of a variable into another variable. For more detail on BPEL can be found in [20, 25, 26].

The metamodel of Figure 4 includes information regarding the annotations which are marked by (*) in the picture. 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, is Failure and typeFailure.

Figure 5(i) represents a metamodel for Deterministic Automaton with Unobservable events, which is based on [27]. It can be seen that a number of states, which with the help of Transitions are connected to each other. Each Transition between two States is Triggered 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 failure, the type of this failure should be categorized.

Figure 5(ii) also represents the metamodel for the Diagnosis Automaton which is an extension of the Deterministic Automaton metamodel. The Diagnosis Automaton metamodel has two more meta-elements which are StateDetail and StatusType. As mentioned in section 2.1, each Diagnosis State has a subset of the observable states represented by StateDetail in Figure 5(ii). For example, \{(x1,N)\} is a system state, where the state detail is at x1 and the status type is (N).
Then, the attributes are transformed to a specific activity. If a failure occurs, the failure type attribute (typeFailure) is set accordingly. The value of these attributes can be set according to the values of the next state (Invoke.nextState) of the activity. For example, consider an Invoke activity, the transformation makes 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(i) the Transition may be Triggered 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 the value of these attributes can be set according to the values of Invoke.isObservable and Invoke.IsControllable. If a BPEL activity is a failure, the failure type attribute (typeFailure) is transformed to a FailureType associated to the corresponding Event. The following snippet of code describes such a transformation:

```
Transformation Invoke2FailureEvent(BPEL, DeterministicAutomaton)
params -- none
source
  invoke : BPEL::Invoke;
target
  event : DeterministicAutomaton::Event;
source condition
  invoke.isFailure=true;
target condition
  event.isObservable =false and event.IsControllable =false;
unidirectional:
  mapping
    invoke.typeFailure <-> event.failure;
```

4.2.1 Transformation rules for mapping BPEL to Deterministic Automaton

The transformation rules specify the mapping from the annotated BPEL metamodel of Figure 4 to the model elements of Figure 5(i). The State model element of BPEL is naturally 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(i) the Transition may be Triggered 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 the value of these attributes can be set according to the values of Invoke.isObservable and Invoke.IsControllable. If a BPEL activity is a failure, the failure type attribute (typeFailure) is transformed to a FailureType associated to the corresponding Event. The following snippet of code describes such a transformation:

```
Transformation Invoke2FailureEvent(BPEL, DeterministicAutomaton)
params -- none
source
  invoke : BPEL::Invoke;
target
  event : DeterministicAutomaton::Event;
source condition
  invoke.isFailure=true;
target condition
  event.isObservable =false and event.IsControllable =false;
unidirectional:
  mapping
    invoke.typeFailure <-> event.failure;
```

Example of Transformation from BPEL to Deterministic Automaton: Figure 6 represents the Deterministic Automata created as result of applying our transformation to the annotated BPEL model of the Customer Technical Support example shown in Figure 3. Consider CheckCustomerAccount which is an Invoke activity. The XML code corresponding to CheckCustomerAccount is presented in snippet of code 4.1.(2). 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.

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, for example in our running example, the transformation produces more than one Deterministic Automaton, see Figure 6. The overall behavior of the system is captured by the parallel composition of created Automaton. For information one parallel composition see [8]. From a parallel composition of the Deterministic Automata with Unobservable Events, it is possible to create a single automaton with equivalent behavior [8]. The second transformation (Diag2BPEL) maps the automaton into a BPEL model which we refer to as the Diagnosing Service. Next, the outline of transformation that creates Diagnosing Service will be described.

4.3.1 Transformation rules for mapping Diagnoser Automaton to BPEL

A Diagnosing Service monitors the behavior of the services to identify if they are in a normal state or, if a failure has occurred. As a result, the Diagnosing Service includes conditional statements in form of BPEL Switch activity with multiple Cases, see metamodel of Figure 4. Each Case in the Switch activity evaluates the current state of the system services and assigns “N” for a normal state and the type of failure if a failure has occurred. In case of a failure, the event which is causing the failure will be included and the type of failure will be assigned to an output variable representing the diagnosing result. To conduct this model transformation, every model-element State of the Diagnoser Automaton metamodel of Figure 5(ii) results in one of the Cases in the Switch. In each Case a BPEL Assign activity is created and StateDetail and StatusType, see Figure 5(ii), are used to determine if the state is normal State or a failure. Next we shall illustrate the transformation with help of an example.

Example of creating a Diagnosing Service: in the Customer Technical Support example, the generated Deterministic Automaton in section 4.2 are passed to UMDES tool to compute and generate the Diagnoser Automaton which depicted in Figure 7. 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 CustomerService is at state “CUS7” and the service GeneralServiceEvaluation (GES) is at state “GES2” see 4.1(1). This is a normal state marked by N. Secondly, the service CustomerService is at state “CUS9”
and the service General Service Evaluation (GES) is at state "GES2" which is a failure of type 1, see 4.1(3).

Due to space restriction, we have included a fragment of the Diagnosing Service in Figure 8. Element of Figure 7 which are transformed and included in Figure 8 are marked with (*) in Figure 7. As depicted in the Figure B Diagnosing Service receives the current state of services as input and presents the result of the diagnosis as output variable.

5. An implementation of the presented approach

We have implemented the presented approach as a Plugin for Oracle JDeveloper. The implementation follows the outline of the method as depicted in Fig. 2. The tool requires passing all annotated BPEL files and their XML Schema Definition (XSD) as inputs. Each BPEL file and its XSD are combined together to collect all required details, to transform the BPEL representation into the Deterministic Automaton. The first transformation (BPEL2FSM) is implemented via SiTra [18].

To create the Diagnoser Automaton the diagnosability algorithms are applied, which is implemented in various tools GID-DES and UMDES-LIB [28]. In our implementation, UMDES-LIB is used, which creates a Diagnoser Automaton from a given Deterministic Automaton. Finally, the created Diagnoser Automaton is transformed into a BPEL representation by using the second transformation method (Diag2BPEL) which is also implemented via SiTra. The models used in the case and all samples of code are available at [22].

6. Discussion and related work

Yan et al. [2] formalize BPEL Web service model as Discrete Event System (DES). In [29], Yan and Dague propose a Model-Based approach to diagnosing of behavior of Web services by extracting synchronized automatons from the BPEL. The synchronized automatons are used to identify the dependency between the variables and to identify the trajectories following the detection of the exception. Our approach differs from [29] 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 and tools [28] reducing the cost of implementation. Our approach can deal with a wide range of failure including the type of failure which is discussed in [29]. 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 are modeled in Web services languages.

Wang et al [1] have applied DES control theory to allow safe execution of flawed workflows by avoiding runtime failure. Their approach makes use of Automaton to identify forbidden states, representing in desirable execution state, to generate the control logic. Hence, the suggested procedure includes Diagnosers of the failure. Our approach produces a separate Diagnoser which can be used in conjunction with any controller.

In this paper, variants of automata are used to represent Discrete Event Systems. Petri nets are another formalism used in diagnosability [1, 4, 30]. Considering the wide adoption of Petri nets for workflow modeling, 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[31, 32]. 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 Services can collaborate with each other to fulfill the task of centralized Diagnoser. We are currently extending our tool set to implement a Decentralized approach.

7. 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 plugin and has been applied and evaluated to a case study involving the monitoring of a Customer Service application to identify Right-first-time failures.

References


Figure 7. Diagnoser Automaton

Figure 8. A fragment of BPEL model for Diagnoser Service

ECOWS05

Journal of Digital Information Management ❖ Volume 7 Number 6 ❖ December 2009 341


