A Model Driven Architecture approach to fault tolerance in Service Oriented Architectures, a performance study

Mohammed Alodib, Behzad Bordbar
School of computer Science, University of Birmingham, UK
M.I.Alodib,B.Bordbar@cs.bham.ac.uk

Abstract

In modern Service oriented Architectures (SoA) identifying the occurrences of failure is a crucial task, which can be carried out by the creation of Diagnosers to monitor the behavior of the system. Model Driven Architecture (MDA) can be used to automatically create Diagnosers and to integrate them into the system to identify if a failure has occurred. There are different methods of incorporating a Diagnoser into a group of interacting services. One option is to modify the BPEL file representing services to incorporate the Diagnoser. Another option is to implement the Diagnoser as a separate service which interacts with the existing services. Moreover, the interaction between the Diagnoser and the services can be either Orchestration or Choreography. As result, there are four options for the implementation of the Diagnoser into the SoA via MDA. This paper reports on an Oracle JDeveloper Plugin tool developed which applies MDA to create these four possible implementations and compares the performance of them with the help of a case study.

1. Introduction

Modern Service oriented Architectures (SoA) for mission critical system must tolerate occurrences of failure. In particular, one of the important tasks in the design of such systems is the detection of the occurrence of failure automatically. This can be achieved by monitoring the interaction between the services to identify if the failure has occurred [1, 2]. The software entity that conducts the monitoring is commonly referred to as a *Diagnoser*. The approach adopted in this paper, applies Model Driven Architecture (MDA) techniques to benefit from diagnosability theories in Discrete Event System (DES) [3] for designing the Diagnoser in SoA. Consider a number of services, which are represented by BPEL [4] models describing the behavior of the services and their interactions. These models are automatically transformed to Deterministic Automaton [5] via an MDA [6] transformation. Then, the created Deterministic Automaton are used to develop a Diagnoser and Observer following the DES techniques [3]. Finally, the Diagnoser is transformed via another MDA transformation to produce a SoA Diagnoser for the monitor the group of services.

This paper studies various methods of incorporating a Diagnoser into the interacting services. There is a choice to implement the Diagnoser as a BPEL service or separate dedicated Web service. Moreover, the interaction between the group of services and the Diagnoser can be an Orchestration or a Choreography [7]. This amounts to four styles of creating the Diagnoser. This paper reports on tool extending Oracle JDeveloper which produces all four types of Diagnoser and studies them from the performance point of view. Our case study demonstrates that a dedicated service provides a better performance, while switch from Orchestration to Choreography make slightly different. The proposed methods have been evaluated with the help of *stress testing* facilities provided by the Oracle Application Server [7].

The paper is organized as follows. Section 2 reviews the preliminary material used in the rest of the paper. Section 3 briefly describes the outline of our method of applying MDA to generate the Diagnoser. Extended description of the method can be found in an earlier paper [8]; the current paper extends [8] by presenting four separate styles of implementation and their comparison. Section 4 presents an outline of a running example, which will be used in the rest of the paper. Different styles of incorporating the Diagnoser methods are explained in section 5. The results related to the comparison are discussed in section 6. Finally section 8 includes the concluding remarks.

2. Preliminaries

This section describes introductory notions used in this paper. Firstly, a short introduction on the theory of Diagnosability in Discrete Event Systems (DES) will be presented. Secondly, an outline of Model Driven Architecture will be discussed. Finally, a brief review of Service Oriented Architecture and Web services will be presented.

2.1. Diagnosability of Discrete-Event System

A Discrete Event System (DES) is a *discreet-state*, *event-driven* system whose state depends on the occurrence of asynchronous discrete events over time [5]. There are a variety of languages used for capturing DES models such as variants of automata and Petri net. A variant of Deterministic Automaton known as Deterministic Automaton with Unobservable Events is used for modeling the services [3]. A Deterministic Automaton with Unobservable Events 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 and $x_0 \in X$ 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 detonated by Σ_o /Σ_{uo} respectively. As result, $\Sigma=\Sigma_o \cup \Sigma_{uo}$. The set of events which represent the occurrence of the failure is denoted by Σ_f. Without any loss of generality it can be assumed that failure events are unobservable, i.e. $\Sigma_f \subseteq \Sigma_{uo}$.

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 cannot 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 [5]. To achieve this, from a Deterministic Automaton, a new model called an *Observer Automaton* is created, which 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. A Diagnoser Automaton is modeled as G_d= (Q_d , Σ_o , δ_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. 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. For example, an initial state is often labeled as $\{(x_0,\{N\})\}$ which means when the system is in state x_0, its behavior is *Normal*, but for example, $\{(x_1,\{F_1\})\}$ means that when the system state is in x_1, a failure of type "1" has occurred [3, 10]. Hence a Diagnoser is produced for two main reasons [3]: i) online detection and isolation of failure ("Did a fault happen or not?", "What type of fault happened?"), and ii) offline verification of diagnosability properties of the system.

2.2. Model Driven Architecture MDA

The methods adopted in this paper relies on Model Driven Architecture (MDA) [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 and models that can be created in the language [11]. 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 [12] , OpenArchitectureWare [13] and SiTra [14].

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 [15]. Web services [16] relies on using a well-accepted standards and XML languages used for communication by interchanging message with the help of an interactive interface such as the Web service Description Language (WSDL) which is an XML language used to define the message formats, data-types and transport protocols [17]. Web services interaction can be created as orchestration or choreography architectures [16]. Orchestration relies on a central process which coordinates the invocation of different Web services. However, each Web service knows when to execute its operation and with whom to

interact in choreography architecture without using a central coordinator. The interaction between services in this paper is captured via Business Process Execution Language (BPEL) [7]. BPEL can be used to express complex sequential, parallel, iterative and conditional interactions.

3. A model driven approach to Diagnosability in SoA.

Figure 1 depicts the outline of our method of generating the Diagnoser via the MDA. Consider a number of services which interact with each other. The behavior of these services and their interaction are captured by a number of BPEL files. First the services are annotated to identify the observable and unobservable events. This is similar to the method adopted by Yan et al.[2]. In Figure 1 this is denoted as "Annotated BPEL", which is transformed automatically to a Deterministic Automaton via the first model transformation BPEL2FSM. Then, algorithms from diagnosability in DES [3] are applied to the Deterministic Automaton to compute and generate the Diagnoser Automaton. Following that, the generated Diagnoser Automaton is transformed to a new BPEL representation addressing the *Diagnosing Service* for the original BPEL models via the model transformation Diag2BPEL. The *Diagnosing Service* is designed to receive the current states of the system as inputs. Then, it diagnoses the system status by determining whether the system behavior is in *Normal* state or a failure has occurred. In case of a failure, the *Diagnosing Service* specifies the event that has caused the failure.

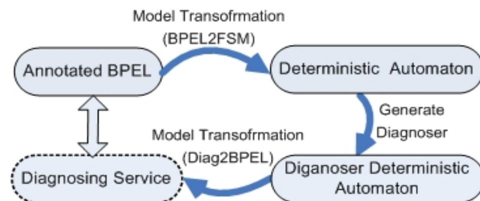

Figure 1. Applying MDA to the design of Diagnoser.

A preliminary implementation of the above approach is described in [8] as an Oracle JDeveloper Plugin. The tool is designed to receive system's annotated BPEL files and their XML Schema Definition (XSD) as inputs which are combined together to collect all required details related to apply the DES method which has been done with the help of UMDES tool [18] and performing the transformations methods with the help of SiTra [14].

This paper is different from [8] in two main ways. Firstly, the transformation in [8] only produces the Diagnoser as a BPEL file. Here, we report on our research on produce three other types of Diagnosers, see section 5 below. Secondly this paper focuses on comparing the four implementations in term of performance.

The tool has been studied with the help of a case study involving the monitoring of a Customer Service application, 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).

4. Example: Diagnosing Right-First-Time failure in services

The following example is based on a scenario[1] 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.

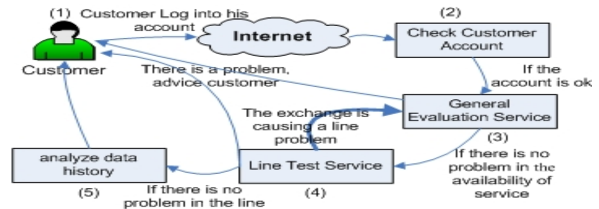

Figure 2. An overview of the interaction of a Customer

As depicted in Figure 2, the customer logs[2] 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

---

[1] This is an imaginary example, real life scenarios and processes can differ substantially.

[2] We 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.

services at the exchange side, the customer is informed of the status and a separate process is invoked to deal with this problem (not shown as part of this example). If everything is fine on the exchange side, the Customer Services sends a request to Line Test Service (LTS). 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 2, 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 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).

5. Incorporating the Diagnosing service.

By applying the above approach, the *Diagnosing Service* is generated automatically. There are four methods to incorporate the generated *Diagnosing Service* into a group of interacting services. These methods are explained as follows:

Method 1: This method is based on generating the *Diagnosing Service* as a BPEL file, which can collaborate with existing services to fulfill the diagnosing task. This method requires an invocation of the *Diagnosing Service* after each BPEL activity, which may change the state of the system. For example, consider the running example of section 4 with the BPEL representations which are *Customer Service* and *General Evaluation Service*. Assume an *Invoke* activity in *Customer Service* tends to invoke *General Evaluation Service* (*GES*). Let *invoke_GES* denotes an *Invoke* activity in *Customer Service* which is used to invoke *GES*. When *invoke_GES* activity invoked *GES*, the invocation result and the current state of *GES* are returned to the *invoke_GES* in *Customer Service*. Although the *invoke_GES* has received the invocation result from *GES*, it dose not know if a failure has occurred during the invocation of *GES*. To know such information regarding to failures, *Customer Service* can interact with *Diagnosing Service* to determine the system behavior after the invocation by using the two current states of *Customer Service* and *GES*. To do so, the *Diagnosing Service* is incorporated by adding a new *Invoke* activity after *invoke_GES* activity. The purpose of this new added *Invoke* activity is to interact with the *Diagnosing Service*.

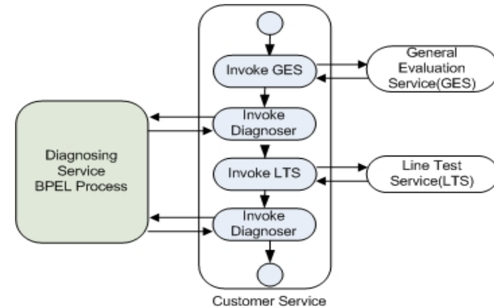

Figure 3. Example of method 1.

Figure 3 represents an example of services interaction between the *Diagnosing Service* and three services, which are *Customer Service*, *GES* and *Line Test Service*. It can be seen that the interaction between services is built as Choreography architecture.

Method 2: This method is an extension of method 1. The idea of this method is based on producing the *Diagnosing Service* as BPEL service with a new service called a *Protocol Service* used to control all interactions between existing services and the created *Diagnosing Service*. In method 1, there is no *Protocol Services*; hence the interaction is conducted by including extra Invoke activities. In contrast to method 1, in method 2 all interacting services should send their request to the *Protocol Service* performing the invocation of the destination service which returns the invocation result and its current state to the *Protocol Service*. To determine the system behavior after the invocation, the *Protocol Service* interacts with *Diagnosing Service* to perform the diagnosing tasks. Then, the diagnosing result will be provided to the *Protocol Service* which returns this result with the invocation results received from the destination service to the invoker service (source service). Figure 4 illustrates an example of the interaction between the *Diagnosing Service* and three services of the running example discussed in section 4, these services are *Customer Service*, *General Evaluation Service (GES)* and *Line Test Service* (*LTS*). It can be seen that the interaction in this method is based on the Orchestration architecture.

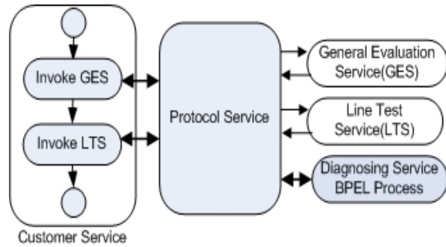

Figure 4. Example of method 2.

Method 3: This method automatically produces the *Diagnosing Service* as a stand- alone Web service interacting with a group of BPEL services. The generated *Diagnosing Service* is incorporated in the same manner of Method 1. However, Method 1 produces a BPEL file, while here a full-fledge *Diagnosing Service* will be created. Similar to method 1 this method is based on the Choreography.

Method 4: This method combines Method 2 and Method 3 to automatically generate a *Diagnosing Service* as a Web service, which interacts with other services through a *Protocol Service*. The interaction between the services is coordinated by the *Protocol Service*; hence method 4 follows the Orchestration architecture.

The *Protocol Service* is generated as a BPEL Service carrying out two important tasks. Firstly, it manages the interaction between the groups of services and ensures that the invocation result is returned to the invoker service. Secondly, it monitors the behavior of the system by interacting with the generated *Diagnosing Service*. Thus, when a BPEL service interacts with another service, it should send the *Protocol Service* a request in order to communicate the details of the destination service and the current state of the source service. When the request is received by the *Protocol Service*, the destination service will be invoked. Consequently, the invocation results and the new state of destination service are returned back to the *Protocol Service*. Then, the *Protocol Service* interacts with the *Diagnosing Service* to determine the behavior of the system after the invocation.

We shall describe the *Protocol Service* with the help of an example as follows. Figure 5 represents a scenario involving a simple interaction between two services, *Customer Service* and *General Evaluation Service* (*GES*), adopted from the running example discussed in section 4. The interaction, which terminates in a failure, can be described as follows. Assume *Customer Service* invokes the *GES*. To perform this invocation, the following six steps (enumerated in the picture) are occurring. Firstly, *Customer Service* sends to the *Protocol Service* a request involving the information regarding to the invocation process, including i) the identity of the destination service, which is *GES* in this example, ii) the destination service's parameters inputs e.g. *Customer ID*, iii) the current state of the source service (*Customer Service*) which is *CUS9* in this example. The *Protocol Service* keeps the current state of the invoker in its record, while it invokes the *GES*. Then, the invocation result and the current state of *GES*, depicted as *GES2* in this example, are returned to the *Protocol Service.* The Protocol Service interacts with the *Diagnosing Service* to identify the behavior of the system. This interaction requires assigning the current state of the *Customer Service* and *GES* to the inputs of the *Diagnosing Service.* The diagnosing result in Figure 5 explains that during the invocation of *GES*, a failure of type "1" has occurred. Finally, the diagnosing result and the invocation result received from service GES are forwarded to the invoker service which is *Customer Service* in this example.

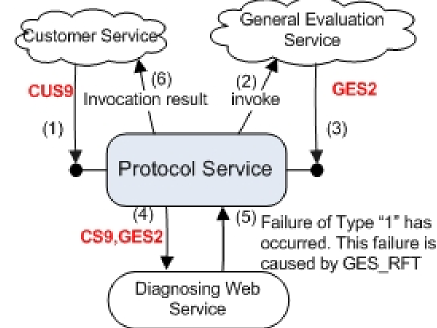

Figure 5. A scenario involving Protocol Service Interaction to identify a failure

5.1. The Diagnosing Service Transformation

To produce the *Diagnosing Service* and incorporate it into a group of service, the model transformation of [8] is modified to satisfy the new requirements. The transformation will be explained for Method 4, which the most elaborate of the four presented design. Method 4 relies on generating the *Diagnosing Service* as Web service interacting with a *Protocol Service*, the modifications basically has only affected the second model transformation (Diag2BPEL) of Figure 1. The outline of the second model transformation is depicted in Figure 6 which results in producing both the Diagnoser Web service and the Protocol Service. In [8], the transformation Diag2BPEL only produces the *Diagnosing Service* as BPEL file without any *Protocol Service*

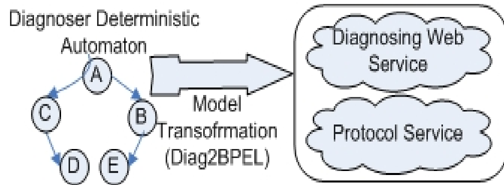

Figure 6. Model Transformation producing both the Diagnoser and the Protocol Service.

To define the model transformation Diag2BPEL two metamodels are required: metamodel of Diagnoser Automaton and metamodel of Web service. Figure 7 presents a simplified metamodel of the Diagnoser Automaton. Metamodels of Web Service are widely available and sometimes can be generated automatically [19]. Then, the transformation rules mapping from the Diagnoser Automaton metamodel to the Web service Metamodel must be created which will be briefly described here.

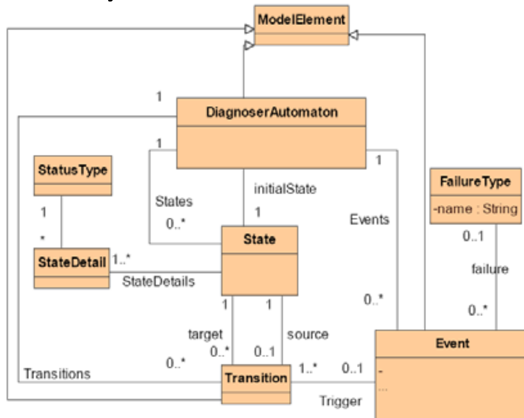

Figure 7. The Diagnoser Automaton metamodel

The Diagnoser model element is mapped into a Service which has an Operation in the Web service model. Then, the rest of the transformation process relies on Model-to-Text [6] transformation techniques used to generate the Java Code of the Service Operation. The Java code implements the behavior of the generated Diagnoser Automaton. The code of the Service is generated as follows. The *Diagnosing Service* may include conditional statements in form of if-then-else statements. Each such statement uses to evaluate the current state of the system services and returns "N" for a normal state or the information related to the occurrence of a failure. In case of a failure, the type of failure and the event which is caused the failure will be included in the diagnosing result. To conduct this model transformation, every model-element *State* of the Diagnoser Automaton metamodel, see Figure 7 is used to specify one of the Conditions in the if-then-else statement. *StateDetail* and *StatusType* of a Diagnoser Automaton model are used to determine if the State is in *Normal* status or a failure has occurred. Following the of DES [3], all failure events in the system should be categorized in a list according to their types. For example, in the running example discussed in section 4, *GES_RFT* is a failure event occurring when the *Line Test Service* indicates problems on the exchange side which were not detected by the GES. This failure forces GES to repeat its course of action violating Right-First-Time. Suppose that we categorize a violation of Right-First-Time as a failure of type "1". In this case, when the *Diagnosing Service* identifies the system status as F1, it means there is a failure of type "1" has occurred, and it has been caused by GES_RFT. Since the failures events are unobservable in the Diagnoser Automata, a method called *FindEventCausedFailure* is added to be used to look for the event which caused the failure. This method basically receives the status type of the system as inputs, and then it starts looking for the event of that type in the list of the categorization failures events. The following snippet of code represents the outline of transforming the Diagnoser Automata to Web service:

```
If (current_state=StateDetail) then
{
  If (StateDetail.StatusType="N") then
    Result_Diagnosing="The System Status is normal";
  else
    Result_Diagnosing="A failure of Type" +
    event.FailureType + " has occurred, this failure is
    caused by " +
    FindEventCausedFailure (StateDetail.StatusType).Name;
}
Event FindEventCausedFailure(StatusType statustype)
{
…\\code for Event FindEventCausedFailure
}
```

5.2. Automated generation of the Protocol Service

The *Protocol Service* is utilized as coordinator for a group of services in the system. It is designed to process the invocation requests from the service called the *source* of the invocation. Then, it transfers the request to another service called the *destination* of the invocation. Then, it interacts with *Diagnosing Service* to identify the behavior of the system. Finally, the invocation is carried out and the result of the diagnosis is returned to the source service. To automatically generate the Protocol Service, Diag2BPEL produces a BPEL service involving a *Switch* activity with multiple *Cases*. Each *Case* includes an *Invoke* activity to interact with one of the services. Then, the *Switch*

activity is followed by an *Invoke* activity added to invoke the *Diagnosing Service*. For example, consider the *Customer Service* and the *General Evaluation Service* of the running example discussed in section 4. The *Protocol Service* for these two services is represented in Figure 8. It can be seen that the process starts with a *Switch* activity involving two *Cases* (*Case 1* to invoke *Customer Service* and *Case 2* to invoke *General Evaluation Service*). As explained, the request is received from the source includes the destination details which are used to decide which Case should be followed (*Case 1* or *Case 2*). The *Invoke* activity which follows the *Switch* activity executes the *Diagnosing Service*.

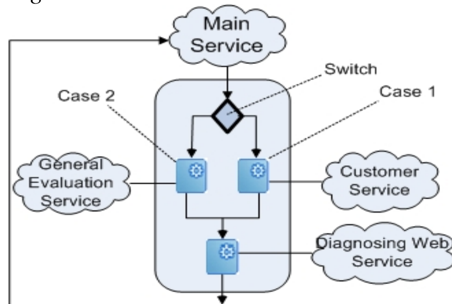

Figure 8. The Protocol Service Architecture.

6. Methods Comparisons.

The four presented methods have been tested and evaluated in terms of performance; a common practice of evaluating the performance is applying the *stress testing* which identifies and verifies the stability, capacity and the robustness of services [7, 20]. The stress testing relies on handling a large number of operations to the service to evaluate the performance in processing the received request. This test has been performed with the help of *Oracle Application Server*, which is used to deploy and to implement the BPEL representation. To carry out this test, there are some attributes, which should be specified before performing the test. These attributes are the number of the concurrent threads allocated to the process and the constant delay between each invocation.

In this paper the stress testing has been applied on each proposed method by handling a different number of concurrent threads representing the Customer. To be accurate, the for each number of threads the process of testing is repeated five times for each method, and then the average of the execution time has been calculated. To perform the stress testing, the running example discussed in section 4 has been created in all proposed methods. Then, these methods have been tested and evaluated in term of the performance. The result of this testing is depicted as line chart in Figure 9 which represents the execution time (in second) and the number of the concurrent threads. All the numerical values are available at [21].

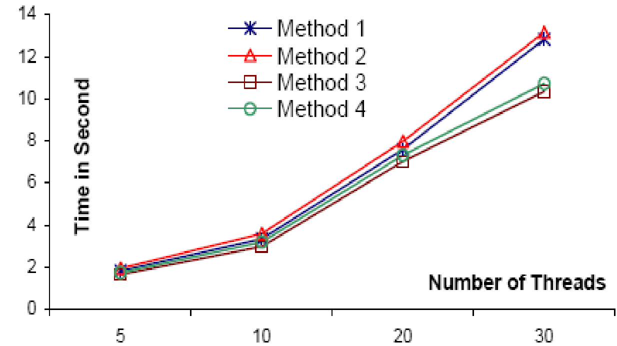

Figure 9. The stress testing result.

It can be seen that producing the *Diagnosing Service* as Web Service in Method 3 and 4 are faster than producing it as BPEL Service in Method 1 and 2. The percentage of the difference in processing 30 threads between the fastest method, which is Method 3, and the slowest method, which is Method 1, is approximately 0.024%, where Method 3 performs executing these threads within 10.35 second whereas the Method 1 takes 13.16 second. As a result, generating the Diagnoser as Web service increases the performance related to the interaction between services.

Generating the Diagnoser as Web Service is highly promising approach to enhance the efficiency of process execution and to maintain the system robustness. As discussed, Method 3 and 4 are the only two methods creating the Diagnoser as Web service. Figure 9 shows that the performance of methods 3 and 4 are very close to each other; Method 3 performs processing 30 concurrent threads within 10.35 second whereas Method 4 takes 10.73 second. The difference is negligible, but Method 4 has great advantages from the programming point of view. In particular, using a *Protocol Service* results in a modularized design. Moreover, the architecture of Method 4 is based on the Orchestration which is a more flexible paradigm offering the following advantages over the Choreography [7]: i) the coordination of component processes is centrally managed by a known coordinator, ii) Web services can be incorporated without being aware that they are taking part in a business process, iii) alternative scenarios can be put in place in case of a fault. The case study scenario, implementation of the four types Diagnosing services and the numerical value related to the experiments are all available at [21].

7. Discussion and related work

The formalizing BPEL models as Discrete Event System (DES) has been achieved by following the lead of Yan et al. [2]. Our approach differs from [2] in using MDA to automatically generate the Diagnoser as Web service interacting with a *Protocol Service* managing the interaction between the Diagnoser and the existing BPEL representations. In addition, there are four methods have been proposed to provide techniques for the interaction between a group of services and the Diagnoser.

In this paper, the Diagnoser is generated as a centralized service which may result in bottlenecks affecting the performance. Various decentralized diagnosing scheme have been proposed to address this issue [22, 23]. A decentralized diagnosing method generates one Diagnoser per each module of the system. We are currently extending our tool set to implement a Decentralized Diagnosers and to incorporate them into a group of services.

8. Conclusion

This paper presents a method using a chain of MDA model transformation to automatically produce Diagnosers for the monitoring of the behavior of the system to identify the occurrence of failure and the type of failure. Automated transformations are used to transfer BPEL models to Deterministic Automata. Relying on Discrete Event System techniques a Diagnoser for the Deterministic Automata is created. A second model transformation produces the Diagnosing Web service. The paper discusses various possible implementations of the Diagnosing service and reports on a case study of evaluating the performance of each implementation.

8. References

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

[2] Y. Yan, Y. Pencole, M.-O. Cordier, and A. Grastien, "Monitoring and Diagnosing Orchestrated Web Service Processes," in *ICWS07*, USA, July 9-13, 2007.

[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] IBM, BEA, Microsoft, SAP, and Siebel, "Business Process Execution Language for Web Services (BPEL4WS) Version 1.1," 2003.

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

[6] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture--Practice and Promise*: Addison-Wesley, 2003.

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

[8] M. Alodib, B. Bordbar, and B. Majeed, "A model driven approach to the design and implementing of fault tolerant Service Oriented Architectures " *submitted for publication*, 2008.

[9] C. M. Ozveren and A. S. Willsky, "Observability of discrete event dynamic systems," *Transactions on Automatic Control* vol. 35, pp. 797-806, 1990.

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

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

[12] kermeta, " http://www.kermeta.org/."

[13] openArchitectureWare, "http://www.openarchitectureware.org."

[14] D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier, "SiTra: Simple Transformations in Java," in *ACM/IEEE 9TH International Conference on Model Driven Engineering Languages and Systems*, 2006, pp. 351-364.

[15] A. Arsanjani, "Empowering the business analyst for on demand computing " *IBM Systems Journal,* vol. 44, pp. 67-80, 2005.

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

[17] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0," W3C, 2006.

[18] L. Ricker, S. Lafortune, and S. Genc, "DESUMA: A Tool Integrating GIDDES and UMDES," in *8th International Workshop on Discrete-Event Systems*, 2006.

[19] B. Bordbar and A. Staikopoulos, "Automated Generation of Metamodels for Web service Languages.," in *European Workshop on MDA*, 2004.

[20] "Stress Test Strategy," Department: Software Verification, University of Minnesota.

[21] http://www.cs.bham.ac.uk/~bxb/Alodib/Ex1.html.

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

[23] 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.