Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

A taxonomic framework for autonomous service management in Service-Oriented Architecture*

Du Wan CHEUN, Hyun Jung LA‡, Soo Dong KIM
(*Department of Computer Science, Soongsil University, Seoul 156-743, Korea*)
E-mail: {dwcheun, hjla80, sdkim777}@gmail.com
Received Dec. 7, 2011; Revision accepted Mar. 7, 2012; Crosschecked Mar. 7, 2012

Abstract: Since Service-Oriented Architecture (SOA) reveals the black box nature of services, heterogeneity, service dynamism, and service evolvability, managing services is known to be a challenging problem. Autonomic computing (AC) is a way of designing systems that can manage themselves without direct human intervention. Hence, applying the key disciplines of AC to service management is appealing. A key task of service management is to identify probable causes for symptoms detected and to devise actuation methods that can remedy the causes. In SOA, there are a number of target elements for service remedies, and there can be a number of causes associated with each target element. However, there is not yet a comprehensive taxonomy of causes that is widely accepted. The lack of cause taxonomy results in the limited possibility of remedying the problems in an autonomic way. In this paper, we first present a meta-model, extract all target elements for service fault management, and present a computing model for autonomously managing service faults. Then we define fault taxonomy for each target element and inter-relationships among the elements. Finally, we show prototype implementation using cause taxonomy and conduct experiments with the prototype for validating its applicability and effectiveness.

Key words: Service-Oriented Architecture (SOA), Autonomic computing (AC), Cause taxonomy, Services, Faults, Causes, Adaptation
doi:10.1631/jzus.C1100359 **Document code:** A **CLC number:** TP311

1 Introduction

In Service-Oriented Architecture (SOA), service providers develop services with reusable features, and service consumers discover and subscribe to appropriate services at runtime. Since services deployed in service registries expose only their interfaces, service consumers and administrators have limited visibility and manageability of services (Erl, 2007). In addition, services may evolve without informing service subscribers of the changes. Existing services may change their interfaces or stop providing functionalities suddenly. These key features of SOA, including the black box nature of services, heterogeneity, service dynamism, and service evolvability, make service management more challenging than conventional system management (Manes, 2005).

Autonomic computing (AC) is a way of designing systems that can manage themselves in an autonomous manner without direct human intervention (Kephart and Chess, 2003). Applying key disciplines of AC to service management is appealing since key technical issues for service management can be effectively resolved by AC.

A key task of service management is to identify probable causes for symptoms detected and to devise actuation methods that can remedy the causes. When symptoms of problems are detected, the probable causes for the symptoms must be identified, and elements that result in the causes should be remedied

‡ Corresponding author
* Project (No. 2011-0002534) supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology
© Zhejiang University and Springer-Verlag Berlin Heidelberg 2012

in an appropriate way. In SOA, a number of key elements become the targets for service remedies, such as service, business process in a Business Process Execution Language (BPEL) document (OASIS, 2007) or service choreography (W3C, 2005), enterprise service bus (ESB) (Chappell, 2004), and service interface in the form of Web Service Description Language (WSDL) (W3C, 2007b) or representational state transfer (REST) (Richardson and Ruby, 2007). There are also a number of different causes associated with the target elements. To diagnose the problems and find their causes effectively, it is essential to have two solutions available: (1) a well-defined taxonomy of all causes for each target element and (2) applications to apply the taxonomy.

Therefore, the taxonomy of causes provides the basis for reasoning about symptoms and remedying accountable target elements in an autonomous way. Hence, the lack of cause taxonomy results in the limited possibility of determining causes and remedying problems in an autonomous way. However, there is not yet a comprehensive taxonomy of causes that is widely accepted. The cause taxonomy we propose in this paper has the following features which are distinguishable from existing works:

1. Presenting a theoretical foundation for fault management in SOA;

2. Defining a meta-model of SOA as the basis for deriving taxonomy;

3. Proposing a comprehensive cause taxonomy for enabling autonomic service management;

4. Defining a taxonomy that is extendable with new types of causes.

In this paper, we present a comprehensive taxonomy of causes that can be identified under SOA environments. We first present key concepts for managing service faults, identify all the possible target elements for service management, and suggest a computing model for autonomously managing service faults. With the theoretical foundation on service fault management, we define comprehensive cause taxonomy by considering internal structures of all the target elements, inter-relationships among the elements at design time, and inter-relationships among the elements at runtime. Finally, we show prototype implementation using the cause taxonomy and conduct experiments with the prototype for validating applicability and effectiveness.

2 Related work

While there is not yet a comprehensive taxonomy of causes that is widely accepted, many researchers have proposed taxonomy for detecting and diagnosing SOA faults and self-adapting. Brüning *et al*. (2007) proposed fault taxonomy for testing SOA faults. They first presented the five steps, publishing, discovery, composition, binding, and execution, because faults may occur during all steps of the service execution process. Then, they defined five types of faults and their subtypes, as well as causal relationships among fault types. In addition, they demonstrated the application of the taxonomy with an example of a travel agency. They focused on presenting a list of faults that can occur and be detected at runtime. However, descriptions of these faults need to be improved for utilizing fault detection and diagnosis and self-adaptation based on faults as well as SOA faults testing. Huang *et al.* (2006) defined a layering model and diagnosis algorithm for managing service faults. They first presented four layers: service interaction layer, service software layer, execution platform layer, and networking layer. Based on these layers, they defined a window-based fault diagnosis algorithm. Then, they showed simulation results for evaluation of the layering model and diagnosis algorithm. Even though their work focused more on the algorithm than on the layering model for classifying faults, it provided abstraction levels of faults based on layers. However, this classification of the faults needs to be specialized for applying to faults related tasks.

The above two works tended to utilize SOA specific elements such as steps or layers as a basis for defining fault taxonomy. Other works, such as IBM Research Center (2006), provided a meta-model and a template to specify symptoms. In IBM Research Center (2006), a symptom is an abnormal condition which can be observed and caused by a fault. Symptom categories were suggested based on security, operation, availability, and quality of service (QoS). Moreover, an extension point was provided for temporarily undefined situations. All these elements were specified with eXtensible Markup Language (XML). IBM Research Center (2006) focused on enabling AC and providing a general way that can be applied to any other software system. Therefore, the Symptom Reference specification needs to be extended with SOA specific features. Pernici and Rosati (2007) proposed

a taxonomy for SOA faults based on the persistency of the fault. In other words, there are three types of faults: permanent, transient, and intermittent. They showed that this classification can be used to analyze faults and to define the recovery mechanism. However, they provided only a general classification for the faults. These two works addressed the baseline for defining taxonomy for SOA fault diagnosis; i.e., they tended not to cover SOA specific features but to provide a general way to be extended.

In summary, there is still room for defining the cause taxonomy for SOA fault diagnosis in terms of the following issues. The first issue is that the basis for defining cause types does not completely cover the domain of faults that occur. The second issue is that an SOA cause itself is difficult to classify by a machine. These features limit the practicability and applicability of the taxonomy and make it much harder to manage services in an autonomic manner. Similarly, diagnosis in the medical domain is to find the cause of disease, not the symptom.

3 Theoretical foundation on service management

3.1 Meta-model

In this section, we present a basic set of definitions that will be used throughout the discussion of the cause taxonomy for autonomic service management. To clearly understand these concepts, we define a meta-model for representing cause and its related elements as in Fig. 1. The meta-model is followed by the standard usage (Avizienis *et al*., 2004).

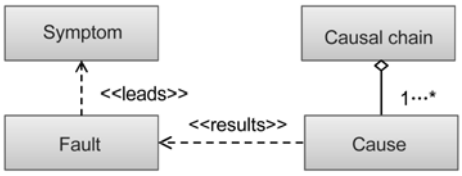

Fig. 1 Meta-model for cause and its related elements

A symptom is an observed state of a service which indicates a pre-defined amount of deviations between a normal state and the observed states. It is a result of a certain fault. A fault is an abnormal condition which may lead to a symptom. A cause initiates a fault, and a causal chain is a tree structure specifying a set of related causes and their relationships.

3.2 Target elements for service management

When a problem occurs in SOA, a cause for the problem occurs in certain elements, and these elements become the targets for remedying the causes. Through our rigorous observations on SOA standards and practices, we first define nine target elements for autonomous service management (Fig. 2): service interface, business process, service component, message, runtime environment (services middleware, BPEL engine, and Simple Object Access Protocol (SOAP) engine), ESB, and service registry.

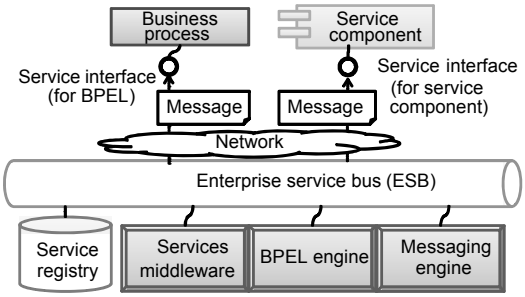

Fig. 2 The nine targets of service management

There are lower-level underlying target elements such as the operating system and network which are accountable for problems, but we do not consider the lower-level elements in this research. While these causes apply to distributed systems in general, this paper focuses on typical faults applying to SOA. Thus, we leave all general elements of distributed systems aside and focus solely on the essential steps of SOA and all causes associated with them. The other causes are described in more detail in the related work.

3.3 Computing model

By extending IBM's AC model (Kephart and Chess, 2003), we present the computing model for managing service faults with five steps as in Fig. 3.

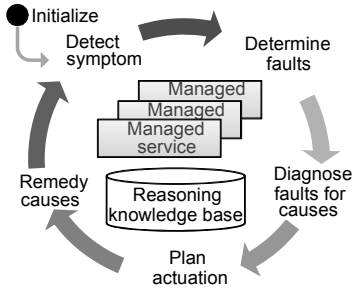

Fig. 3 Five phases of the computing model for managing service faults

The phase 'detect symptom' is to monitor managed services and recognize abnormal states of the services which indicate a significant deviation between a service delivered and the service expected which is specified in service-level agreements. The phase 'diagnose faults for causes' is to identify the cause for the symptom based on a cause taxonomy and to determine the root cause by looking up the knowledge of the causal chain. The phase 'plan actuation' is to make a plan to solve the causes and prepare for the remedy. Finally, the phase 'remedy causes' is to execute adapters on services and/or the underlying SOA environment and resolve the causes.

4 Classification of causes

Typically, it is not feasible to define complete taxonomies of causes, symptoms, and actuators. This is because the numbers of possible causes, symptoms, and actuators could be large and new types can be introduced. On the other hand, taxonomies of the Service Component Architecture (SCA) elements should be available to define effective methods to identify underlying causes for given symptoms and to choose effective actuators for the diagnosed causes.

4.1 Criteria for defining the taxonomy

We define the taxonomy of causes by considering the characteristics of service-oriented computing, SOA standards, and SOA elements/artifacts. Our taxonomy is not meant to be complete but comprehensive and extensible, meaning that new cause types can be defined by subtyping existing cause types. We apply three criteria in defining the taxonomy: completeness on classification, exclusiveness among elements, and incremental extensibility (Hunter, 2002).

A classification is complete if the set of all cause types for a given target element is complete, i.e., not missing any probable cause. We consider the key components of each target to derive a complete list of probable causes. A classification is exclusive if there is not a common property among the cause types for a target element. That is, any two cause types C_i and C_j for a target element should not have any common property between them. A classification is incrementally extensible if a given cause can be extended into its subtypes. With this criterion, abstract cause types can initially be defined and they can be refined into more concrete subtypes.

4.2 Cause types for defining the taxonomy

In software engineering, two types of information, static and dynamic, are considered for software analysis, design, and construction (Arban *et al*., 2005). Static information of the software consists of components and their relations. In object-oriented analysis and design, for example, such components could be classes and the relations could include association between classes. Dynamic information of the software contains states of components and interactions among the components at runtime. The interaction can be specified with sequential and/or concurrent events. By utilizing this analogy, we first classify causes into three types, namely, embedded cause for SOA specific components, relation related cause for relations among the components, and execution related cause to specify runtime behavior.

Fig. 4 shows the three types of causes and their subtypes. These subtypes are elaborated upon in the remaining sections. The main reason for classifying causes into these three types is that the methods for symptom detection, fault diagnosis, and adaptation would considerably differ. For an embedded cause, adaptation is applied to only the target element that results in the cause. For a relation related cause, adaptation is applied to mismatched elements that result in the cause. In remedying execution related causes, a non-trivial reasoning process is required for determining which of the participating targets is responsible for the cause, and for planning actuation on the selected target. However, the determination of the responsible target could not usually be done autonomously.

Embedded cause is a problem or a cause embedded in a service target without being involved in any relation or interaction among target elements. For example, a service interface specified in WSDL may have operation signatures which use undefined data types. This type of cause is not involved in any interaction in form of external references or invocation. Instead, it specifies a problem or a cause on the target element itself.

Relation related cause is a problem of inconsistency or incompatibility existing in the relation

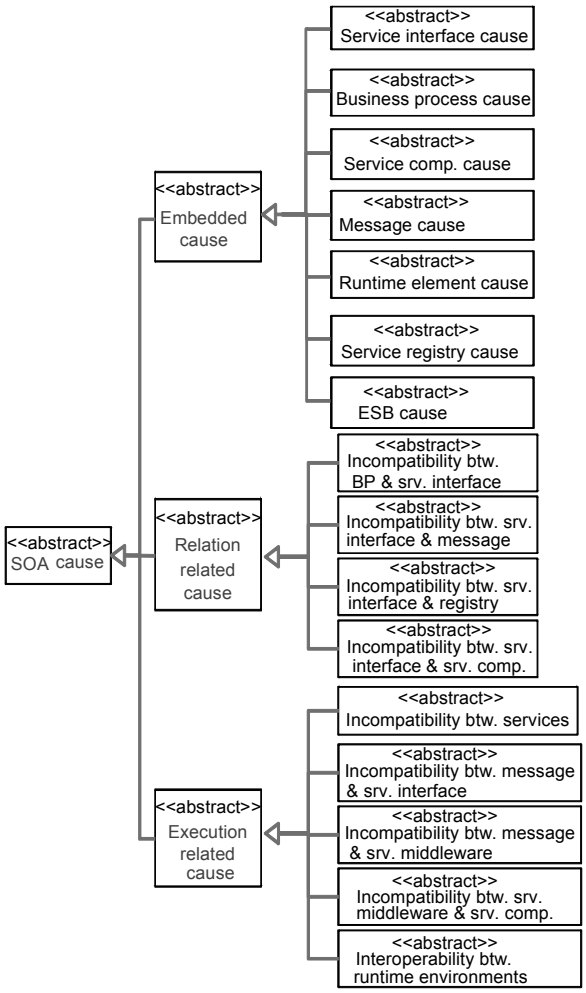

Fig. 4 Taxonomy of SOA-specific causes
btw.: between; srv.: service; comp.: component

between two target elements at static time. In SOA, there is a considerable amount of relations among the target elements, and the relations themselves can result in certain causes.

Let us consider a case of relation between a service interface and a service component (Fig. 5). For a problem in an interaction between them, either the service interface or the service component can be to blame for the cause. Hence, determining the target that results in a relation related cause requires context information beyond the specification of the target elements themselves. In many cases, this context information describing the situation is not readily available. To reason about interaction related causes, such context information should be provided and specified in a machine readable form such as in an XML document.

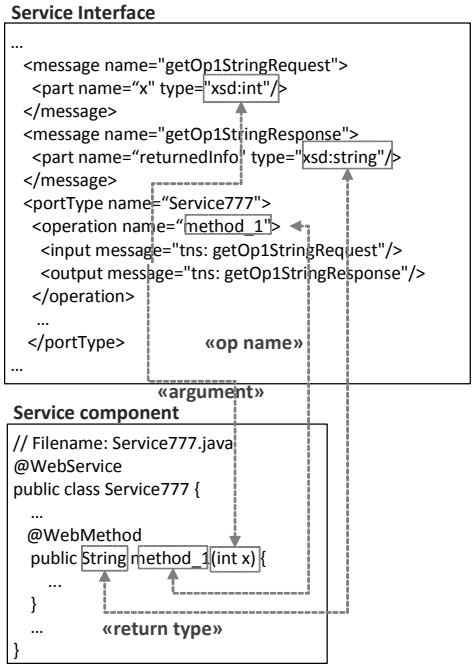

Fig. 5 Matching between the service interface and the service component

Execution related cause is a problem occurring when a service or a business process is invoked at runtime. Even if elements within an SOA based system have no problem and relations among elements are correctly specified, a runtime element such as a parameter or required resources can raise problems.

4.3 Taxonomy of embedded causes

Embedded causes are classified into seven types based on target elements for service management (Fig. 6).

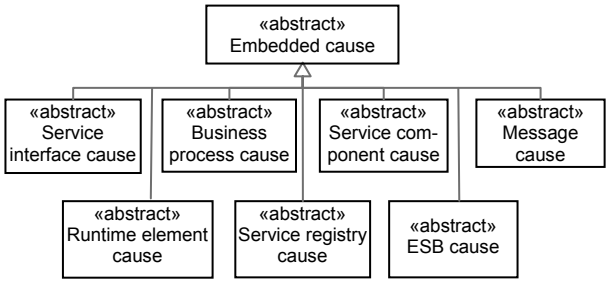

Fig. 6 Seven types of embedded causes

For each target element, probable cause types are derived by considering its sub-elements.

Service interface cause: A service interface specifies capability provided by a service, and they are specified in WSDL with the following key

elements: data type, operation signature, and binding information. Hence, there are four causes for each element. An example of the binding cause is that the required information is missing such as interface and type. Because of this cause, a service cannot be invoked by its endpoint specified in service interface specification, known as a reachability problem (Hamadi and Benatallah, 2003).

Business process cause: A business process (BP) defines a workflow among participating services and it is specified in BPEL with four elements: invocation, assignment, control flow, and waiting statement (OASIS, 2007). Because the BP controls a series of service invocations, there are three variations on each element: invalid information on an element, no information on an element, and incompatible information on an element. Fig. 7 depicts four causes and their sub-causes for each element specified in a BPEL document.

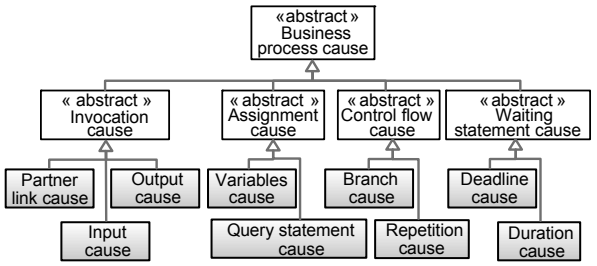

Fig. 7 Cause types on business process specification

For example, there are two sub-causes for the assignment cause: variable cause and query statement cause. To specify a variable attribute of an assignment element for a BPEL document, there are five variants to be selected, i.e., variable, property, expression, literal, and empty. An example of the variable cause is that one of these variants is incorrectly copied from source to destination values. Another example of the branch cause is deadlock (Martens, 2005). This cause can occur more when utilizing choreography for defining services (Brogi *et al*., 2004).

Service component cause: A service component realizes one or more service interfaces. It is distributed as a black-box form. Hence, cause types on service components can be derived from externally exposed properties.

One cause type is about functionality related elements, i.e., input and output. Another is about state related elements, i.e., pre-condition and post-condition. Here, pre-condition and post-condition depict the state of a service component or its related resources before and after executing its functionality. For example, one raw is updated with newer values after a service component updates one table in the database. This updated state can be treated as post-condition. The other is about QoS related elements: unacceptable QoS and no responses (OASIS, 2010). In this case, there are existing references to well-known causes of these two symptoms, such as internal logic cause and inefficient data manipulation cause (Zheng and Lyu, 2010). Fig. 8 depicts three abstract causes and their concrete causes for each element of the service component.

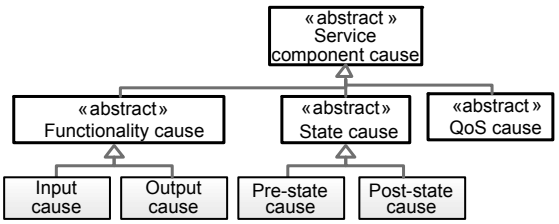

Fig. 8 Cause types on service components

Message Cause: Message is an information exchange among SOA elements through a variety of underlying protocols such as SOAP. This message consists of header and body. The header specifies a destination that a message should be delivered to. The body specifies message contents and its encoding styles. Therefore, there are two causes (i.e., header cause and body cause) and one sub-cause for a body cause. One example of the header cause is the version mismatch cause (W3C, 2007a).

Runtime element cause: A runtime element such as BPEL engine, SOAP engine, or service component middleware has functionalities of processing and executing SOA artifacts. Typically, the runtime element has a form of software provided by a vendor. Hence, cause types on the runtime element can be derived from externally observed states. OASIS (2006) defined four types to classify the states: available, partially available, unavailable, and unknown. That is, all the runtime elements can have causes based on four states. For example, the unavailable cause on a BPEL engine can occur when required resources such as memory are not enough to execute a BP deployed on the BPEL engine.

Service registry cause: Service registry is an infrastructure that enables one to publish and subscribe

to services, conforming to UDDI (universal description, discovery and integration) which defines data structure for describing services and API (application programming interface) for registering services (OASIS, 2004).

Based on these, we define two abstract causes and their concrete causes for each element of the BP specification (Fig. 9). For example, there are two possible situations for the identification cause. First, a unique identity for each data structure is defined for two or more structures. Second, a unique identity for each data structure is not specified or assigned. In addition, there are two possible situations for the description cause. First, the published description for a service is updated and does not conform to the previous one. Second, mandatory information for each data structure is not specified.

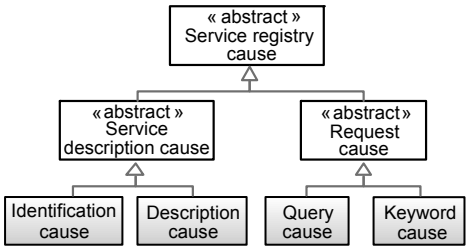

Fig. 9 Cause types on service registry

ESB cause: ESB is an integration platform that supports message routing and data transformation. Java business integration (JBI) is mentioned as a way for implementing ESB (Chappell, 2004). Hence, we extract key elements for JBI based ESB: normalized message router (NMR), binding component (BC), and service engine (SE).

Hence, there are three causes and their sub-causes for each cause (Fig. 10). For example, there are two sub-causes for the NMR cause. To operate NMR, there are two key elements: configuration including the routing table and resource for executing NMR itself. An example of the resource cause occurs when the amount of memory allocated for running ESB is not enough for routing service invocations.

4.4 Taxonomy of relationship related causes

Relationship related causes are classified into four types based on target elements for service management (Fig. 11).

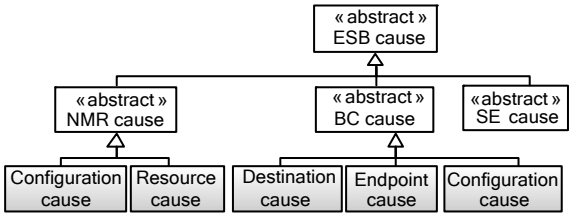

Fig. 10 Cause types on enterprise service bus (ESB)
NMR: normalized message router; BC: binding component; SE: service engine

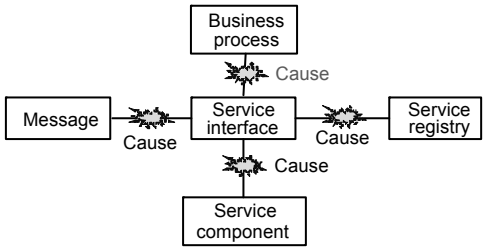

Fig. 11 Relationship among elements and their related causes

For each relationship between target elements, probable cause types are derived by common attributes among SOA elements.

Incompatibility between business process and service interface: Typically, a BP includes one or more service invocations. Hence, the BPEL standard (OASIS, 2007) provides rules to specify the service invocations. First, PartnerLink, which specifies the relationship between a BP and a target service, is declared. In detail, PartnerLink declaration includes an identification of each target service and description of the service such as a role. Second, invocation related information is declared based on WSDL. There are two key elements, a reference to a target service and portType for binding operations. Third, values produced by services are assigned to BP variables through an assignment activity of the BP. Based on these rules, we define three causes: PartnerLink declaration cause, invocation declaration cause, and assignment declaration cause (Fig. 12).

For the invocation cause, two sub-causes could occur. The reference cause can occur when references specified in BPEL and WSDL are not matched and the portType cause can occur when the portType specified in BPEL and the portType specified in WSDL are not matched.

Incompatibility between service interface and message: In SOA, a message is a unit for communication between services or a BP and a service through a service interface. To construct a message, three

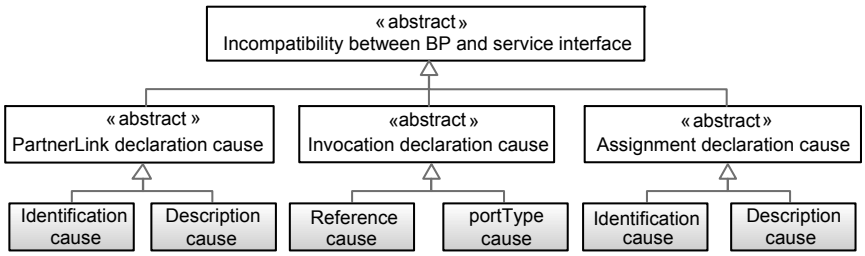

Fig. 12 Cause types on interaction between business process (BP) and the service interface

elements need to be considered: a protocol for communication, a message format, and a set of data types utilized in the message. These elements are predefined in a service interface specification. Therefore, we extract three causes on these elements: protocol declaration cause, message format definition cause, and data type definition cause.

Let us consider an example of the protocol declaration cause. There are two representative protocols to communicate services: SOAP and REST (Richardson and Ruby, 2007). When a service interface is declared in REST and a service request message is specified in SOAP, the resulting fault is due to the protocol declaration cause.

Incompatibility between service interface and service registry: When a service provider publishes his/her service on a service registry, he/she makes a description for the service. The description is based mainly on a service interface specification. Due to this nature, service registry related standards such as UDDI define data structures for describing services, and information on the structure is closely related to a service interface specification. Among attributes to be described and registered on a service registry, the service endpoint and an access point of a service interface are important, because these are related to invoking services. Hence, we extract two causes on these elements, endpoint cause and interface access point cause. Fig. 13 shows an example of the endpoint cause.

In this example, endpoint information stored in the service registry and the access point specified in WSDL are different. Because of an invalid reference, the FooService in the example will not be invoked.

Incompatibility between service interface and service component: A service component realizes one or more service interfaces. That is, a service component is implemented by conforming to predefined

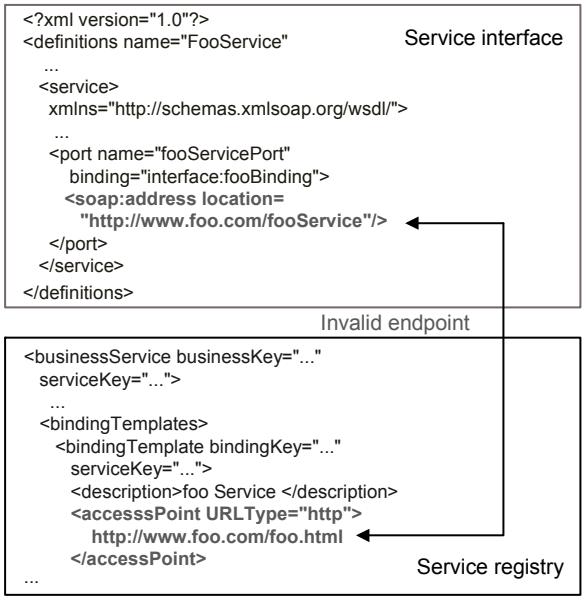

Fig. 13 An example of the invalid reference

values of elements and their attributes of a service interface. These elements include data type, operation, binding, and endpoint. Hence, there are four causes for each element.

For example, message mismatch has two possible situations. First, the data type for the input or output message specified in a service interface and declared in a service component is different from each other. Second, a data type for the input or output message declared in a service component cannot be supported by the WSDL standard, even if the data type is specified in a service interface.

4.5 Taxonomy of execution related causes

Execution related causes are classified into two types: inter- and intra-interaction related causes. For each execution between target elements, probable cause types are derived by incompatibility problems of corresponding elements.

The inter-interaction related cause means a problem that occurs when a service invokes another service. When a service invokes another service, the results of the caller service are sent to the callee service. Hence, runtime messages with real values can raise incompatible problems.

Incompatibility between services: Typically, syntactic information on messages between services is defined in a BP document. Hence, data type and the number of inputs and outputs have already been specified technically. However, a valid range of variables (i.e., inputs and outputs) is varied depending on a domain. Moreover, passed values sent by a sender service are different from the intention of the service whose role is a receiver. Therefore, we extract two probable causes: invalid range cause and incompatible semantic cause (La and Kim, 2011).

Intra-interaction cause means a problem occurring when a service performs its functionality. Different elements participate in executing a service, such as service interface, service component, and service middleware. These elements are managed by a service provider.

As shown in Fig. 14, a client makes a request through the service interface (Erl, 2007) when a service is executed. Then, the request message arrives at a service middleware. Based on the message, the service middleware executes the appropriate service component. Hence, three incompatible problems among three elements can raise incompatibility causes.

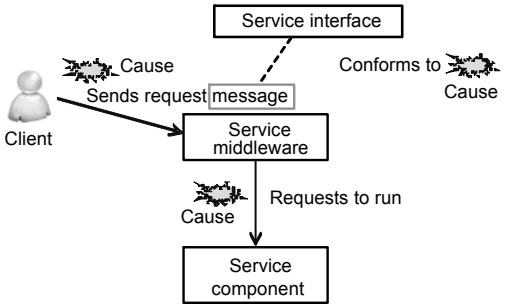

Fig. 14 Intra-interaction on a service

Incompatibility between message and service interface: Typically, a client program is implemented based on a service interface. When the client program requests to invoke the service with the service interface, the program generates a request message that syntactically conforms to the service interface. However, current standards on specifying a service interface have limitations on description for user-defined types and arrays. That is, data types supported by a service interface cannot include all the variable array types and user-defined types, even if additional facilities such as JavaScript Object Notation (JSON) are provided. Due to this limitation on specifying a service interface, i.e., limited supports for data types, a message cannot be interpreted even if it is successfully sent to a service middleware. In this situation, an incompatible problem between the message and service interface may occur.

Incompatibility between message and service middleware: To receive a message, a service middleware is required to support the protocol used for the message. However, there is no mechanism to specify protocols supported by a service middleware. Thus, a service middleware generates errors such as not responding when a message arrives at the middleware. In this situation, an incompatible problem between message and service middleware may occur.

Incompatibility between service middleware and service component: To run functionality provided by a service component, resources of a service middleware are required, such as files, database, and memory. In the service middleware, there can be one or more service components. Hence, resources of the middleware are shared among service components. Therefore, the availability of the resources is varied depending on current status such as how many service components are executed. Hence, required resources of a service component may not be available depending on the status of the middleware. Therefore, there can be an incompatible problem between available resources of service middleware and required resources of a service component.

Interoperability between runtime environments: to execute services, more than one runtime environment is placed at the provider side. Therefore, if there are conflicting settings on the configurations, an incompatible configuration cause occurs.

4.6 Guidelines for handling the evolution of cause taxonomy

The proposed cause taxonomy is not complete, but evolvable with introductions of new SOA implementation technology. In this section, we describe how the taxonomy can evolve, and how the evolution of the taxonomy can be reflected in various fault

manager implementations.

The proposed cause taxonomy is specified with the notion of generalization: abstract causes at the upper layer and concrete causes at the lower layer. Note that the classification of cause types at the abstract level is complete, as illustrated in Section 4.2. Hence, taxonomy evolution will occur at the concrete cause type layer. More specifically, extending the cause taxonomy will practically be tasks of adding new cause types, deleting existing cause types, and modifying existing cause types. These tasks will not require altering the classification of and relationships among abstract cause types. This is mainly due to the nature of 'abstract' in defining high level cause types, as illustrated in Fig. 15 with adding two cause types at the concrete level.

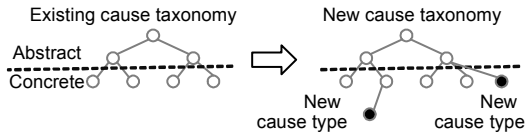

Fig. 15 Extending cause taxonomy with new cause types

Now, we consider the impact of modifying taxonomy in implementations. Tasks of modifying cause types at the concrete level will have a minimal influence on implementations of the cause-related fault manager such as our proof-of-concept (POC) implementation in Section 5. Adding a new concrete cause type will require implementing a new subclass for the type. Deleting an existing cause type will require deleting its corresponding class and handling any other class that depends on the class being deleted. Modifying a cause type will require replacing its corresponding class with a newly implemented class for the type. In summary, modifying the taxonomy can be reflected in implementations without major structural changes.

5 Prototype implementation of the autonomous service fault manager

To show applicability and effectiveness of the proposed cause taxonomy for managing service autonomously, we implemented a POC version of the autonomous service fault manager.

5.1 Functionality of the fault manager

The functionality of the fault manager is captured in three main use cases. The use case 'detect symptom' is to monitor services and find a symptom, i.e., a service state that would potentially yield an abnormality. This is done by observing the output, state, and QoS values of the workflow and its participating services.

The use case 'determine faults' is to determine the occurrence of a fault by the symptom. This is done by comparing the output to the expected output.

The use case 'diagnose faults for causes' is to infer service faults and determine the cause(s) for the given fault in a deterministic and autonomic manner. This is done by employing the Bayesian network inference with our proposed taxonomy.

5.2 Design of the fault manager

The six key components and the control flow of the fault manager are shown in Fig. 16.

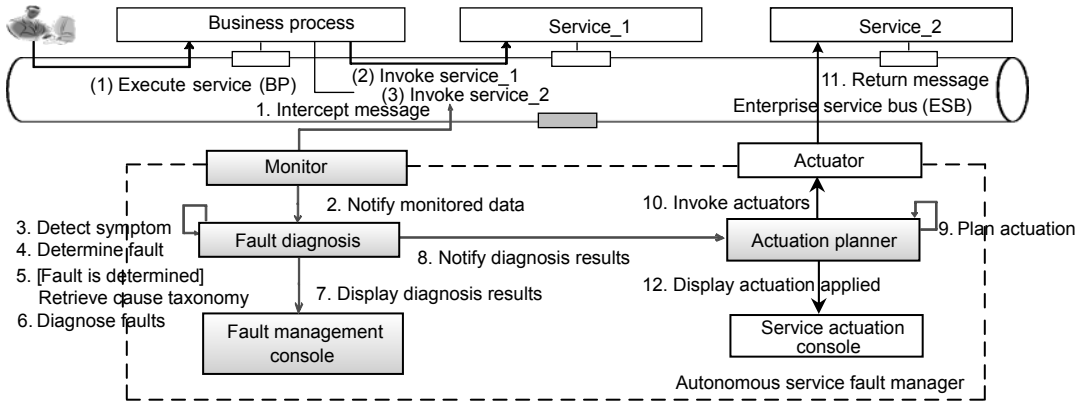

Fig. 16 Control flow of the autonomous service fault manager

The component 'monitor' intercepts messages from service invocation and gathers information such as inputs and outputs, states, and QoS values. Then, the 'fault diagnosis' component detects symptoms, determines faults, retrieves the cause taxonomy, and diagnoses faults in an autonomous manner.

5.2.1 Method to detect SOA symptom

In this subsection, we present a method to detect an SOA symptom as follows:

Step 1 is to extract the set of expected quality values which is typically specified in a Service Level Agreement (SLA) (Dan *et al*., 2004). An SLA document typically addresses a number of quality attributes, and each quality attribute is given a range of valid values.

Step 2 is to compute the current value of each quality attribute for the service, by acquiring quality-related raw data and applying quality metrics.

Step 3 is to compare the expected values to the measured values for the target service. When measured values are out of expected value ranges, an occurrence of abnormality, i.e., a symptom, is detected.

This three-step method for detecting symptoms is further specified as an algorithm below:

Algorithm 1 Detect-Symptom

```
Input:
  SLAs // Retrieve SLA documents
  QualityMetrics // Retrieve a set of quality metrics
Output:
  DetectedSymptomSet
  // Step 1: Extract the set of expected quality values
  ExtractExpectation(SLAs) {
    for each service level agreement SLA_i from SLAs
      // exp_i=(QA_i, ValidRanges_i, condition_i)
      for each exp_i from Expectations
        Assign valid range of SLA_i to ValidRanges_i of QA_i
        Assign condition of SLA_i to condition_i of QA_i
      endfor
    endfor
    return Expectations
  }
  // Step 2: Compute the current values of QAs
  ComputeQoS(QualityMetric) {
    for each quality metric qm_k
      // raw data which is only for qm_k
      Get MonitoredValues of service
      Measure quality value based on qm_k
    endfor
    return MeasuredValues
  }
  // Step 3: Compare expectations to measured values
  CompareExpectationToMonitoredValues
    (Expectations, MeasuredValues) {
    for each expectation exp_i from Expectations
      for each measured values mQV_j from MeasuredValues
        if (exp_i.QA==mQV_j.QA) then
          Check whether exp_i.ValidRange includes
            mQV_j.value
          if (result==false)
            Assign current mQV_j to DetectedSymptomSet
          endif
        endif
      endfor
    endfor
    return
  }
```

5.2.2 Methods to diagnose SOA faults

After symptom detection, there are two sequential steps: fault determination and fault diagnosis. In this subsection, we propose one method for each step. The first method is about detecting candidate faults by following guidelines from detected symptoms to faults.

Candidate faults derived from the function symptom: A function symptom is closely related to inputs and/or output of an execution unit, a service. Let ExecutionUnit_i be the ith execution unit within one invocation of a service or a BP. Then, there are two possible mappings:

Rule 1: $\text{Fault}(\text{ExecutionUnit}_{i-1})$
$\rightarrow \text{Symptom.input}(\text{ExecutionUnit}_i)$

Rule 2: $\text{Fault}(\text{ExecutionUnit}_i)$
$\rightarrow \text{Symptom.output}(\text{ExecutionUnit}_i)$

Candidate faults derived from the state symptom: A state symptom is derived from internal states of a service instance. There are four possible mappings:

Rule 3: $\text{Fault}(\text{RuntimeElement}_k)$
$\rightarrow \text{Symptom.pre\_state}(\text{ExecutionUnit}_i)$
$\vee \text{Symptom.post\_state}(\text{ExecutionUnit}_i)$
$\&\& \text{utilizedBy}(\text{RuntimeElement}_k, \text{ExecutionUnit}_i)==\text{true}$

Rule 4: $\text{Fault}(\text{ESB}_k)$
$\rightarrow \text{Symptom.pre\_state}(\text{ExecutionUnit}_i)$
$\vee \text{Symptom.post\_state}(\text{ExecutionUnit}_i)$
$\&\& \text{utilizedBy}(\text{ESB}_k, \text{ExecutionUnit}_i)==\text{true}$

Rule 5: Fault($Unit_k$)
→Symptom.post_state($ExecutionUnit_i$)
&& $Unit_k \in$ {$ExecutionUnit_1$, $ExecutionUnit_2$, …, $ExecutionUnit_{i-1}$}
&& share($Unit_k$, $ExecutionUnit_i$)==true

Rule 6: Fault($ExecutionUnit_i$)
→Symptom.post_state($ExecutionUnit_i$)

In this mapping, we utilize two additional predicates, i.e., utilizedBy(i, j) and share(k, l). The first is interpreted as element i is utilized by element j and the second as k and l share the same resources such as database.

Candidate faults derived from the QoS symptom: A QoS symptom is detected by comparing computed values of quality attributes to requirements. There are four possible mappings:

Rule 7: Fault($ExecutionUnit_i$)
→Symptom.QoS($ExecutionUnit_i$)

Rule 8: Fault($RuntimeElement_k$)
→Symptom.QoS($ExecutionUnit_i$)
&& utilizedBy($RuntimeElement_k$, $ExecutionUnit_i$)==true

Rule 9: Fault(ESB_k)
→Symptom.QoS($ExecutionUnit_i$)
&& utilizedBy(ESB_k, $ExecutionUnit_i$)==true

Rule 10: Fault($ServiceRegistry_k$)
→Symptom.QoS($ExecutionUnit_i$)
&& utilizedBy(ESB_k, $ExecutionUnit_i$)==true

Through this mapping process, a list of candidate faults is identified. Based on this and the taxonomy, a causal chain is constructed, i.e., a form a of Bayesian network. To do this, we need to transform a propositional expression with a service network to a directed acyclic graph (DAG), i.e., Bayesian network construction. This construction consists of four steps, as follows:

Algorithm 2 Construct-Causal_Chain

```
Input:
  // Retrieve a list of services
  Services: array [Svc1, Svc2, …, Svck]
  // Retrieve a list of relationships among the services
  Relationships: array [Rel1, Rel2, …, Relm]
  // Retrieve a list of causes
  Causes: array [C1, C2, …, Cl]
  // Retrieve a list of relationships among the causes
  Specializations: array [Spe1, Spe2, …, Spen]
  SymptomSet: array [Symptom1, Symptom2, ..., Symptomn]
Output:
  CausalChain: Graph(Vertex, Edge)
  // Step 1: Map a service network to a DAG
  MapFromServiceNetToDAG(Services, Relationships) {
    for each service Svci from Services
      // Vertexi∈Vertices && Vertices⊂DAG
      Map Svci to Vertexi
      Set Vertexi to DAG
    endfor
    for each Relationship Reli from Relationships
      Map Relj to Edgei  // Edgei∈Edges && Edges⊂DAG
      Set Edgei to DAG
    endfor
    return
  }
  // Step 2: Refine graph with candidate faults
  GenerateFaultTree(SymptomSet, MappingRules, DAG) {
    for each symptom Symptomi from SymptomSet
      // by applying Rule 1 and Rule 2
      Retrieve candidate fault types
      for each vertexk in DAG
        if (Symptomi.occurencePlace==vertexk)
          // x is the number of candidate fault types
          Append x vertices
          Append 2x edges
          Update DAG with x vertices and 2x edges
        endif
      endfor
    endfor
    return
  }
  // Step 3: Refine graph with cause taxonomy
  ConstructCausalChain(DAG, Causes, Specializations) {
    for vertexx and vertexx+1 in DAG
      for each Cl in Causes
        if (Cl.Elements.contains(vertexx) &&
          Cl.Elements.contains(vertexx+1))
          Append C1 to CausalChain
        endif
      endfor
    endfor
    return
  }
  // Step 4: Refine graph with cause taxonomy
  Retrieve conditional probability table (CPT) for each edge
```

This causal chain is optimized with the cause taxonomy in terms of the range of possible causes and correctness for identifying cause. First, irrelevant causes are removed from the causal chain through the mapping process with the 10 rules, in terms of two parameters: the number of targets to be traced and the number of cause types to be checked. Second, the

place in which the cause occurs is identified as well as the types of causes. For the reasoning about the cause, we utilize the two-step Bayesian inference algorithm (Fig. 17).

Step 1

```
set e;
for each v in G, V
   for each ('p1', 'v') in G
      for each ('p2', 'v') in G
         if 'p1' != 'p2' and ('p1', 'p2') is not in G, E and
          ('p2', 'p1') is not in G, E then
            e=e union ('p1', 'p2');
         end
      next
   next
next
G, E=G, E union e;
```

Step 2

```
Constant: V=set of all vertices in graph;
          n=number of vertices in graph;
          E=set of all edges in the graph;
Variable i, j: integer;
         v, w: vertex;
         set: array [0, n-1] of subset of vertices;
         size: array[V] of integer;
         alpha: array [V] of integer;
         alphainv: array[1…n] of vertex;
Begin
   for i:=0 to (n-1) do
      set[i]:=0;
   for each v in V
      begin
         size['v']:=0;
         add 'v' to set [0];
      end
   i:=1; j:=0;
   while 'i'<=n do
      begin
         'v':=delete any from set['j'];
         alpha['v']:='i'; alphainv['i']:='v';
         size['v']:=-1;
         for w is in V–{'v'} do
            if ('v', 'w') is in E and size['w']>=0 then
               begin
                  delete w from set[size['w']];
                  size['w']:=size['w']+1;
                  add 'w' to set[size['w']];
               end
         i:='i'+1;
         j:='j'+1;
         while j>=0 and set['j']=0 do
            j:='j'–1
      end
End
```

Fig. 17 Two-step Bayesian inference

The first step is to moralize the Bayesian network. A moral graph is a concept in graph theory, used to find the equivalent undirected form of directed acyclic graph. It is a key step of the junction tree algorithm, used in belief propagation on graphical models.

The second step is to triangulate the moralized graph through the maximum cardinality search algorithm with cause taxonomy. An undirected graph is triangulated if every cycle of length greater than three possesses a chord. Through these two steps, we can derive possible causes for the target element.

Our cause taxonomy can especially ease the burden of diagnosing faults. This is because the taxonomy provides classifications of all causes for each target element and relationship among the elements. Hence, it reduces steps for tracing elements within the service execution path. Typically, a symptom is observed in a specific target element. Then, a range of possible causes for the symptom is from the causes of the target elements to the causes of related target elements within next nodes in the graph.

5.3 Implementation of the fault manager

Based on the design, we implemented a POC version of the fault manager, which consists of three inference-related components: monitor, fault diagnosis component, and actuation planner. Fig. 18 shows the results of monitoring QoS of four SOA elements with the monitor: business process B, service A.I, service A.II, and service registry R. B, A.I, A.II, and R indicate identities of each element.

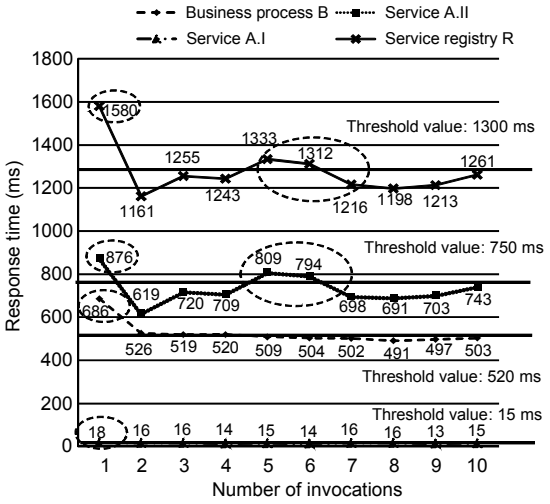

Fig. 18 Monitoring QoS results on SOA elements
The area marked with an ellipse indicates that a symptom is observed at that time frame

Based on the monitoring results, the fault diagnosis component first reduces the ranges of possible causes based on the cause taxonomy (Fig. 19).

```
1   public void mapFunctionFactToTaxonomy() {
2     Vector<Fact> addition=new Vector<Fact>();
3     for (Fact info: additionalInfo) {
4       if (info.getName().equals("Exception")) {
5         addition.add(info);
6       }
7     }
8     int exceptionIndex=0;
9     for (int i=0; i<falseFunctionPredicate.size(); i++) {
10      for (Fact info: addition) {
11        if (falseFunctionPredicate.elementAt(i).
           equals(info.getDescription())) {
12          exceptionIndex=i+1;
13          falseFunctionPredicate.remove(exceptionIndex);
14        }
15      }
16    }
17    Vector<String> trueRelation=
        makeRelation(trueFunctionPredicate, Causes);
18    Vector<String> falseRelation=
        makeRelation(falseFunctionPredicate, Causes);
19    findFunctionConflicts();
20    functionMismatch=
        findFunctionConflicts(trueRelation, falseRelation);
21  }
```

Fig. 19 Source code for mapping function symptoms to candidate faults

Then, this component infers the causes using existing Bayesian network inference engines. As a result, specific cause types for the faults are identified.

5.4 Experiments with the fault manager

Our experiment environment consisted of JDK 1.6.0_15 as the Java virtual machine, Bayesian network tools in Java as the probability-based reasoning engine, and mySQL 5.0.67 as the database management system. The experiment platform ran 64-bit Windows 7 with Intel 3.0 GHz Core 2 Duo and 4 GB of RAM. For the experiment, we considered one BP and its nine services, i.e., ServiceSet:={A, B, C, D, E, F, G, H, I}. And, there were four possible execution paths: A→B→E→G→I, A→C→F→G→I, A→C→F→H→I, and A→D→F→H→I. We assumed that we detected state symptom on service F. With execution paths and rules, mappings from symptom to fault were constructed (Fig. 20).

After mapping, a causal chain was constructed with CPTs (Fig. 21a). By conducting inference using the Bayesian network, we could obtain an inference result of reasoning about cause; i.e., service component F has a cause. This shows the way to find the most problematic service within a given BP. To acquire more accurate causes, the fault diagnosis component should trace repeatedly over all the underlying elements of service component F based on the proposed cause taxonomy. In summary, Fig. 21 shows that our cause taxonomy can be used in reasoning causes.

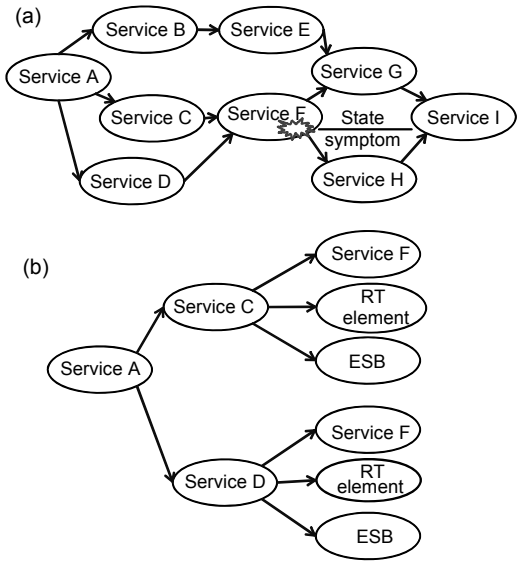

Fig. 20 Results showing mapping from symptom to fault
(a) Service network; (b) Refined service network with faults

However, we cannot show how accurate the diagnosis with cause taxonomy is. Therefore, we compare two cases: the proposed cause taxonomy is applied to the reasoning method or not. Our evaluation on accuracy of diagnosis results is based on coverage of the results. An accuracy of 100% means that the diagnosis result includes the exact set of faulty components. For example, we assume that services A and B within the target SOC based system are faulty components. If our diagnosis system diagnoses faults and concludes that only service A is a faulty component, the accuracy of our diagnosis results is 50%.

As shown in Fig. 22, the accuracy of the Bayesian network with the proposed cause taxonomy is higher than that of a plain Bayesian network, in terms of cause types (Zhang *et al.*, 2009). By applying the proposed cause taxonomy, the search space for Bayesian inference is reduced; i.e., the domain of Bayesian inference is limited by the proposed cause taxonomy. Therefore, the accuracy of the reasoning result becomes higher and the reasoning becomes faster.

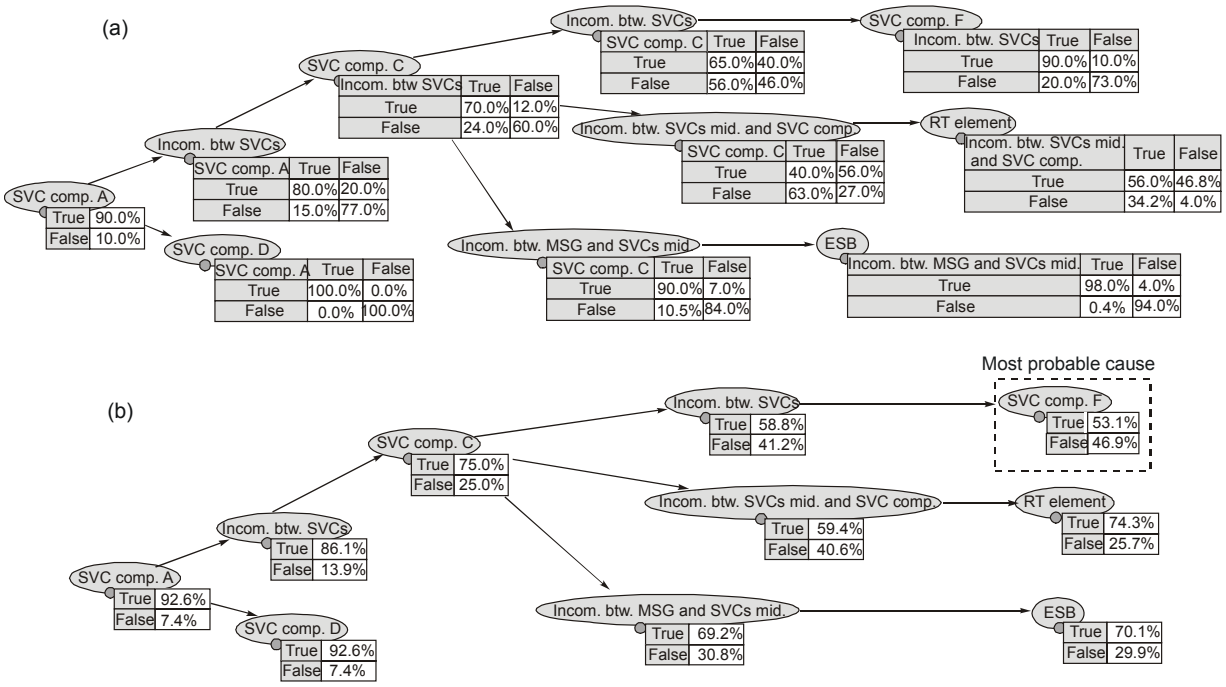

Fig. 21 Bayesian inference result: (a) Bayesian network, i.e., causal chain; (b) inference result

SVC component is the most probable cause. comp.: component; btw.: between; incom.: incompatibility; mid.: middleware

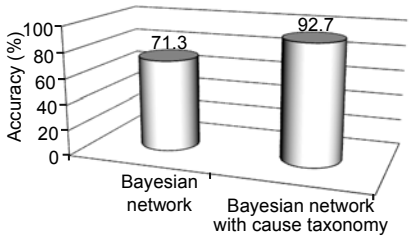

Fig. 22 Comparison of accuracy of reasoning

6 Assessment and conclusions

A prerequisite to applying autonomic service management is a reliable and comprehensive taxonomy for causes of services. This is because the taxonomy provides the basis for reasoning about symptoms and remedying accountable target elements in an autonomous way. Hence, in this paper, we propose a comprehensive but extendable taxonomy of services with a technical observation on the computing paradigm and characteristics of SOA.

We first define the theoretical foundation for fault management issues in SOA, which becomes the basis for the reasoning process in service management. That is, we define a meta-model of SOA for deriving the cause taxonomy. There are a number of components collaborating together in an SOA-based system, and the provided meta-model clarifies inter-component which is an essential prerequisite to effective inference for fault diagnosis.

Then, we define cause taxonomy for enabling autonomic service management. In complex systems such as the SOA-based system, it is infeasible to manage a number of potential faults and cause types in a manual manner. The taxonomy is derived by considering internal structures of all the target elements, inter-relationships among the elements at design time, and inter-relationships among the elements at runtime. Based on the taxonomy, subsequent management tasks such as fault diagnosis and quality actuation methods can be performed in autonomous ways. The proposed taxonomy is designed for further extension with new types of causes.

The results of our proof-of-concept implementation of a cause diagnosis application based on the taxonomy show the applicability of our proposed taxonomy which helps infer the causes of faults. We believe that the proposed taxonomy of causes in SOA would become an essential foundation for implementing autonomic service management applications in SOA.

References

Arban, A., Moore, J., Bourque, P., Dupuis, R., 2005. Guide to the Software Engineering Body of Knowledge. IEEE Computer Society, California, USA.

Avizienis, A., Laprie, J., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Depend. Secure Comput.*, **1**(1):11-33. [doi:10.1109/TDSC.2004.2]

Brogi, A., Canal, C., Pimentel, E., Vallecillo, A., 2004. Formalizing Web service choreographies. *Electron. Notes Theor. Comput. Sci.*, **105**(10):73-94. [doi:10.1016/j.entcs.2004.05.007]

Brüning, S., Weißleder, S., Malek, M., 2007. A Fault Taxonomy for Service-Oriented Architecture. Proc. 10th IEEE High Assurance Systems Engineering Symp., p.367-368. [doi:10.1109/HASE.2007.46]

Chappell, D., 2004. Enterprise Service Bus. O′Reilly, California, USA.

Dan, A., Davis, D., Kearney, R., Keller, A., King, R., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., Youssef, A., 2004. Web services on demand: WSLA-driven automated management. *IBM Syst. J.*, **43**(1):136-158. [doi:10.1147/sj.431.0136]

Erl, T., 2007. SOA Principles of Service Design. Prentice Hall, Boston.

Hamadi, R., Benatallah, B., 2003. A Petri Net-Based Model for Web Service Composition. Proc. 14th Australasian Database Conf., p.191-200.

Huang, X., Zou, S., Wang, W., Cheng, S., 2006. Layering Model and Fault Diagnosis Algorithm for Internet Services. Proc. Int. Multi-Conf. on Computing in the Global Information Technology, p.22. [doi:10.1109/ICCGI.2006.45]

Hunter, E.J., 2002. Classification Made Simple. Ashgate Publishing, Surrey, England.

IBM Research Center, 2006. Symptoms Reference Specification, Version 2.0. IBM Autonomic Computing Symptom Specification.

Kephart, O., Chess, M., 2003. The vision of autonomic computing. *Computer*, **36**(1):41-50. [doi:10.1109/MC.2003.1160055]

La, H., Kim, S., 2011. Static and dynamic adaptations for service-based systems. *Inform. Software Technol.*, **53**(12):1275-1296. [doi:10.1016/j.infsof.2011.06.001]

Manes, A., 2005. The Elephant Has Left the Building. Intelligent Enterprise, NY.

Martens, A., 2005. Analyzing Web service based business processes. *LNCS*, **3442**:19-33. [doi:10.1007/978-3-540-31984-9_3]

Organization for the Advancement of Structured Information Standards (OASIS), 2004. UDDI Version 3.0.2, UDDI Specification Technical Committee Draft.

Organization for the Advancement of Structured Information Standards (OASIS), 2006. Web Services Distributed Management: Management of Web Services (WSDM-MOWS 1.1).

Organization for the Advancement of Structured Information Standards (OASIS), 2007. Web Services Business Process Execution Language Version 2.0 (WS-BPEL 2.0).

Organization for the Advancement of Structured Information Standards (OASIS), 2010. SOA-EERP Business Quality of Service Version 1.0.

Pernici, B., Rosati, A.M., 2007. Automatic Learning of Repair Strategies for Web Services. Proc. 5th European Conf. on Web Services, p.119-128. [doi:10.1109/ECOWS.2007.13]

Richardson, L., Ruby, S., 2007. RESTful Web Services. O′Reilly, California, USA.

World Wide Web Consortium (W3C), 2005. Web Services Choreography Description Language Version 1.0 (WS-CDL 1.0).

World Wide Web Consortium (W3C), 2007a. Simple Object Access Protocol (SOAP) 1.2.

World Wide Web Consortium (W3C), 2007b. Web Services Description Language (WSDL), Version 2.0, Part 0: Primer.

Zhang, J., Chang, Y., Lin, K., 2009. A Dependency Matric Based Framework for QoS Diagnosis in SOA. Proc. IEEE Int. Conf. on Service-Oriented Computing and Applications, p.1-8. [doi:10.1109/SOCA.2009.5410261]

Zheng, Z., Lyu, M.R., 2010. An adaptive QoS-aware fault tolerance strategy for Web services. *Emp. Software Eng.*, **15**(4):323-345. [doi:10.1007/s10664-009-9126-8]