A Model Driven approach for Application-specific Mismatch Detection

Amit Raj
School of Computer Science and Statistics
Trinity College of Dublin, Ireland
Email: araj@scss.tcd.ie

***Abstract*—Several large-scale systems are developed in modules that are assembled together to deliver a desired functionality. The modules developed in isolation may not implement the desired requirements. It creates several problems either at design-time, deployment-time or at run-time. Existing research defines a class of such problems as mismatches. A mismatch may cause malfunctioning or crash of a system. Therefore, a mismatch needs to be detected and resolved. The paper describes a model-driven generic framework to detect mismatches across a variety of systems. A case-study of a large-scale system has been demonstrated to illustrate the application of the framework. An evaluation of the framework illustrates that it is highly re-usable, requires comparatively less efforts and detects a large variety of mismatches.**

Keywords- mismatch detection; models@run.time; model driven engineering;

I. INTRODUCTION

Advanced modern systems are built to cope with the evolving requirements and environment. Such systems are equipped with several mechanisms and services that dynamically adapt. When such systems are required to adapt, several adaptation mechanisms are carried out to assemble, configure, connect and execute the services. These mechanisms are likely to work in an unforeseen environment with the unforeseen services. For example, a crisis-management system may work either with a transport service or with the army service to evacuate the people from a flood site [1]. It makes a system error prone and may deliver undesirable results.

A class of reasons for the erroneous results is the mismatches. A mismatch is a deviation from the required structure and behavior [2]. For example, a system requirement state that a particular service should be available all the time, however, when the system violates this requirement, a service-not-found mismatch occurs. Several past studies exist that formally define the mismatches [3] [4] [1]. We found that the mismatches can be categorized into two categories: (1) Application-independent mismatches (AIM), and (2) Application-specific mismatches (ASM). An AIM defines a mismatch on application-independent concepts such as a component, service, protocol, partner-link, etc. For example, a service-not-found mismatch is an AIM that means a service does not exist that should ideally be available, according to the pre-defined requirements. An ASM is a mismatch in the application-specific concepts of an application such as a flight reservation service, HTTP protocol, etc. An example of ASM is Air-Lingus flight-reservation service does not exist due to failure or inability to register itself in the service registry. A detailed description of AIM and ASMs are given in Figure 1.

Several past studies exist that seems to carry out the detection of AIMs only [5] [6] [7] [8]. According to our knowledge, no studies exist that enable the detection of ASMs. The detection of ASMs is important because AIMs do not exhibit sufficient information to locate the exact problem. A service not found mismatch is not sufficient to locate and resolve the problem. So, a mechanism is required to identify that the Air-Lingus flight reservation service does not exist. It makes the problem resolution more efficient. In the lack of such a mechanism, the system developers and tester invest a huge amount of time in locating the unavailable service among thousands of other services in the service registry. A similar case can be seen in Java programming language when a null pointer exception is thrown, a developer invests a lot of time in locating the variable that was null. The key problem is to detect the application-specific mismatches in a large complex system.

The paper presents a novel approach that supports to detect and locate the ASMs in a system. The approach leverages the use of models@run.time [9] to analyze the run-time conditions of a system. A run-time model is an abstract or reduced representation of system concepts at run-time [9]. A runtime model can be constructed by collecting the run-time data through monitoring. On the other hand, the domain knowledge of a system is required to compose the ASMs. A mismatch detection engine evaluates the ASMs on the run-time model to check if an ASM exist. The detected ASMs will be reported to the developer for their resolutions.

The key contribution of the paper is the mechanism to detect and locate the ASMs automatically to analyze the exact problem in a large complex system. The main benefit of the approach is the detection of exact problem that helps the developers to apply an accurate recovery action quickly. Existing approaches find the type of the problem rather than the exact problem. Out mechanism requires only domain knowledge and run-time monitored

data of a system. The runtime data is used to instantiate application-specific run-time model (ASRM). The domain knowledge is used to describe the ASMs. We evaluated our approach for the re-usability and efforts against other mismatch detection approaches. We found that our approach requires far less input and lines of code.

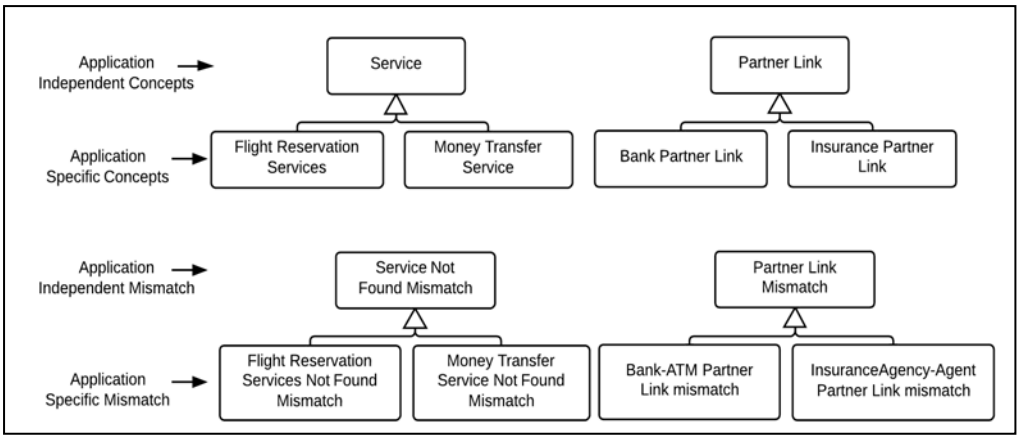

Figure 1. Categorization of Application concepts and mismatches.

II. STATE OF THE ART

Service Oriented Architecture (SOA) has evolved as a standard to developed service-based applications where services are loosely coupled and can be developed independently. Bruning et al. [3] present a taxonomy of faults for the service invocation in SOA based applications. They argued that an SOA based application can have faults of broadly five categories: publishing faults, discovery faults, composition faults, binding faults and execution faults. Li et al. [10] present a classification of service composition mismatches.

Composition and binding mismatches occur when a system composes and binds its constituents to deliver a functionality. In a component composition environment, several components are likely to be developed in isolation and integrated at the time of deployment. While integration, it has been noticed that several components are not able to integrate with others that raises component mismatches [11]. There might be several specific reasons of such inability for integration such as interface mismatch, no common communication protocol mismatch, etc., that may be listed as the sub categories of the component mismatches. Egyed et al. [11] list a set of mismatches by checking the constraints against the description of components and their connectors. They check the constraints on the system at design-time. A violation of a constraint is reported as a mismatch. Gacek [12] has worked on detecting architectural mismatches during system composition using the conceptual features for comparing the architectural styles.

The execution faults occur when the execution behavior of a component deviates from the desired behavior. In such cases, a component faces behavioral mismatches and behavioral adaptations are carried out to recover a system. Carlos et al. [13] provide an algorithm to carry out the behavioral adaptations to overcome behavioral mismatches. They argue that two or more components cannot interact until they reach a correct termination state. The interaction of components requires a match between component signatures. Xie et al. [8] provide an approach to find a component's signature level mismatches with the use of a context free expressions. A component is designed in mathematical specification and a set of rules/conditions decides if a mismatch has occurred or not. Canal et al. [14] describe the adaptation process when a behavioral mismatch occurs, but does not describe how a mismatch can be detected.

In order to detect run-time mismatches, a system is required to be monitored for run-time situations. Moser et al. present the VieDAME environment [15] to monitor the BPEL process and partner service substitution by extending the ActiveBPEL engine. It registers the services and marks them as replaceable which are replaced on request. However, it does not address what are the faults and how these faults are processed to make decisions about replacement of services. Barbon et al. [16] present RTML, an executable language that is used to monitor service-based applications (SBA) properties. This monitoring mechanism is based on the interception of messages that are passed from one service to another. The interception provides the data that are verified against SBA properties. Khaled et al. [17] and George [18] have worked on the event-based monitoring of functional and non-functional properties such as SLA. They proposed the EC assertion to specify service guarantee in terms of different types of events. They assess the EC assertions based on the Event Calculus (EC). This monitoring works in parallel to business process execution, however does not address several erroneous situations. Farrel et al. [19] also worked on SLA-based monitoring using EC. It assess whether the resources available are of certain quality characteristics. Several contracts are defined to assure quality characteristics and contract analysis is performed using EC to assess any violation of contracts.

Several modern systems are developed as multi-layered systems to allow loose coupling between components and layers. The monitoring and mismatch detection in such systems are extremely difficult as different layers may be developed using different programming languages and heterogeneous components. Nickelsen et al. [7] provide probabilistic network fault diagnosis mechanism using Bayesian network of cross-layer observation of a multi-layer network system. Popescu et al. [1] present a dynamic cross-layer adaptation mechanism using adaptation templates. They present a taxonomy of mismatches and provide a matchmaking algorithm to find the best matching adaptation template for a mismatch. When a mismatch is detected in a layer, the best match adaptation is carried out. However, it does not present any mechanism to describe the detection of mismatches. Moreover, they discuss only AIMs not the ASMs. Guinea et al. [20] present a framework that allows different techniques to monitor different layers, but employs a centralized adaptation agent to collect the events and analyze the violation of key performance indicators. They employ different monitoring agents for different layers such as Dynamo for software and Laysi for infrastructure. These agents monitor the events and send to Ecoware [21] for correlation analysis.

The above state of the art describes that mismatch detection techniques or mismatch monitoring techniques are addressing only AIMs such as architectural mismatches, component mismatches, protocol level mismatches, etc. When existing techniques detect such mismatches, a further investigation is required for application specific problem determination. It is clear that existing techniques are working at an abstract level and do not directly relate to any run-time entity of a system, for example, Popescu et al. [1] detects a protocol-mismatch and trigger a desired adaptation. However, they do not specify which protocol had a mismatch that may create ambiguity because a system may have several protocols. In such situations, mismatches may be mis-handled as mismatches are usually ambiguous and represent different meanings in different situations [22]. The state of the art illustrates that a mechanism is required to detect the ASMs in a system.

III. PROBLEM DESCRIPTION

The paper addresses the problem of ASMs detection in large complex systems.

A. Case Study

Our previous work [23] describes a crisis management system (CMS) that is a large-scale complex software system used to manage the evacuation process at the time of flood (see Figure 2). The CMS has sensor components to record the observations sent from sensors located near river or sea side. When a flood incident is noticed by the sensor components, the flood information is notified to the EmergencyCentre. The EmergencyCentre coordinates with several other components and services to carry out a rescue operation at the flood location. For more information about the CMS, please read [1].

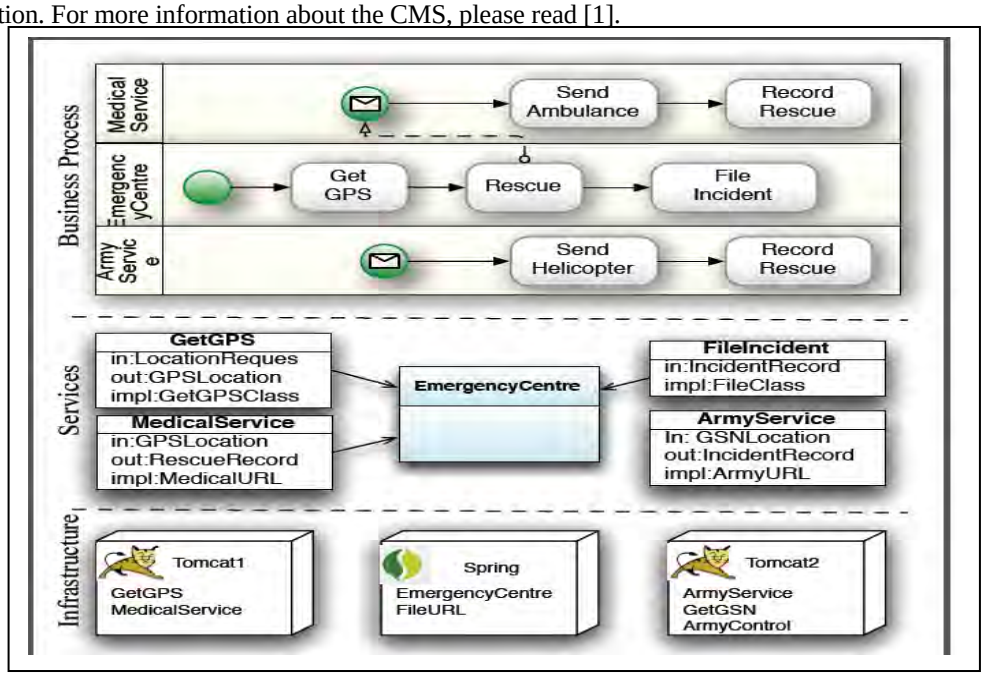

Figure 2. A Crisis Management System.

B. Problem

The described CMS is a multi-layer complex system that has different services and servers. In order to carry out the evacuation process, the 'EmergencyCentre' asks the 'MedicalService' to send the ambulances at the flood site. When the communication protocol between 'EmergencyCentre' and 'MedicalService' is failed, the requested message will not be delivered. In this case, existing mismatch detection techniques detect an AIM

'communication-protocol mismatch' has occurred in the system. This information is ambiguous and insufficient to locate the exact problem. Thus, the developers of the system will have to further investigate whether 'EmergencyCentre'-'MedicalService' or 'EmergencyCentre'- 'ArmyService' communication-protocol has failed. In order to detect the exact problem, a mechanism is required that can inform a specific ASM 'EmergencyCentre'-'MedicalService' communication-protocol has failed. In another scenario where evacuation requires helicopter ambulance to evacuate people, the 'EmergencyCentre' need to invoke 'ArmyService' service. When the 'ArmyService' is not available, existing mismatch detection techniques detect an AIM 'service not found mismatch'. However, in this case a specific ASM 'ArmyService not found mismatch' should be detected.

Several approaches exist to identify the AIMs such as 'communication-protocol mismatch', 'service not found mismatch', etc. However, no technique exist that can identify ASMs. In the current state of the art, the detection of an AIM just hints a developer about a mismatch. A developer has to further analyze the system and its data to identify the exact mismatch that is a cumbersome and error-prone process. For example, when an AIM 'service not found mismatch' is detected, a developer still has to spend a lot of time and effort in identifying the exact service that is not available. Therefore, a key problem is to identify the AIM and ASM both in a system.

IV. THE APPROACH

Our model-driven approach presents a generic architecture that collaborates with a set of models to detect ASMs. The following sections present the architecture of the proposed mismatch detection framework, describe its constituents modules and present a rule engine that detects an ASM.

C. A-Mistec Framework

This section presents a generic and extensible Application-specific Mismatch deTection (A-MisTec) framework to detect the ASMs (Figure 3). The framework consists of five basic modules that are Application-independent Run-time Model (AIRTM), Application-specific Run-time model (ASRTM), Application-independent Mismatches Metamodel (AIMM), Application-specific Mismatches Model (ASMM) and a mismatch detection engine (MDE).

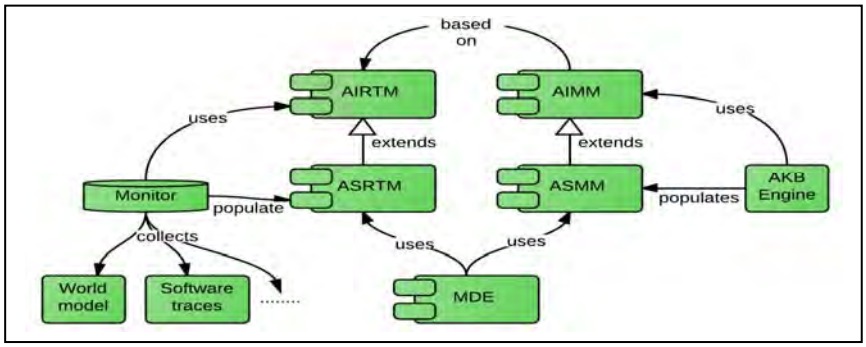

Figure 3. A-MisTec Framework.

1) Application-independent Run-time Model : Different applications generally differ from each other in term of structure and behavior. For example, flight reservation service's structure and behavior differs from a banking transaction service. However, several initiatives, in particular OMG UML [24], work on unifying the structure and behavior of applications. Our detailed analysis illustrates that several abstract concepts exist that are common for a variety of applications. Such concepts can be used to represent AIRTM. The paper presents a generic and extendable AIRTM (Figure 4), however our framework is flexible to use any existing model as AIRTM such as UML Class diagram, sequence diagram, object diagram, SCA models [25], EMF models, etc. These existing models capture only a very specific aspect of an application that is not sufficient to represent the complete run-time situations of a system. Moreover, such a model supports the detection of mismatches for a specific aspect of a system. For instance, a UML Class diagram supports the detection of class related mismatches, but not the protocol related mismatches. The paper presents an extendable AIRTM that is motivated from the existing work [26] [27] [28] [29].

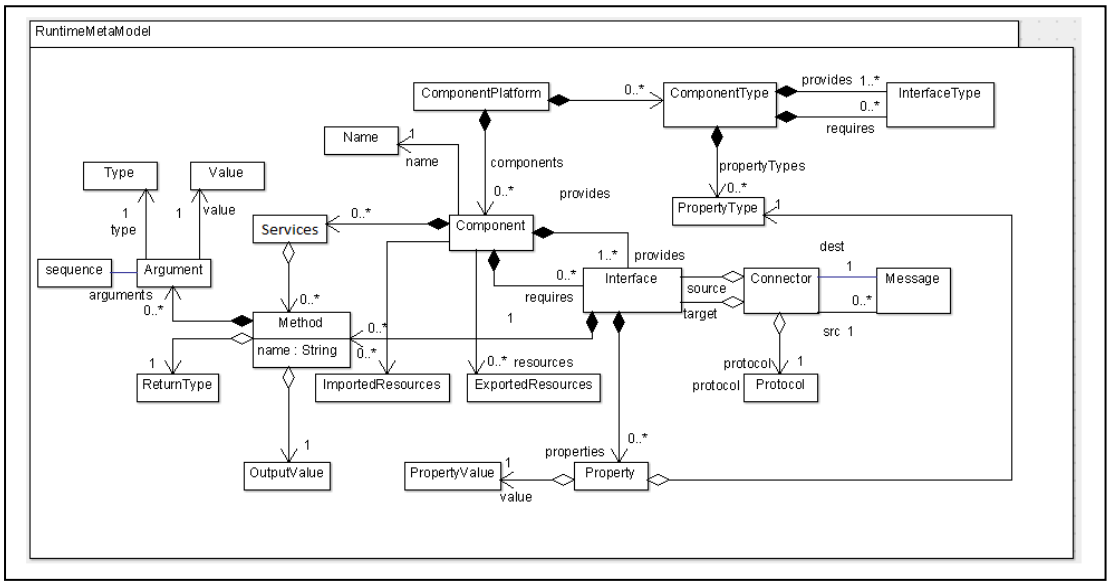

Figure 4. AIRTM: Application-independent Run-time Model.

2) Application-specific run-time model – ASRTM : It is a concrete model instance of AIRTM, which represents the concrete instances of the abstract concepts specified in the AIRTM. The abstract concepts are instantiated with the data collected by monitoring a system. Several monitoring mechanisms are described in section II such as reflection mechanism, run-time trace collection, log analysis, etc. We found runtime traces collection suitable for run-time model generation because traces represent rich semantic information about a system. Moreover, several past research exist on run-time traces and supporting tools that may provide traces in a specified syntax and semantics [30] [31] [32]. However, the framework is not bound to use only the run-time traces rather any method can be used for ASRTM instantiation. Ideally, the monitoring mechanism is responsible to collect the run-time data and instantiate the ASRTM.

3) Application-independent Mismatches Metamodel – AIMM : It is a metamodel that describes the relationshipsbetween the abstract concepts defined in AIRTM. Suchconcepts and their relationship present a mismatch template that can be further used to define a concrete mismatch. The concepts have been imported from AIRTM because those concepts are the building block of a system and the deviation in their values from a desired value results in a mismatch. A mismatch has a specific schema that is built upon the concepts responsible for the mismatch. The AIMM represents a set of schemata for a set of mismatches. These schemata assist to define the concrete mismatches. Several modeling languages exist, however the Java programming language has been used to implement the AIMM for simplicity. It is not bound to use Java language rather AIMM can be implemented in any language.

4) Application-specific Mismatches Model – ASMM : This model is an instance of AIMM that is instantiated by the data provided by the AKB. The AKB provides the data required to assess a situation that can cause a mismatch. This data is populated in AIMM that results in ASMM. A detailed description of ASMM and AIMM are given in Figure 5. AKB describes the domain knowledge that is usually provided by a subject matter expert (SME) or a domain expert who knows the details of the system. In our case, such an expert has to provide the data required to instantiate an ASMM.

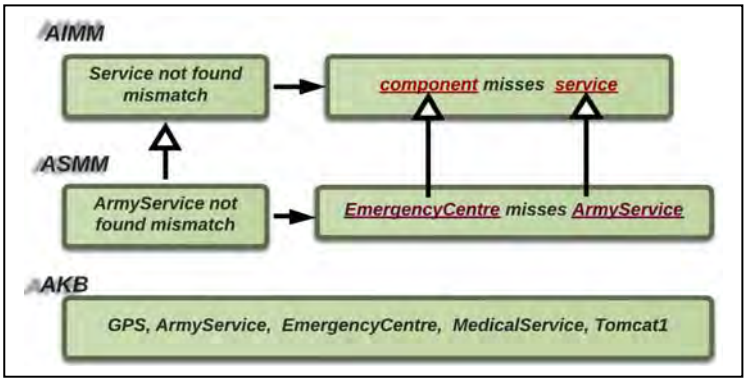

Figure 5. An example of AIMM, ASMM and AKB.

5) Mismatch Detection Engine – MDE : The MDE detects whether the condition represented by an ASMM holds true for a given ASRTM. When the ASMM condition holds true against the ASRTM, the corresponding mismatch is reported. For example, the MDE parses an ASMM 'EmergencyCentre misses ArmyService' and analyze that if ArmyService is unavailable to EmergencyCentre, the 'ArmyService not found mismatch' is detected. In order to assess this, the MDE looks for EmergencyCentre component's registered services in the given ASRTM. When the ArmyService does not exist as a registered service, the MDE confirms that 'ArmyService not found mismatch' exist in the EmergencyCentre component. Several research exist for such reasoning tools [33] [34] that can be used in our framework.

D. Methodology

The methodology of the A-MisTec framework is described in Figure 6. In this framework, the most substantial component is the AIRTM because ASRTM are a concrete instance of AIRTM and AIMM is based on the concepts of AIRTM. Therefore, the first step is to construct or adopt an existing AIRTM. Several models (discussed in section IV-A1) exist that can be used as an AIRTM. An AIRTM should be selected carefully to ensure that it has sufficient concepts to define the intended mismatches. For instance, workflow related mismatches cannot be constructed using a class diagram as an AIRTM. When the AIRTM is ready, get the list of its concepts. At the same time, prepare a list of mismatches to be detected [1] [35] [10]. The considered mismatches should be analyzed properly to understand its semantics. Then, the AIRTM's concepts should be used to define the AIMMs. The AIMMs can be developed in any language that is rich enough to define the mismatches. An SME is responsible to develop the AKBs that will be used to define the ASMMs. The ASMMs are the instances of AIMMs instantiated by the AKB data.

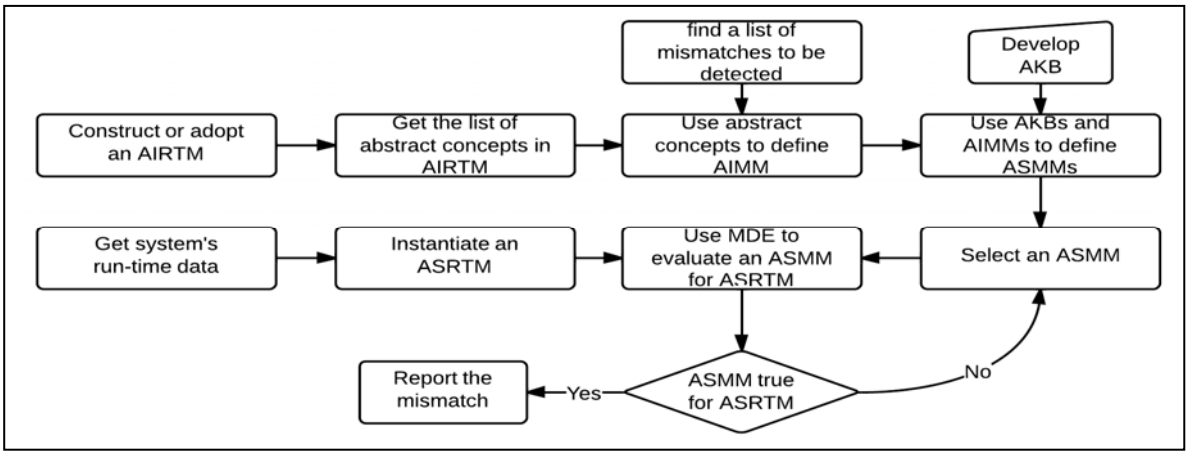

Figure 6. Flowchart of A-MisTec framework methodology.

In order to describe the run-time model of a system, a monitoring mechanism is required to gather the run-time data. A monitoring mechanism gathers the run-time data that is used to instantiate an ASRTM. An ASRTM is a concrete instance of AIRTM instantiated by the run-time data. When the AIRTM and ASMMs are instantiated, the MDE starts the mismatch detection mechanism. The MDE selects an ASMM and evaluates whether the selected ASMM holds true for a given ASRTM. The MDE reports a mismatch when an ASRTM holds true, otherwise it selects the next ASMM and continue with the mismatch detection mechanism. The MDE should investigate each ASMM for a given ASRTM. In a highly dynamic system, the ASRTM changes very often, thus the MDE requires a time efficient mechanism to investigate each ASMM for the frequently changing ASRTMs in a time efficient manner. The reported mismatches can be further resolved by manual interventions or using an adaptation mechanism.

V. IMPLEMENTATION

A-Mistec framework is developed in Java technologies; however it can be developed in any other language suitable to implement the models. The framework is primarily composed of five modules that are AIRTM, ASRTM, AIMM, ASMM and MDE developed in Java. The framework runs standalone that captures the run-time information from a monitoring mechanism. The AIRTM's concepts represent the basic building blocks of a system. These concepts are implemented as Java classes (see Listing 1). In a concept class, say Component, the associated concepts are described as attributes. Each class is provided with a static attribute of type java.lang.List to keep a list of all the instances of that class. For example, Medical, Army and EmergencyCentre components can be accessed through allInstances attribute of the Component class.

```
public class Component extends Concept{
    public List<Interface>  provides;
    public List<Interface>  requires;
    public List<ImportedResources>  myImportedResources;
    public List<Service>  Service;
    public List<ExportedResources>  exportedResources;
        public static List<Component> allInstances = new
ArrayList<Component>();
        public Component(String name, List<ImportedResources>
myImportedResources,
            List<ExportedResources>  exportedResources ){
        setName(name);
        this.myImportedResources = myImportedResources;
        this.exportedResources = exportedResources;
        this.Service = new ArrayList<Service>();
        this.provides = new ArrayList<Interface>();
        this.requires = new ArrayList<Interface>();
        allInstances.add(this);
    }
}
```

Listing 1. Description of Component concept of AIRTM.

The ASRTM is represented as a Java class that can be instantiated with the run-time information received from monitoring mechanism. An automatic mechanism in Java can be written that can collect the run-time information from monitor and instantiates an ASRTM. In an ASRTM, AIRTM's concepts are instantiated as described in listing 2. The described ASRTM instantiates a component 'EmergencyCentre' and adds a service 'MedicalService'. This data should ideally be provided by a monitoring mechanism, but it is out of scope of the paper, thus we have manually provided the run-time data to make the prototype experiment simpler.

In order to detect the mismatches in a system, a list of mismatches is required to be prepared. Several existing research list a set of mismatches [1] [35] [10], however 'service not found' mismatch is considered here for the demonstration purpose. A detailed analysis of this mismatch illustrates that its definition uses two concepts: (1) Component, and (2) Service. In consideration with this fact, an AIMM for the mismatch is described in listings 3. An AIMM can be generated automatically or manually, depending upon its complexity. Each AIMM includes an attribute 'type' that represents the type of a mismatch, e.g., 'ArmyService not found' mismatch is of type 'service not found' mismatch. The type plays an important role in deciding a recovery action for a mismatch [1]. When the AIMMs are prepared, an SME is required to create the ASMMs using the AIMMs and AKBs (see listing 4). A major limitation of this approach is whenever the AIRTM changes, the AIMMs and ASMMs are required to be updated. The AIMMs and ASMMs are described using XML language, however any other language can be used to define these models.

```
public class ASRTM {
public ASRTM(){

      List<ImportedResources>  myImportedResources =  new
      ArrayList<ImportedResources>();
      List<ExportedResources>  exportedResources =    new
      ArrayList<ExportedResources>();
      Component c =   new Component("EmergencyCentre",
      myImportedResources,   exportedResources);
      c.Service = new ArrayList<Service>();
      Service service = new Service("MedicalService");
      c.Service.add(service);
   }
}
```

Listing 2. Construction of ASRTM through run-time data.

```
<mismatch>
<type></type>
<name></name>
<component></component>
<service></service>
</mismatch>
```

Listing 3. An AIMM for 'Servivce not found' mismatch.

```
<mismatch>
<type>service-not-found</type>
<name>ArmyService-not-found</name>
<component>EmergencyCentre</component>
<service>ArmyService</service>
</mismatch>
<mismatch>
<type>service-not-found</type>
<name>MedicalService-not-found</name>
<component>EmergencyCentre</component>
<service>MedicalService</service>
</mismatch>
```

Listing 4. An ASMM for 'ArmyServivce not found' mismatch.

When the ASRTM and ASMMs are ready, the MDE module analyzes the mismatches in a system. A specific MDE is created for each type of mismatches. An MDE is a Java class that has a constructor and three methods: setType, setValues and isExist. For example, a Java class 'ServiceNotFoundMDE' is created to detect 'service not found' type mismatches (see listing 5). The isExist method describes the logical reasoning to detect a mismatch and outputs the mismatch status. The method setType specifies the type of a mismatch and setValues method sets the attributes values. An XML parser parses the ASMMs and calls setValues and setType method. For instance, when 'ArmyService not found' (listing 4) is parsed, the setValue method is called that sets component to 'EmergencyCentre' and service to 'ArmyService'. Then, the parser calls setType method that sets the type to 'service-not-found'. The isExist method return 'yes' when a component does not have a service that is defined in the service attribute of the ServiceNotFoundMDE class. In consideration with the given ASRTM (listing 2), the ServiceNotFoundMDE is instantiated for both 'ArmyService not found' and 'MedicalService not found' mismatch. When the isExist method is called for each mismatch, only 'ArmyService not found' mismatch is detected for the 'EmergencyCentre' component. This is because the ASRTM describes that 'EmergencyCentre' has 'MedicalService' but not the 'ArmyService'. This process will go in loop for each ASMMs defined and finally presents a list of detected mismatches in a system.

```
public class ServiceNotFoundMDE extends MismatchMDE{
	String component = "";
	String service = "";
	@Override
	public void setValues(String... args) {
		component = args[0];
		service = args[1];
	}
	@Override
	public STATUS isExist() {
	Iterator<Component> itC = Component.allInstances.iterator();
	while(itC.hasNext()){
		Component c = itC.next();
		if(c.getName().equals(component)){
		Iterator<Service> itServices = c.Service.iterator();
				while(itServices.hasNext()){
					Service s = itServices.next();
					if(s.getName().equals(service)){
						return STATUS.NO;
					}}
				return STATUS.YES;}}
		return STATUS.UNDETERMINED;
	}
	@Override
	public void setType() {this.type = "service-not-found";}
	public ServiceNotFoundMDE(String instance{super(instance);}
}
```

Listing 5. MDE: Implementation of ServiceNotFoundMDE to detect mismatches of type service-not-found.

VI. EVALUATION

Several mismatch detection techniques exist (described in section II), however they are not generic and likely to be implemented from scratch for each system. The re-usable infrastructure for self-adaptation and run-time modeling exist [36] but no re-usable mismatch-detection infrastructure exist. Re-usable infrastructures are more effective than system-specific because they can be extended to work across a variety of systems [36]. The A-MisTec framework is evaluated for the re-usability, a variety of mismatches handled and effort required. The framework is customized for the following two systems. A simulator has been designed to automate the framework. In order to make the evaluation simpler, 'service not found' mismatch is detected in the following two systems.

A. Crisis Management System

In the CMS (discussed in section 2), when a flood occurs, the EmergencyCentre component carry out the evacuation process using MedicalService and ArmyService. The simulator detects a mismatch only by giving the run-time information and mismatch definition as ASMM. The simulator loads the pre-defined AIRTM and AIMM, and uses the given run-time information and ASMM, to detect a mismatch. The simulator is able to identify 'ArmyService not found' mismatch as described in Figure 7.

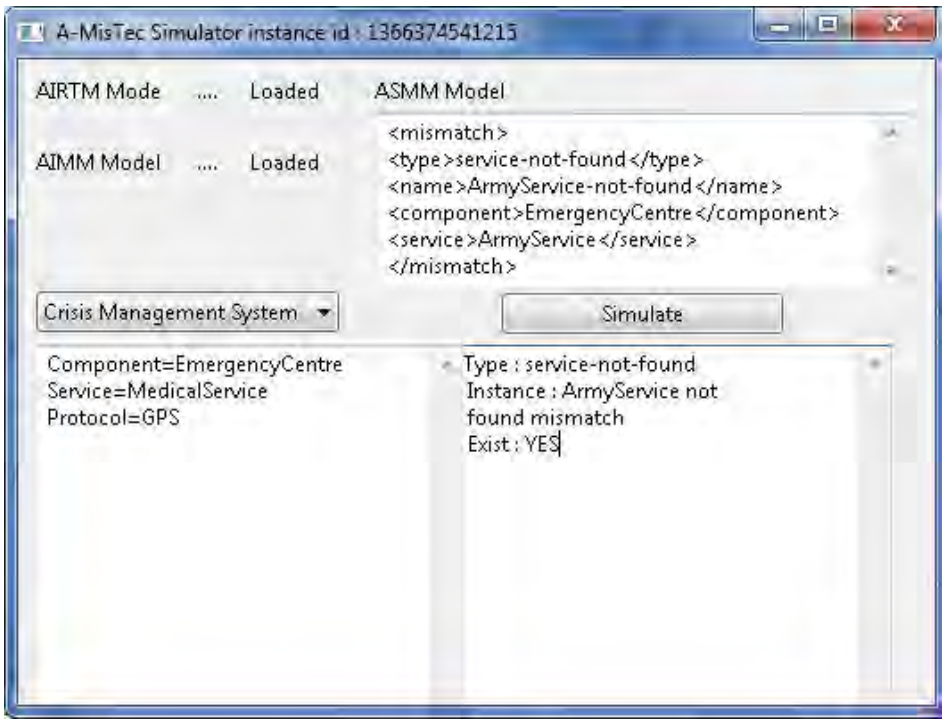

Figure 7. A simulation excerpt of CMS.

B. Smart Office

A smart office [37] enables a communication between two office employees. This is an adaptive system that senses an employee's location and sends a message in appropriate format. For example, when an employee is driving, a voice message is delivered whereas an email message is delivered when the employee is at his office desk. The system uses several services such as voice mail service, email service, etc. When email service is not available, the system reports 'EmailService not found' mismatch that is illustrated by our simulation. In order to find this mismatch, the AIRTM and AIMM remain same as their concepts represent the system's building blocks. The run-time information and 'EmailService not found' mismatch definition is given as input to the simulator to detect the mismatch (see Figure 8).

The above two simulations detect the mismatches of type 'service-not-found' in two different system. In our framework, the only input required was run-time information and mismatch definition. The table I describes a comparison of our approach with existing mismatch detection approaches. The first column describes the name of the approach, second describes the mismatches that can be detected by the approach, third describes how much and what type of effort is required when the same approach is applied in different systems and fourth column describes whether a framework developed on the basis of the approach, can be used for different systems. Egyed et al. [11] detect only component mismatches and its application across systems. This approach, when applied for another system, requires many changes as several features of a system are required to be re-defined. Gacek [12] detects architectural mismatches and seems to be inefficient to apply approach across system having different

architectures. Xie et al. [8] detect only component signature-level mismatches, however the framework developed on this approach seems to be re-usable across systems having the same components' definitions. The approach does not work when a component's definitions does not match with their component's definitions. Nickelsen et al. [7] find the network mismatches through the use of Bayesian Belief network (BBN). The approach collects cross-layer data and instantiates BBN. The generation and learning of BBN is a computational intensive process. Ecoware [21] is an event correlation framework that can be re-used across systems. However, it needs a huge set of rules and Key performance indicator (KPIs) to evaluate a mismatch. Considering existing techniques, our A-MisTec framework seems to work across several domains and systems that can detect almost all known mismatches. In terms of efforts, only ASMMs are required to be generated for different systems.

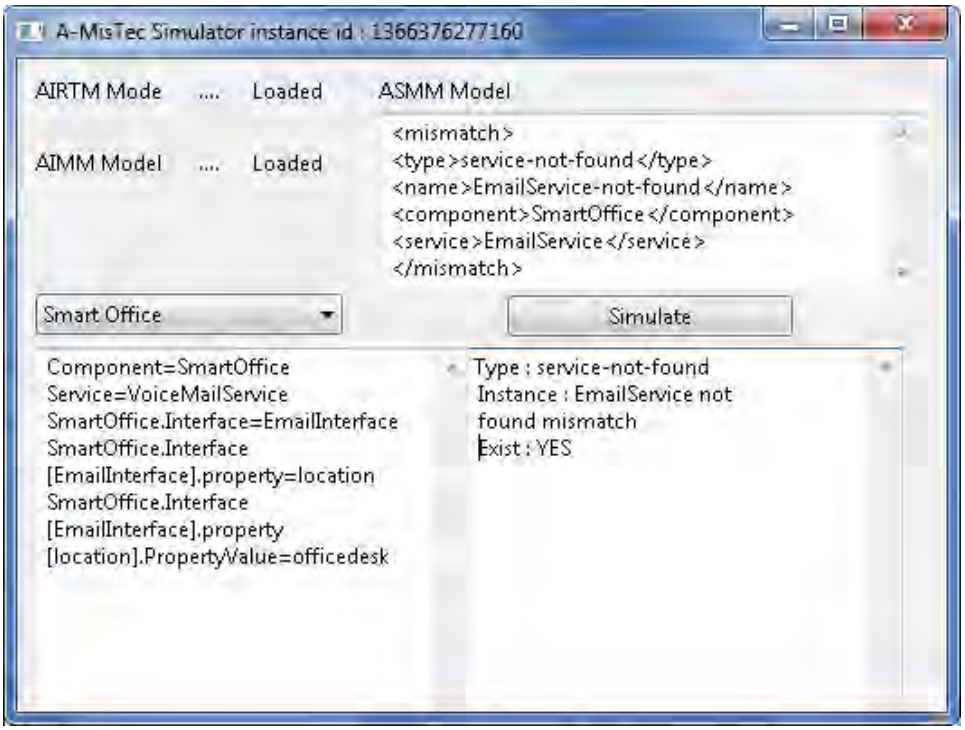

Figure 8. A simulation excerpt of Smart Office.

TABLE I. COMPARATIVE ANALYSIS

Approach	Mismatches	Effort	Re-usable Framework
Egyed et al. [11]	Component mismatches	More than 100 LOC	No
Gacek et al. [12]	Architectural mismatches	More than 100 LOC	No
Xie et al. [8]	Component signature mismatches	More than 100 LOC	Yes
Nickelsen et al. [7]	Network mismatches	Generate BBN	Networks specific
Ecoware [21]	Event mismatches	Huge KPIs	Yes
A-MisTec	Any mismatch	Less than 100 LOC	Yes

VII CONCLUSION

Large-scale systems encounter several problems while designing, publishing, composition and run-time. Mismatches emerged as a class of problems that represent a deviation from a desired structure or behavior of a system. The detection of mismatches is not trivial due to complex nature of the systems. Several mismatch detection techniques exist, however they are not sufficient as developers still have to dig out the exact mismatch in a system. The paper categorized the mismatches into two categories: (1) application-independent mismatches that represent mismatches in abstract concepts such as component, service, etc., and (2) application-specific mismatches that represent mismatches in specific instances of a system such as a Bank component, transaction service, etc. The latter one is capable to pinpoint the exact problem and its location in a system, thus developer need no further diagnosis.

In order to detect application-specific mismatches, a model-driven framework is presented. The framework leverages the use of run-time models to detect the mismatches. A subject matter expert defines the mismatches and put them into a mismatch repository. The framework's mismatch detection engine takes each mismatch from the pool and assesses its existence in the run-time model. When a mismatch exists in a run-time model, the engine reports the detected mismatch to developer for further recover actions. The framework is compatible with the evolving systems where new entities may be added in the future. In case of evolving systems, the framework requires no changes rather developer/SME has to define new mismatches corresponding to new entities added and

a corresponding mismatch detection rule has to be added in the detection engine. The framework automatically analyzes the new entities and new mismatches declared, and detects if any mismatch exists. A major limitation of the framework is the dependency upon the AIRTM. When the AIRTM changes, all the models are required to be updated, however such cases occur rarely. The future work addresses this limitation and attempts to integrate an automatic monitoring mechanism with the framework.

ACKNOWLEDGMENT

The project is supported by Trinity College Postgraduate Award fund. A special thanks to Prof. Siobhan Clarke, Dr. Hui Song, and Saeed Hajebi for useful discussions and suggestions.

REFERENCES

[1] R. Popescu, A. Staikopoulos, P. Liu, A. Brogi, and S. Clarke, "Taxonomy-driven adaptation of multi-layer applications using templates," in Proceedings of the 2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, ser. SASO '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 213–222. [Online]. Available: http://dx.doi.org/10.1109/SASO.2010.23J.

[2] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, "Towards an engineering approach to component adaptation," Architecting Systems with Trustworthy Components, vol. 3938, pp. 193– 215, 2006. [Online]. Available: http://dx.doi.org/10.1007/11786160 11

[3] S. Bruning, S. Weissleder, and M. Malek, "A fault taxonomy for service-oriented architecture," in High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE, nov. 2007, pp. 367 –368.

[4] C. Canal, P. Poizat, and G. Salan, "Synchronizing behavioural mismatch in software composition," Lecture Notes in Computer Science, vol. 4037, pp. 63–77, 2006. [Online]. Available: http://cat.inist.fr/?aModele=afficheN&cpsidt=20039756

[5] A. Egyed, N. Medvidovic, and C. Gacek, "Component-based perspective on software-mismatch detection and resolution," Software, IEE Proceedings -, vol. 147, no. 6, pp. 225 –236, 2000.

[6] C. Gacek, "Detecting architectural mismatches during systems composition," University of Southern California, Los Angeles, CA, USA, Tech. Rep., 1998.

[7] A. Nickelsen, J. Gronbaek, T. Renier, and H.-P. Schwefel, "Probabilistic network fault-diagnosis using cross-layer observations," in Advanced Information Networking and Applications, 2009. AINA '09. International Conference on, may 2009, pp. 225 –232.

[8] X. Xie and W. Zhang, "A checking mechanism of software component adaptation," in Proceedings of the Fifth International Conference on Grid and Cooperative Computing, ser. GCC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 347–354. [Online]. Available: http://dx.doi.org/10.1109/GCC.2006.2

[9] G. Blair, N. Bencomo, and R. France, "Models@ run.time," Computer, vol. 42, no. 10, pp. 22 –27, oct. 2009.

[10] X. Li, Y. Fan, and F. Jiang, "A classification of service composition mismatches to support service mediation," in Proceedings of the Sixth International Conference on Grid and Cooperative Computing, ser. GCC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 315–321. [Online]. Available: http://dx.doi.org/10.1109/GCC.2007.1

[11] A. Egyed, N. Medvidovic, and C. Gacek, "Component-based perspective on software-mismatch detection and resolution," Software, IEE Proceedings -, vol. 147, no. 6, pp. 225 –236, 2000.

[12] C. Gacek, "Detecting architectural mismatches during systems composition," Ph.D. dissertation, Los Angeles, CA, USA, 1998, aAI9931882.

[13] G. S. Carlos Canal, Pascal Poizat, "Synchronizing behavioural mismatch in software composition," in Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems FMOODS, June 2006.

[14] C. Canal, P. Poizat, and G. Salaun, "Model-based adaptation of behavioral mismatching components," Software Engineering, IEEE Transactions on, vol. 34, no. 4, pp. 546 –563, july-aug. 2008.

[15] O. Moser, F. Rosenberg, and S. Dustdar, "Viedame - flexible and robust bpel processes through monitoring and adaptation," in Companion of the 30th international conference on Software engineering, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 917–918. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370186

[16] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, "Run-time monitoring of instances and classes of web service compositions," in Web Services, 2006. ICWS '06. International Conference on, 2006, pp. 63–71.

[17] K. Mahbub and G. Spanoudakis, "Monitoring ws-agreements: An event calculus based approach," in In Test and Analysis of Web Services, (eds) Baresi L. & di Nitto E. Springer Verlang, 2007, pp. 265–306.

[18] G. Spanoudakis, "Non intrusive monitoring of service based systems," International Journal of Cooperative Information Systems, vol. 15, pp. 325–358, 2006.

[19] A. D. H. Farrell, M. J. Sergot, D. Trastour, and A. Christodoulou, "Performance monitoring of service-level agreements for utility computing using the event calculus," in Proceedings of the First IEEE International Workshop on Electronic Contracting, ser. WEC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 17–24. [Online]. Available: http://dx.doi.org/10.1109/WEC.2004.15

[20] S. Guinea, G. Kecskemeti, A. Marconi, and B. Wetzstein, "Multi-layered monitoring and adaptation," in Proceedings of the 9th international conference on Service-Oriented Computing, ser. ICSOC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 359–373. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25535-9 24

[21] "Ecoware (event correlation middleware)," http://www.maurocaporuscio.net/index.php/software.

[22] J. C. Munson and A. P. Nikora, "Toward a quantifiable definition of software faults," in Proceedings of the 13th International Symposium on Software Reliability Engineering, ser. ISSRE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 388–. [Online]. Available: http://dl.acm.org/citation.cfm?id=851033.856288

[23] H. Song, A. Raj, S. Hajebi, S. Clarke, and A. Clarke, "Model driven engineering of cross-layer monitoring and adaptation," in Conjunction with the International Conference on Model-Driven Engineering and Software Development - MODELSWARD, Barcelona, Spain, 2013.

[24] OMG, "Omg uml infrastructure specification," 2001, http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF.

[25] D. Chappell, "Introducing sca," 2007, white paper, Chappell & Associates.

[26] T. Vogel and H. Giese, "Adaptation and abstract runtime models," in Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, ser. SEAMS '10. New York, NY, USA: ACM, 2010, pp. 39–48. [Online]. Available: http://doi.acm.org/10.1145/1808984.1808989

[27] J. Q. Ning, "A component model proposal," in Proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE'99, May 1999.

[28] K.-k. Lau and M. Ornaghi, "A formal approach to software component specification," Components, no. Idl, pp. 88–96, 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.7710

[29] C. Geisterfer and S. Ghosh, "Software component specification: a study in perspective of component selection and reuse," in Commercial-offthe-Shelf (COTS)-Based Software Systems, 2006. Fifth International Conference on, feb. 2006, p. 9 pp.
[30] S. Maoz, "Using model-based traces as runtime models," Computer, vol. 42, no. 10, pp. 28–36, 2009.
[31] "Models in software engineering," M. R. Chaudron, Ed. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Model-Based Traces, pp. 109–119. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01648-6 12
[32] S. Jiang, H. Zhang, Q. Wang, and Y. Zhang, "A debugging approach for java runtime exceptions based on program slicing and stack traces," in Quality Software (QSIC), 2010 10th International Conference on, 2010, pp. 393–398.
[33] B. Parsia and E. Sirin, "Pellet: An owl dl reasoner," in 3rd International Semantic Web Conference (ISWC2004), 2004.
[34] P. Klinov, "Pronto: A non-monotonic probabilistic description logic reasoner," in The Semantic Web: Research and Applications, ser. Lecture Notes in Computer Science, S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, Eds. Springer Berlin Heidelberg, 2008, vol. 5021, pp. 822–826. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68234-9 66
[35] C. Zeginis, K. Konsolaki, K. Kritikos, and D. Plexousakis, "Ecmaf: an event-based cross-layer service monitoring and adaptation framework," in Proceedings of the 2011 international conference on Service-Oriented Computing, ser. ICSOC'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 147–161.
[36] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," Computer, vol. 37, no. 10, pp. 46 – 54, oct. 2004.
[37] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," Computer, vol. 42, pp. 44–51, October 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.327