2017 International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), Porto, Portugul, 19-21 February 2017.

Testing Environment Emulation - A Model-based Approach

Jian Liu[1], John Grundy[2], Iman Avazpour[2], and Mohamed Abdelrazek[2]

[1]School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia [2] Deakin University, School of Information Technology, Burwood, VIC 3125, Australia

jianliu@swin.edu.au, {j.grundy, iman.avazpour, mohamed.abdelrazek}@deakin.edu.au

Keywords: Model-driven engineering, domain-specific visual modeling language, software interface description framework, testing environment emulation.

Abstract: Modern enterprise software systems often need to interact with a large number of distributed and heterogeneous systems. As a result, integration testing has become a critical step in their software development lifecycle. Service virtualization is an emerging technique for creating testing environments with realistic executable models of server side production-like behaviours. However, building models in existing service virtualization approaches is very challenging, requiring either significant human effort or the availability of interactive tracing records. In this paper, we present a domain-specific modeling approach to generate complex, virtualized testing environments. Our approach allows domain experts to use a suite of domain-specific visual modeling languages to model key interface layers of applications at a high level of abstraction. These layered models are then transformed into a testing runtime environment for application integration testing. We have conducted a technical comparison with two other existing approaches and also carried out a user study. The user study demonstrated the acceptance of our new testing environment emulation approach from software testing experts and developers.

1 INTRODUCTION

Testing environment emulation provides integration testing to an enterprise System Under Test (SUT) that interacts with many external systems. Currently, there are two kinds of approaches to develop such integration testing environments. Specification-based approaches are used by IT professionals to develop simplified versions of applications with external behaviour only (often called “endpoints”) manually (Hine et al., 2009, Yu et al., 2012). They perform this using available knowledge of the underlying message syntax, interaction protocol and system behaviour. Interactive tracing data record-and-replay based approaches (called “interactive tracing” hereafter) create endpoint models from recorded request-response pairs between the endpoint system and an earlier version of a SUT automatically (Du et al., 2013). Each endpoint’s simulated response is generated by finding a close-match request in the previously recorded trace database.

Both approaches have their strengths but also shortcomings. Specification-based approaches have high development and set-up cost and require access to a detailed system specification and/or implementations, if available. Interactive tracing approaches depend on the availability of trace records for all integration testing cases between a SUT and its operational environment. In recent years, the interactive tracing approaches are getting more popular, and many major players have released their commercial products (Giudice, 2014). However, these products need to have a complementary specification-based development tool for modeling those endpoints, which do not have all interactive tracing data available.

Aiming to achieve high development productivity and ease of use for domain experts, we have developed a novel specification-based Domain-Specific Modeling (DSM) approach for testing environment emulation. Our approach is based on model-driven engineering, where users work on high level abstraction models and executable code will be generated automatically by transforming these models using code generators. Our DSM approach divides software interfaces into different abstraction layers, where each layer represents a modeling problem domain – e.g. endpoint signature, protocol and behaviour layers. The approach introduces a suite of Domain-Specific Visual Modeling Languages (DSVLs) for Testing Environment Emulation

(TeeVML) (Liu et al., 2016), each is dedicated to a specific interface layer. Users use TeeVML to model testing endpoints by layers. A testing runtime environment is then provided by Axis2 Web Service platform (Jayasinghe, 2008), together with the Java code generated automatically from endpoint models.

The key contributions of this paper include: (1) a solution to model endpoint external behaviours for enterprise application integration testing, (2) a software interface description framework to abstract testing endpoints into multiple logic layers, (3) and a model-driven, domain-specific approach to generate emulated testing endpoints using our TeeVML toolset. The scope of our approach is for emulating complex interactions between a SUT and an endpoint. Thus other interactions behind the endpoint for providing composite services are not considered in this study. The applications include those using Remote Procedure Call (RPC) communication style and with stateful session management as a typical business scenario. However our approach can be generalized to cover a large number of real-world situations.

The remainder of this paper is organized as follows: Section 2 motivates our work with an example case study, followed by an introduction of our approach in Section 3. In Section 4, we show how an endpoint is modeled and then describe the steps to convert endpoint models into testing runtime environment. In Section 5, we evaluate our approach and discuss the key findings from the results of a technical comparison and a user survey. This is followed by a review of related work in section 6. Finally, we conclude this paper and identify some key future works in Section 7.

2 MOTIVATION

Assume a company has an in-house Enterprise Resource Planning (ERP) system to support its daily operations. For the purpose of streamlining its sales process and improving internal efficiency, the company plans to introduce a public cloud-hosted Customer Relationship Management (CRM) application. To ensure the interconnectivity and mutual operability between the ERP system and CRM application, integration testing must be conducted before putting the CRM in production. For this study, we treat the cloud CRM as the SUT, and the ERP as the testing endpoint to be emulated.

The sequence diagram in Figure 1 illustrates a typical purchase process, where a sales representative uses the CRM application to place a Purchase Order (PO) for his/her client. Our main interest is on the interactions between the testing endpoint and the SUT, which represent the endpoint protocol behaviours. We describe the interactive behaviour between the CRM and ERP below.

Whenever the endpoint receives a logon request (#1) from its SUT, it transits from Idle state to Home state and an interactive session starts. The next valid operation is a PO request (#2), and followed by an inventory check (#3). The returned value of the inventory check will determine whether or not supplier chain related steps will be executed. If the purchase item has enough stock for the PO, the process flow will jump over those supplier purchasing steps and directly go to a payment request (#8) state. Otherwise, we have to go through all supplier purchase steps (#4, #5, #6 and #7) to buy the missing quantity of the PO item. Supplier PO approval (#5) and approval notification (#6) are iteration operations, informing all approvers one-by-one to give his/her approval. If all required approvals for the supplier PO have been obtained, the rest of purchasing steps will be executed in the order as in Figure 1. Otherwise, the purchase process will be aborted without success.

In addition, there are some other important protocol behaviours: (1) Timeouts – a timeout event will automatically terminate an interactive session and the endpoint state will be changed from Home to Idle, if no valid operation request is received within a defined period of time; (2) Synchronous operation simulation – if an endpoint operation is in synchronous mode, all further operation requests will be rejected when it is processing the operation; and (3) Unsafe operation (i.e. not an idempotent operation that will produce the same result if executed once or multiple times) simulation – the payment request (#8) is considered to be an unsafe operation, and multiple requests for a same payment request operation are not allowed.

It is infeasible to test the CRM with the production ERP system, and there is large cost involved in duplicating the ERP. Conventional interactive tracing data and specification-based approaches are similarly infeasible or difficult to use, as the former relies on existing interactive tracing data and the latter requires development of detailed endpoint model implementations.

Just as any other software development tools, users' primary concerns about our endpoint modeling approach will be: what can it do for their service emulation modeling and generation, will it improve endpoint development productivity, how easily can it be used.

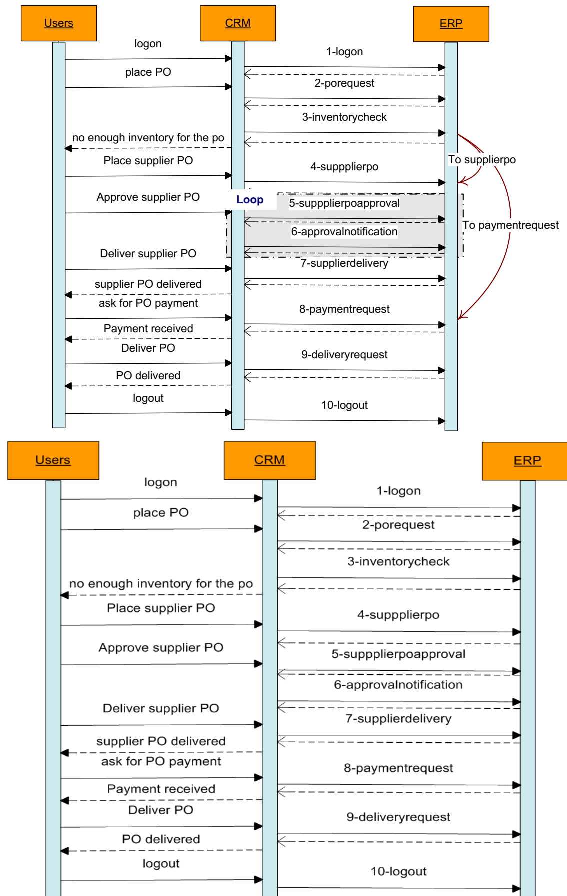

Figure 1: A CRM and an ERP sales process flow diagram.

Therefore, we have defined the following three research questions to be addressed by our approach described in this paper:

1) RQ1 – Can we emulate an integration testing environment capable of detecting interface defects of an existing or a non-existing system under test from a high level service model?
2) RQ2 – Would such a model-based approach improve testing environment development productivity, compared to using third generation languages or specification-based manual coding approaches?
3) RQ3 – Can we develop a user centric approach, easy enough to learn and use for specifying testing endpoints by domain experts?

3 OUR APPROACH

A testing endpoint is a server-side application, receiving, validating and processing operation requests from a SUT. Our goal is to make the emulated testing environment rich enough to "fool" the SUT that it is talking to the real system. Specifically, an endpoint is a simplified version of its real system with three assumptions: (1) only external behaviours of the real system are considered and all internal implementations will be ignored; (2) subset operations of the real system invoked by the SUT are provided; and (3) all SUT interface defects, together with their types and origin information, should be able to be detected and reported.

3.1 Domain Analysis

To identify endpoint common entities and find out their relationships, we conducted our testing environment emulation domain analysis by investigating three applications interacting with their clients. These applications included the ERP system introduced in Section 2, a LDAP server and an e-commerce application. These applications represent a variety of application domains in a typical enterprise environment. The domain analysis focused on two areas: the interaction abstraction between a service provider and a service consumer, and the requirement on integration testing environment. From the domain analysis, we proposed a layered software interface description framework for testing environment emulation, and defined interface defect types to be detected by endpoints. Consecutively, we designed our modeling approach for each interface layer. The detailed design of the TeeVML visual notations are out of scope for this paper. Interested readers can refer to our previous publication (Liu et al., 2016).

3.2 Software Interface Description Framework

Our new layered software interface description framework builds on top of Han's comprehensive interface definition framework for software components (Han, 2000). Our framework logically separates software interfaces into three horizontal and two vertical layers. Horizontal layers include signature, protocol and behaviour. Vertical layers include data store (data persistence) and Quality-of-Service (QoS) (or call non-functional requirement). A SUT operation request is processed horizontally by an endpoint step by step from signature, protocol, down to interactive behaviour layer. Whenever an error occurs at any layer, the request process will be terminated.

The signature and protocol layers act as message pre-processors for checking the correctness of an

operation request syntax and temporal sequence, before handing it over to the behaviour layer for generating response. Vertical layers are not directly involved in request processing, but provide support to horizontal layers. We use a modular development approach to model an endpoint – i.e. each module represents a particular interface layer.

In this paper, we describe our approach to model endpoint functional layers. The data store layer is integrated to the behaviour layer, and integrating the Quality of Service requirements is part of our future work.

3.3 Interface Defects

Corresponding to endpoint horizontal and vertical layers, there are also two types of interface defects: functional defects, which are directly related to operation request processing, such as incorrect message signature and invalid operations; and non-functional defects, such as non-compliance with endpoint security policy. Table 1 lists all the functional defect types a SUT operation request may contain.

Table 1 does not include any behaviour defect types. This is because a SUT's obligation is to send correct operation requests to endpoint and the way these requests are to be processed is defined in the endpoint's internal implementation. The reason why we still model endpoint behaviour is that the validity of alternative requests may depend on what values are returned in a response message it has received for a previous operation request (refer to P3 defect type of Table 1). Below we discuss how these request defect types can be detected.

Table 1: Operation request defect types.

No	Defect Type Description
	Signature
S1	An operation request is not an operation provided by endpoint.
S2	The parameters in an operation request are not matched with the parameters of the corresponding operation provided by endpoint, in terms of parameters' name, data type and order in the operation request.
S3	One or more operation request mandatory parameter(s) is (are) missing.
S4	One or more parameter(s) in an operation request is (are) beyond the defined value range of the corresponding endpoint operation.
	Protocol
P1	An operation request is invalid for the current endpoint state.
P2	An operation request is invalid for the current endpoint state, as one or more parameter(s) violate(s) the defined constraint condition(s).
P3	An operation request is invalid for the current endpoint state, as one or more returned value(s) from a previous operation request violate(s) the defined constraint condition(s).
P4	An operation request is invalid, due to endpoint state transition driven by some internal event, such as time out.
P5	An operation request is invalid, as endpoint is in processing a synchronous operation request.
P6	An operation request is invalid, as one such request for an unsafe operation has been received by endpoint.

3.4 Signature Modeling Approach

Endpoint signature is modeled by a signature DSVL. We used Web Service Definition Language (WSDL) 1.1 as its model and adopted a three-level architecture design (refer to Figure 2). The top-level WSDL DSVL (refer to Figure 2a) is used to define the five WSDL entity types: Service, Port, Binding, PortType and Operation. To link these entities together, we added two relationships: Composition and Association. The middle-level operation DSVL (refer to Figure 2b) is used to define request and/or response message(s) contained in an operation. The bottom-level message DSVL (refer to Figure 2c) is based on the W3C XML Schema 1.1 for defining complex elements in a message. By using the multi-level modeling approach, lower level models can be reused by upper level models.

The benefits from using WSDL specification as our signature DSVL metamodel include: (1) WSDL supports RPC communication style, covering a wide range of endpoint signature types; (2) we can use Axis2 wsdl2java utility to generate Axis2 Web Service engine as our domain framework automatically; and (3) Axis2 engine provides a signature defect detection mechanism from a WSDL file definition. There are some open-source or commercial WSDL tools available, such as Eclipse WTP Plugin and XMLSpy. The motivation for developing our own WSDL tool is to increase the consistency among different parts of TeeVML. Behaviour model imports operations and their parameters from the corresponding signature model; and message DSVL is reused to define data store model.

The signature defects S1 to S3 in Table 1 can be detected by the Axis2 Web Service engine. For S4 defect debugging, two fields are added to the element type of message DSVL for specifying the minimum and maximum values of a request parameter.

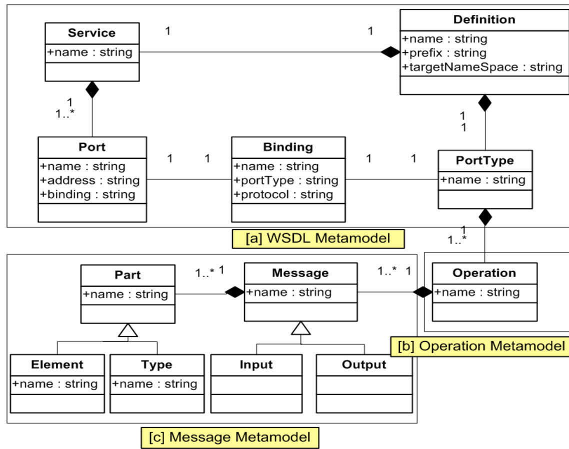

Figure 2: Signature metamodel.

3.5 Protocol Modeling Approach

A standard operation-driven Finite State Machine (FSM) is often used to represent endpoint protocol behaviour (Hine et al., 2009). To deal with incomplete protocol specification problems and capture runtime constraints, we used an Extended Finite State Machine (EFSM) to enrich our protocol modeling capability with dynamic protocol aspects. Our EFSM adds one entity type and two entity properties (marked yellow in Figure 3). The entity type is the *InternalEvent*, which is used to define state transitions triggered by a time event. One of the entity properties is the *StateTransitionConstraint* of transition entity, and it is for specifying either static or dynamic constraints on state transition function. Another one is the *StateTimeProperty* of state entity, which allows users to simulate synchronous and unsafe operations. As endpoint protocol modeling is relatively simpler than the other two layers, we use a flat view presentation structure.

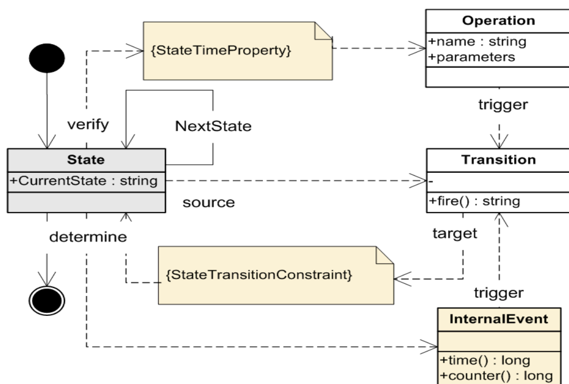

Figure 3: Protocol metamodel.

All the protocol defect types listed in Table 1 can be detected by a testing endpoint, developed by a modeling tool based on the EFSM model: (1) P1 – the operation-driven state transition FSM; (2) P2 and P3 – the *StateTransitionConstraint* property of transition entity; (3) P4 – the *InternalEvent* entity; and (4) P5 and P6 – the *StateTimeProperty* of state entity.

3.6 Behaviour Modeling Approach

Endpoint behaviour DSVL was designed based on the dataflow programming paradigm (Sousa, 2012). We chose this metaphor as it allows complex specification of behaviour models but is understandable by a wide range of software stakeholders. The dataflow programming execution model is represented by a directed graph; the nodes of the graph are data processing units, and the directed arcs between the nodes represent data dependencies. Data flow in each node from its input connector; and the node starts to process and convert the data whenever it has the minimum required parameters available. The node then places its execution results onto its output connector for the next nodes in the chain. Data store operators are used to access and manipulate persistent data.

To handle complicated business logics, we designed our behaviour DSVL using hierarchical structure. The benefits from using the hierarchical structure are two-fold: First, we can reuse some of the nodes, if they perform exactly the same task but are located at different components. Second, it can help us to manage diagrammatic complexity problem. At the bottom level, a node consists of some primitive visual constructs for performing operations on data and flow controls. At the top level, there are discrete service nodes to represent all operations provided by an endpoint. To prevent the data inconsistence between behaviour model and signature model, each of the service nodes imports its request and response parameters from the same endpoint signature model.

3.7 Testing Runtime Environment

Our testing runtime environment is generated by transforming the endpoint models using a set of code generators. There are four code generators: (1) a signature code generator to transform an endpoint signature model to a WSDL file, (2) a protocol code generator to transform an endpoint protocol model to a protocol processing class, (3) a behaviour code generator to transform an endpoint behaviour model to several behaviour model classes for handling all operation requests from a SUT, and (4) a data store

code generator to transform data store and operation models to JDBC classes.

An Axis2 Web Service engine, generated by Axis2 wsdl2java utility, binds both server side and client side implementations to a service contract defined by a signature WSDL file. On the server side, Axis2 provides a skeleton class as interface for integrating business logic processing Java classes. On the client side, Axis2 also has a stub class for allowing client to access the server operations. We developed Java API classes for facilitating the integration with SUTs. To provide the integration testing service to a SUT, Axis2 service must be placed into a servlet container. We use Tomcat 7.0 as our Web application server. Figure 4 illustrates our integration testing runtime environment, where a SUT is on the right-hand side and a testing endpoint is on the left-hand side. They interact with each other by SOAP over HTTP protocol.

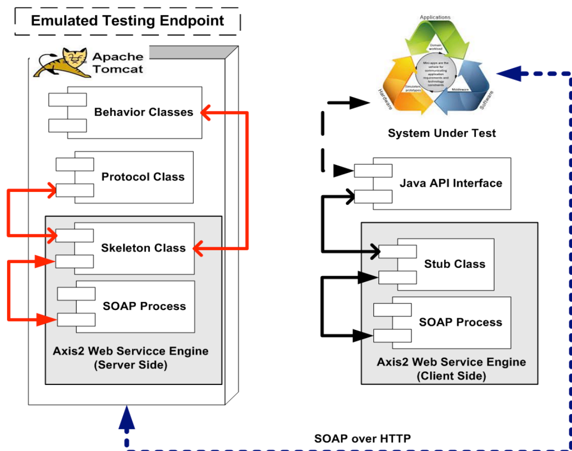

Figure 4: Testing runtime environment deployment view.

4 USAGE EXAMPLE

We use the ERP system from Section 2 as an example to show how a testing endpoint is modeled by use of TeeVML and how an integration testing runtime environment is built.

4.1 Signature Modeling

We start signature modeling by specifying the five WSDL entity types using WSDL DSVL: Service, Port, Binding, PortType and Operation. Then, we link them together by using either a composition or an association relationship. All the entity types are instantiated by providing their names. In addition, the operation has a pattern property with a value of "in/out", "in-only" or "out-only"; and the port must specify the endpoint service address.

We use an operation named *paymentrequest* as an example to show how operations can be modeled. The operation contains *paymentrequest_request* and *paymentrequest_response* messages, and they are defined by operation DSVL. The message label is "in" for the request message and "out" for the response message. The elements in the request and response messages are defined by using message DSVL. The request message contains only one element *pono* (purchase order number), and it is defined as integer and mandatory. Since a valid *pono* is a five-digit integer, the element's minimum field is specified as 10000 and maximum field as 99999. The response message consists of three elements: *amount*, *errorcode* and *errormessage*. They are placed in the message in alphabetic order. The *amount* is a float data type, *errorcode* integer and *errormessage* string. Figure 5 illustrates the hierarchical signature model of the ERP system endpoint. It contains the top-level WSDL model (refer to Figure 5a; for a better view representation, we only show five operations), the middle-level *paymentrequest* operation model (Figure 5b), and the bottom-level request and response message models (Figure 5c).

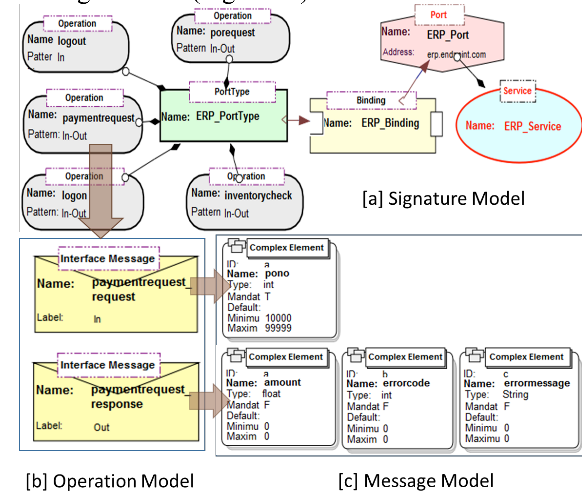

Figure 5: Example signature model.

4.2 Protocol Modeling

We use protocol DSVL to model the testing endpoint protocol layer. Figure 6 illustrates the endpoint protocol model, where the emulated enterprise purchase process flows in clockwise direction. To explain how the endpoint protocol is modeled, we select three typical protocol behaviours of interactive session management, constraint state transition and

transition iteration. They are marked as A, B and C in the diagram, respectively.

A -- Session management: Endpoint protocol modeling starts from specifying an interactive session by using a logon transition relationship from Idle state to Home state. On the opposite direction, a logout transition relationship terminates a session. A session can also be terminated by a timeout event, which is defined by using a timeout relationship linking a *from* state to a *to* state.

B – Constraint transition relationship: When the endpoint is at *inventorycheck* state, there are alternative flows either to *supplierpo* or to *paymentrequest*. The choice of the flows is subject to whether the purchase item stock can meet the PO requirement. We use a constraint transition relationship to link the *inventorycheck* state to the *supplierpo* state. Its constraint condition is specified in the relationship dialog box by comparing the *quantity* parameter of *porequest* request with the *inventory* parameter of *inventorycheck* response. If the former is greater than the latter, the state transition will happen. Similarly, we specify another constraint transition from the *inventorycheck* state to the *paymentrequest* state, and the constraint condition is the item stock less than or equal to the PO quantity.

C – Transition iteration: A loop relationship is used to specify that all the operations between the *from* state and the *to* state of the loop relationship will be repeatedly executed. We use a loop relationship to specify the approval process of a supplier PO, which includes an *approvalnotification* and a *supplierpoapproval* operations. The approval process starts from the immediate manager of the purchaser until the manager with authority for the PO amount.

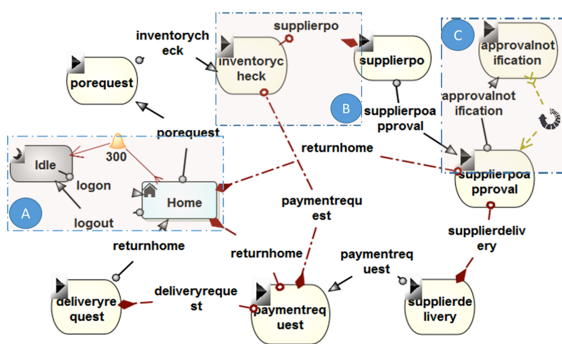

Figure 6: Example protocol model.

4.3 Behaviour Modeling

We use one operation *paymentrequest* as example to explain how endpoint behaviour is modeled by our behaviour DSVL. We model an operation behaviour by using a service node construct and instantiate it by providing the operation name. The operation request and response parameters are imported from the corresponding signature operation model automatically. The *paymentrequest* operation node consists of two sub nodes: *poinformationretrieve* to retrieve PO, product and client information, and *poamountcalculation* to calculate the total PO amount. These two nodes are placed between a pair of input and output bars.

The *poinformationretrieve* node is used to show how behaviour DSVL visual constructs are used to implement business logics. Figure 7 illustrates the operations and dataflows within the *poinformationretrieve* node. The node has one input parameter *pono*, and four output parameters: *quantity*, *unitprice*, *discount* and *errormessage*. The node includes three data query operations: (1) to retrieve PO *category*, *item*, *quantity* and *clientname* from *PurchaseOrderTable* by the *pono*; (2) to retrieve *unitprice* from *ProductTable* by the *category* and *item*; and (3) to retrieve *discount* from *ClientTable* by the *clientname*. If searching records are found, searching results will be placed on the normal output port (black circle) of data store operator. Otherwise, a *FatalError* variable will be assigned by following the exceptional output port (yellow circle).

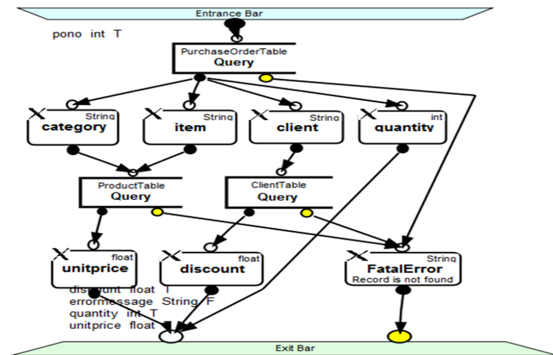

Figure 7: Example behaviour model.

4.4 Testing Environment Creation

We build our testing runtime environment by converting the above models into executable codes to be run inside Axis2. We use Eclipse as our Java IDE; and two Java projects *purchaseserver* and *purchaseclient* are created for hosting server and client side codes. The testing environment creation process is described as followings:

Testing environment platform creation: The signature model is transformed to a WSDL file, and the file is copied to both the client and server project folds. Then, the WSDL files in both project folds are converted to Axis2 Web Service platform by using Axis2 wsdl2java utility.

Protocol and behaviour models transformation and integration: The endpoint protocol and behaviour models are transformed to Java classes; and the Java classes are copied to the *purchaseserver* folder. Then, these Java classes are integrated into an Axis2 skeleton class. Figure 8 shows a code snippet of the skeleton class for *paymentrequest* operation. Signature parameters are verified against defined value ranges first at Line 8. Then, the protocol validity of the *paymentrequest* operation is tested at Line 9. If both the signature and protocol are verified correctly, the operation response message will be generated from Line 14 to 17.

```
<1>   public com.endpoint.purchase.Paymentrequest_response paymentrequest(
      com.endpoint.purchase.Paymentrequest_request paymentrequest_request)
<2>  {
<3>   //TODO : fill this with the necessary business logic
<4>    protocolError = protocol.OperationValidation("paymentrequest");
<5>
<6>    Paymentrequest_response resp = new Paymentrequest_response();
<7>
<8>    if (signature.paymentrequest(paymentrequest_request.getPono()) == 0)
<9>        if (protocolError.ErrorCode == 10)   {
<10>          resp.setErrormessage(protocolError.ErrorMessage);
<11>          resp.setErrorcode(protocolError.ErrorCode);
<12>        }
<13>        else {
<14>          paymentrequestResult = myPaymentrequest.
                    paymentrequest(paymentrequest_request.getPono());
<15>          resp.setErrorcode(paymentrequestResult.errorcode);
<16>          resp.setErrormessage(paymentrequestResult.errormessage);
<17>          resp.setAmount(paymentrequestResult.amount);
<18>        }
<19>    else {
<20>        resp.setErrorcode(1);
<21>        resp.setErrormessage("signature parameter value range error.");
<22>    }
<23>    return resp;
<24>  }
```

Figure 8: A code snippet of Axis2 skeleton class.

•

```
<1>  package com.endpoint.purchase;
<2>  import com.endpoint.purchase.PurchaseServiceStub;
<3>
<4>  public class PaymentrequestClient{
<5>    private String[] retMessage = new String[10];
<6>    public String[] Paymentrequest(int pono){
<7>      try{
<8>        PurchaseServiceStub stub =new PurchaseServiceStub
             ("http://localhost:8080/axis2/services/PurchaseService");
<9>        PurchaseServiceStub.Paymentrequest_request paymentrequest_req =
                 new PurchaseServiceStub.Paymentrequest_request();
<10>       paymentrequest_req.setPono(pono);
<11>
<12>       PurchaseServiceStub.Paymentrequest_response
                     paymentrequest_res = null;
<13>       paymentrequest_res = stub.paymentrequest(paymentrequest_req);
<14>
<15>       retMessage[0] = Integer.toString(paymentrequest_res.getErrorcode());
<16>       retMessage[1] = paymentrequest_res.getErrormessage();
<17>       retMessage[2] = Float.toString(paymentrequest_res.getAmount());
<18>     } catch(Exception e){
<19>       e.printStackTrace();
<20>       System.err.println("\n\n\n");
<21>       }
<22>     return retMessage;
<23>    }
<24>  }
```

Figure 9: An Axis2 stub class API.

Axis2 Web Service generation and deployment: We developed an Apache Ant build XML file to build the endpoint Axis2 Web Service automatically, and the built service *purchaseservice.aar* file is loaded to Tomcat webapps folder for providing the endpoint testing service.

SUT integration: A Java API file is provided for integrating a SUT with Axis2 stub file in the *purchaseclient* folder. Figure 9 shows the *paymentrequest* API file. A SUT accesses the endpoint testing service through a Tomcat application server URI at Line 8. The *paymentrequest*_request is sent by providing input parameter *pono* at Line 10. The *Paymentrequest* _response method is defined and assigned the response parameters from the testing endpoint at Line 12 and 13. Finally, these response parameters are assigned to the API return string array "retMessage" from Line 15 to 17.

Table 3: Approaches comparison.
(**MC**: Manual Coding, **IT**: Interactive tracing,
OA: Our Approach).

Attribute Description	MC	IT	OA
Functionality			
The approach detects all interface defects.	M	H	H
The approach reports signature defects.	H	N	H
The approach reports static protocol defects.	H	N	H
The approach reports dynamic protocol defects.	L	N	H
Business scenario can be simulated (time event, synchronous and unsafe operations).	L	N	H
Overall ranking of functionality	**M**	**L**	**H**
Productivity			
Endpoint is generated automatically.	N	H	N
The approach supports high-level abstraction.	L	N	H
The approach supports components reuse.	M	N	H
The approach has built-in error prevention mechanisms.	H	N	M
Network interface is generated automatically.	L	N	H
Overall ranking of productivity	**L**	**H**	**M**
Ease of Use			
Special training is needed.	N	L	M
Endpoint application knowledge is needed.	L	N	L
Programming skills are needed.	L	N	N
Visual notations are used.	N	N	H
Overall ranking of ease of use	**L**	**H**	**M**

5 EVALUATION

To assess how well the three research questions are addressed by our new endpoint modeling approach, we define three corresponding evaluation criteria: (1) testing endpoint functionality (addressing RQ1) – the approach should be able to develop various types of testing endpoints, which could be used to detect all sorts of interface defects; (2) development productivity (addressing RQ2) – the approach should ideally have high endpoint development productivity

with less development effort and time; and (3) ease of use (addressing RQ3) – the approach should be easy to learn and adapt.

These criteria were assessed by a technical comparison of the two currently available testing endpoint emulation approaches: specification-based manual coding and interactive tracing approaches. This comparison motivated our new specification-based model-driven approach. Thus, we provide details of how our approach compares with these two other approaches as well. After our approach was ready to use, we conducted a user study to evaluate to what extent our approach is accepted by IT professionals in respect to each of these criteria. It is also good to mention that we have made some improvements on our early versions of TeeVML based on the feedbacks from the user study.

5.1 Technical Comparison

We conducted the technical comparison by looking into what key techniques these approaches adopt to address the issues related to the evaluation criteria (refer to Table 2). We have added some attributes to these evaluation criteria. We then gave a four-point ranking subject to the level of support (N – n/a, L - low, M - medium or H - high) the approaches provide for each attribute and each evaluation criterion (refer to Table 3). The overall ranking of each evaluation criterion summarizes individual attribute's ranking and takes their importance into consideration.

The interactive tracing approaches have the H ranking for both the productivity and ease of use, as they create testing endpoints from interactive tracing data automatically. However, these approaches have two key shortcomings in terms of testing functionality. One is their usability, which is subject to the availability of interactive tracing data; another one is that they cannot report defect type and cause information. So, we give these approaches the L ranking for the testing functionality.

Table 2: Approaches' techniques.

	Manual Coding	Interactive tracing	Our Approach
Functionality	The key motivation of these approaches is to provide SUT performance testing by emulating large number of endpoints of the same type. To achieve this objective, these approaches adopt a light-weight architecture design (Hine et al., 2009) and some testing features are deliberately neglected. Dynamic protocol behaviour cannot be modeled, as state transition is triggered only by an operation. Unless great effort is made, behaviour layer modeling will be limited.	To provide integration testing, these approaches search for the right request matching on data byte level without any knowledge about upper-level message syntax. They can only tell whether a test is passed or failed, but cannot provide any defect information. These approaches are not usable for testing a new application, as its trace data are not available.	Our testing endpoint provides integration testing from signature, protocol and behaviour abstraction layers. The signature layer model supports all RPC style communications; the protocol layer can model both static and dynamic protocol behaviours; and the behaviour layer uses a hierarchical structure dataflow programming for modeling complicated logic implementation.
Productivity	The approaches adopt a modular architecture design, where an endpoint type dependent message engine module is separated from an endpoint type independent network infrastructure and a system configuration module (Hine et al., 2009). However, as the message engine is coded manually, significant amount of development effort is needed for each new endpoint type.	Testing endpoint is created by recording the interactive tracings between an endpoint and an earlier version of the SUT application. These approaches do not need any endpoint development work, but some effort on trace data recording.	An endpoint is modeled by layers, and layer models are transformed to executable source codes. The key solution to productivity improvement is to maximize components reusability. We have adopted multi-level design for the signature DSVL and node hierarchical structure for the behaviour DSVL.
Ease of Use	To develop an endpoint, developers must have both business domain knowledge and programming skills.	Neither business domain knowledge nor programming skills are required.	Developers must have business domain knowledge, and some modeling skill is preferred. To achieve ease of use, we applied the principles of Physics of Notations (Moody, 2009) to optimize our visual nation designs.

In contrast, both specification-based types of approaches need to develop endpoint by using different techniques. As our approach uses higher level of abstraction models than code to express design intent, we have given our approach the M ranking and manual coding the L ranking for both the productivity and ease of use. Both types of approaches can report static defects, but ours can report dynamic defects as well. So, our approach ranks H for the testing functionality, while manual coding is given the M ranking.

5.2 User Study

Our user study was conducted in two phases to measure the perceived usefulness and perceived ease of use (Davis, 1989) of our endpoint modeling tool, respectively. Phase One study was conducted through interviews, including an introduction to the toolset and a Q&A session. We invited 16 testing experts, who had two or more years industry testing experience and were knowledgeable about integration testing, to take part in the survey. For Phase Two, we asked participants to model the deposit operation of a banking system endpoint. We then collected their opinions based on their experience with our TeeVML toolset. Total of 19 software developers and IT research students took part in the survey. Most of them had five or more years software development experience and were familiar with at least one third generation language.

The questionnaires included 5-point Likert Scale, single-choice and multi-choice questions. We counted the frequency of participant responses to measure the degree of acceptance to a question statement. We had total 58 questions and selected some of them for this paper results presentation. The full result reports are available at https://sites.google.com/site/teevmlapsec/..

5.2.1 Phase One Results

The main objective of Phase One survey is to evaluate the first criterion -- testing endpoint functionality. We selected four questions from Phase One questionnaire (refer to Table 4) to analyse this criterion from two different angles. The first one is about participants' acceptance of emulated testing endpoint in general and by each interface layer. The second one is to find out possible reasons why the participants would consider using (or not using) our emulated testing endpoints in their future projects.

Q8 on usefulness of emulated testing environments received 14 out of 16 in favour response rate (scoring 4 or 5). This is a good indication of participants' acceptance of the overall usefulness of testing endpoints. From Q9 we see that protocol layer received in favour responses from all participants. The reason could be that most applications do not have a well-documented protocol specification. SUT protocol related defects can only be found by conducting integration testing.

Table 4: Phase One user study results.

No	Statement	Frequency				
		5	4	3	2	1
Q8	In your opinion, an emulated testing environment is useful for an application inter-connectivity and inter-operability test.	**8**	**6**	**0**	**1**	**1**
Q9	What kinds of testing features do you want to see an emulated testing environment provides to system under test for interconnectivity and inter-operability test?					
	Correctness of message signature					**13**
	Correctness of interactive protocol					**16**
	Correctness of interactive behaviour					**14**
	Correctness of non-functional requirement					**11**
	Other					**1**
Q13	What are the main motivations for you to use emulated testing environment?					
	Cost saving on application hardware and software investment					**14**
	Effort saving on application installation and maintenance					**10**
	Lack of application knowledge					**5**
	Early detection of interface defects					**15**
Q14	What are your main concerns, which could prevent you from using emulated testing environment?					
	Extra development effort on testing endpoints					**6**
	Learning a new technology					**6**
	Inadequate testing functionality					**7**
	Emulation accuracy					**7**
	Result reliability					**12**

Responses to Q13 indicate that the top reason for using endpoints is early detection of interface defects. Integration testing is normally conducted during the later stages of software development lifecycle. This is partly because integration testing environment is not available before then. If a rapid and cheap solution for testing environment deployment was available, developers may have preferred to conduct at least part of integration testing earlier. Responses to Q14 indicates that most participants' concerns are on the reliability of testing endpoint results. We believe the main reason behind is that software developers are used to using real applications for their integration testing. However, an endpoint is actually a simplified version of its real

application. This might have some impacts on SUT testing results.

Table 5: Phase Two user study results

No	Statement	Frequency				
		5	4	3	2	1
Q9	How long did it take you to complete the task?					
	10 – 15 minutes					**1**
	16 – 20 minutes					**4**
	21 – 25 minutes					**7**
	26 – 30 minutes					**3**
	30+ minutes					**4**
Q10	How many times have you asked for support?					
	None					**4**
	One time					**4**
	Two times					**4**
	Three times					**5**
	Four times or more					**2**
Q12	You would like to use the tool in your future project.	**7**	**11**	**1**	**0**	**0**
Q13	You found the tool unnecessarily complex.	**0**	**1**	**2**	**12**	**4**
Q14	You found the tool was easy to use.	**8**	**10**	**1**	**0**	**0**
Q15	You would need support to be able to use the tool.	**0**	**2**	**9**	**8**	**0**
Q16	You found the various features of the tool were well integrated.	**8**	**10**	**0**	**1**	**0**
Q17	You found there was too much inconsistency in the tool.	**0**	**0**	**0**	**11**	**8**
Q18	You would image that most people would learn to use the tool very quickly.	**5**	**12**	**1**	**1**	**0**
Q19	You found the tool very cumbersome to use.	**0**	**0**	**2**	**10**	**7**
Q20	You felt very confident using the tool.	**4**	**13**	**2**	**0**	**0**
Q21	You needed to learn a lot of things before you could get going with the tool.	**0**	**1**	**3**	**8**	**7**
Q22	In your opinion, comparing to a third generation language (e.g. Java) you are familiar with, how much would a typical endpoint development effort be reduced by using the tool?					
	Almost the same					**0**
	10 – 25%					**2**
	26 – 50%					**6**
	51 – 80%					**9**
	81%+					**2**

5.2.2 Phase Two Result

In Phase Two, we evaluated the ease of use and development productivity criteria. The former used the ten questions from Software Usability Scale (SUS) (Brooke, 1996) (refer to Q12 to Q21 of Table 5), and the latter was captured by the three questions specifically related to performing the assigned task (refer to Q9, Q10, Q22 of Table 5).

The responses to the SUS questions were quite positive with average 85% in favour. Particularly, all participants did not agree (scoring 1 or 2) with Q17 negative statement on tool's inconsistency. Q15 received 8 out of 19 in favour responses and 9 participants voted neutral. This result is confirmed by Q10, where only 4 participants did not ask for support for finishing their tasks. We believe this is due to the fact that our introduction video was targeted toward introduction of the tool and the approach in general rather than stepwise instruction of using the tool for similar examples. As a result, more participants felt they needed to ask for instructor's support. We believe this will be rectified overtime with more usage of the approach and tool support.

Q22 captures participants' opinions on how much of their time and effort will be reduced through using our approach. 12 out of 19 respondents chose "50% - 80%" and "80%+" and no participant voted "almost the same". As a result, we can conclude that most participants believed that our approach could increase endpoint development productivity. Confirming this is the fact that 15 out of 19 participants finished their task in less than 30 minutes (see responses to Q9). Based on this result, we can generalize that it is possible to model a relatively complex endpoint with more than ten operations within a day through using our tool support for testing environment emulation.

6 RELATED WORK

Testing distributed systems is a complex problem. Ghosh and Mathur raised nine issues to be addressed in testing distributed systems (Ghosh and Mathur, 1999). Method stubs, mock objects, and existing emulation approaches have been used to emulate the behaviour of endpoint systems (Gibbons, 1987, Freeman et al., 2004, Hine et al., 2009, Du et al., 2013, Yu et al., 2012, Giudice, 2014). However, some of these approaches introduce a large implementation overhead; while the others depend on the availability and accessibility of existing SUT applications.

Pact is an open-source tool, enabling consumer contract driven testing (Pact, 2016). It provides a DSL for users to specify operation requests and expected responses from service provider. However, each interaction in Pact is verified in isolation without context maintained from previous

interactions. Therefore, Pack is not suitable for performing protocol testing.

Model-driven engineering is an avenue to raise the level of abstraction beyond programming by specifying solution directly using problem domain concepts (Poole, 2001). UML is a general purpose modeling language, its visual presentation makes it understandable by a wide range of software stakeholders. UML Testing Profile (UTP) provides a generic extension mechanism for the automation of test generation processes (Schieferdecker et al., 2003); state chart simulates finite-state automaton (Zhang and Liu, 2010); and activity diagram graphically represents workflows of stepwise activities and actions (Dumas and Terhofstede, 2001). However, there are two main problems with using UML to define new modeling languages (Abouzahra et al., 2005): one is usually hard to remove parts of UML that are not relevant in a specialized language; another one is that all diagram types have restrictions based on UML semantics.

DSLs are a model-driven development approach, where the first class entities are the models and the model transformations (Selic, 2003). DSLs often support higher-level abstraction than general purpose modeling languages (e.g. UML), so they require less programming effort and low-level details to specify a given system. DSLs have been widely used in various of business and technical domains, such as, WebDSL for web applications (Visser, 2008), MaramaEML for business process modeling (Li et al., 2014) and SDL (Kim et al., 2015) for supporting statistical survey process. In contrast, we use a suite of DSVLs tailored to modeling signature, protocol and behaviour aspects of endpoints.

7 SUMMARY

Modern software development needs quick yet cost effective solutions to develop and deploy integration testing environment. Due to their unique advantage on endpoint development productivity over others, interactive tracing based service virtualization approaches are gaining momentum in recent years. But they still need a specification-based tool to specify testing endpoint, as application trace data may be neither available nor usable.

Our model-driven approach divides a software interface into three abstraction functional layers, and a suite of domain-specific visual languages have been developed for modeling these layers. By this layered modeling, we have achieved high development productivity and rich testing functionality. In addition, our approach supports partial endpoint development, where a testing endpoint may have only one or two of these layers to meet SUT testing requirement.

Existing specification-based testing environment emulation approaches cannot validate SUT's runtime protocol behavior, as they check the validity of an incoming service request based on endpoint state only. Our tool protocol model is based on an EFSM and we use behavior models to capture dynamic protocol aspects. Furthermore, our testing environment has a rich set of functions for simulating typical business scenarios, such as time-driven state transition, synchronous and unsafe services.

In a realistic enterprise environment, application security requirements may put extra constraints on the validity of operation requests. Some of the constraints are role-based, so that some operations are accessible only to a certain group of users. Other constraints are security policy related, such as restriction on available time or specific pattern required for some operation parameters. Also, there are some performance limitations such as response time, and robustness requirements such as handling endpoint malfunctions. These and other QoS requirements are part of our future works.

ACKNOWLEDGMENT

The authors gratefully acknowledge support for this research by an Australian Post-graduate Award and an Australian Research Council Discovery Projects grant DP140102185.

REFERENCES

ABOUZAHRA, A., BÉZIVIN, J., DEL FABRO, M. D. & JOUAULT, F. A practical approach to bridging domain specific languages with UML profiles. Proceedings of the Best Practices for Model Driven Software Development at OOPSLA, 2005. Citeseer.

BROOKE, J. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry,* 189**,** 4-7.

DAVIS, F. D. 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly***,** 319-340.

DU, M., SCHNEIDER, J.-G., HINE, C., GRUNDY, J. & VERSTEEG, S. 2013. Generating service models by trace subsequence substitution. *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures.* Canada: ACM.

DUMAS, M. & TERHOFSTEDE, A. 2001. UML activity diagrams as a workflow specification language. *≪UML≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools.* Springer.

FREEMAN, S., MACKINNON, T., PRYCE, N. & WALNES, J. 2004. Mock roles, objects. *In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications.* Canada: ACM.

GHOSH, S. & MATHUR, A. P. Issues in testing distributed component-based systems. First ICSE workshop on testing distributed component-based systems, 1999. Citeseer.

GIBBONS, P. B. 1987. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. *IEEE Transactions on Software Engineering,* 13**,** 77-87.

GIUDICE, D. L. 2014. The Forrester Wave™: Service Virtualization And Testing Solutions. *In:* FORRESTER (ed.).

HAN, J. 2000. Rich Interface Specification for Software Components. *Peninsula School of Computing and Information Technology Monash University, McMahons Road Frankston, Australia*.

HINE, C., SCHNEIDER, J.-G., HAN, J. & VERSTEEG, S. Scalable emulation of enterprise systems. Software Engineering Conference, Australian, 2009. IEEE, 142-151.

JAYASINGHE, D. 2008. *Quickstart apache axis2*, Packt Publishing Ltd.

KIM, C. H., GRUNDY, J. & HOSKING, J. 2015. A suite of visual languages for model-driven development of statistical surveys and services. *Journal of Visual Languages & Computing,* 26**,** 99-125.

LI, L., GRUNDY, J. & HOSKING, J. 2014. A visual language and environment for enterprise system modelling and automation. *Journal of Visual Languages & Computing,* 25**,** 253-277.

LIU, J., GRUNDY, J., AVAZPOUR, I. & ABDELRAZEK, M. 2016. A Domain-Specific Visual Modeling Language for Testing Environment Emulation. *IEEE Symposium on Visual Languages and Human-Centric Computing.* Cambridge, UK.

MOODY, D. L. 2009. The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. *Software Engineering, IEEE Transactions on,* 35**,** 756-779.

PACT. 2016. *Enables consumer driven contract testing* [Online]. Available: https://github.com/realestate-com-au/pact [Accessed].

POOLE, J. D. Model-driven architecture: Vision, standards and emerging technologies. Workshop on Metamodeling and Adaptive Object Models, ECOOP, 2001.

SCHIEFERDECKER, I., DAI, Z. R., GRABOWSKI, J. & RENNOCH, A. 2003. The UML 2.0 testing profile and its relation to TTCN-3. *Testing of Communicating Systems.* Springer.

SELIC, B. 2003. The pragmatics of model-driven development. *IEEE software,* 20**,** 19.

SOUSA, T. B. Dataflow Programming Concept, Languages and Applications. Doctoral Symposium on Informatics Engineering, 2012.

VISSER, E. 2008. WebDSL: A Case Study in Domain-Specific Language Engineering. *Generative and Transformational Techniques in Software Engineering II.* Springer Berlin Heidelberg.

YU, J., HAN, J., SCHNEIDER, J.-G., HINE, C. & VERSTEEG, S. 2012. A virtual deployment testing environment for enterprise software systems. *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures.* Italy: ACM.

ZHANG, S. J. & LIU, Y. An Automatic Approach to Model Checking UML State Machines. Fourth International Conference on on Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 9-11 June 2010 2010. 1-6.