TeeVML: Tool Support for Semi-automatic Integration Testing Environment Emulation

Jian Liu
School of Software and Electrical Engineering
Swinburne University of Technology
Hawthorn, VIC 3122, Australia
jianliu@swin.edu.au

John Grundy
School of Information Technology
Deakin University, Burwood, VIC 3125, Australia
j.grundy@deakin.edu.au

Iman Avazpour
School of Information Technology
Deakin University, Burwood, VIC 3125, Australia
iman.avazpour@deakin.edu.au

Mohamed Abdelrazek
School of Information Technology,
Deakin University, Burwood, VIC 3125, Australia
mohamed.abdelrazek@deakin.edu.au

ABSTRACT

Software environment emulation provides a means for simulating an operational environment of a system. This process involves approximation of systems' external behaviors and their communications with a system to be tested in the environment. Development of such an environment is a tedious task and involves complex low level coding. Model driven engineering is an avenue to raise the level of abstraction beyond programming by specifying solution directly using problem domain concepts. In this paper we propose a novel domain-specific modeling tool to generate complex testing environments. Our tool employs a suite of domain-specific visual modeling languages for modeling emulation environment at a high level of abstraction. These high level specifications are then automatically transformed to runtime environment for application integration testing, boosting development productivity and ease of use. The tool demonstration video can be accessed here: https://youtu.be/H3Vg20Juq80.

CCS Concepts

•Software and its engineering → Model-driven software engineering •Software and its engineering → Interoperability •Software and its engineering → Software testing and debugging •Software and its engineering → Empirical software validation •Software and its engineering → Integrated and visual development environments.

Keywords

Model-driven engineering; domain-specific visual modeling language; software component interface description; testing environment emulation.

1. INTRODUCTION

Modern enterprise software systems usually operate in a distributed and heterogeneous environment. These systems interact and cooperate with other systems in their environment for providing composite services to support daily enterprise operations. Thus, the performance of a software system is governed not only by its internal implementation but also driven by interactions with other systems. With the increasing complexity of the environments enterprise systems are deployed in, it is becoming more difficult and expensive to replicate a realistic production environment for software systems integration testing. On the other hand, a production environment is generally unsuitable to conduct this kind of testing, as a fault in the enterprise system may cause disruption or irreversible damage to that production environment.

The UML Testing Profile (UTP) is often used to systemically define tests for static and dynamic aspects of systems modeled in UML [1]. UTP provides a generic extension mechanism for the automation of test generation processes. The Model Language (TML) is another testing language for describing Markov chain usage models to characterize the probabilities of all usages using some statistic techniques and generate test cases accordingly [2]. However, both the UTP and TML are for server-side system testing and do not have abstractions suitable for developing a testing environment for client-side application integration testing.

Testing environment emulation is an emerging technique to provide integration testing environment for a System Under Test (SUT) that interacts with many external systems. The main idea is to model the run-time behaviors of each system (also known as endpoint) in the environment and replace each real system by an instantiation of the corresponding model in the emulation environment. The aim is to make the emulated testing environment rich enough to "fool" the SUT that it is talking to the real systems. Other behaviors and the systems which sit underneath and in the background are ignored from the emulated environment perspective where possible.

There have been two approaches to develop such integration testing environment. The first one is specification-based approach [3], where IT professionals manually develop interaction models with the use of available knowledge about the underlying interaction protocol and system behaviors, respectively. The second one is interaction trace data record-and-replay approach [4]. This approach uses a traffic recording tool to sit between a SUT and an endpoint, recording information about how the SUT and endpoint interact. Later those recordings can be used to simulate the endpoint response for each corresponding request by searching for the close-matching request in traffic recordings. Both of these approaches have their shortcomings: former approach has high development and set-up costs and requires access to detailed system knowledge and implementations. The

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970291

later one depends on the availability of traffic recordings for all interactive scenarios between a SUT and its operational environment.

Aiming to achieve high development productivity and ease of use for domain experts, we have developed a novel specification-based domain-specific Visual Modeling Language for Testing environment emulation (TeeVML). Our TeeVML is based on a layered software components interaction description framework, where each layer represents a modeling problem domain. We provide a separate Domain-Specific Visual Language (DSVL) for each of these interaction layers. Domain experts use these DSVLs to model their endpoint by layers. TeeVML's testing runtime environment is provided by transforming endpoint signature model WSDL XML file to Axis2 Web Service platform [5].

2. MOTIVATION

To motivate our TeeVML tool, we use a business case as an example and describe the interactive behaviors between a SUT and an endpoint. The SUT for this case study is an Internet banking application. It communicates with a core banking system (as the endpoint) for accessing user and bank account records for each end user service request. From description simplicity consideration, we assume that the Internet banking application provides six services to its end users: *logon*, *logout*, *searchaccount*, *deposit*, *withdraw* and *moneytransfer*.

From the SUT's perspective, the endpoint must provide integration testing functionalities, which should mimic its real system services. Therefore, we can realistically assume that the main testing endpoint characteristics are: (1) an endpoint only considers the external behaviors (or call services) of the real system, and all internal implementations will be ignored; (2) an endpoint only provides a subset of the real system invoked by the SUT; and (3) an endpoint should be able to detect all SUT interface defects, identifying their types and origins. Based on the above assumptions, we can describe the core banking system endpoint from three different abstraction layers: (1) service signature – service request name and parameters, and response return values; (2) protocol – valid temporal sequence of services; and (3) interactive behavior -- service request process and response generation. We describe the endpoint from these three layers in Table 1.

It is not feasible to test the Internet banking application with the production core banking system. It would also be very expensive to duplicate the core banking system. Hence, conventional interaction trace data record-and-reply and specification-based approaches would not be feasible or would be difficult to use, as the former relies on the existence of interaction trace data and the later requires development of detailed endpoint model implementations. We thus define three key objectives for our testing environment emulation tool:

- **Testing endpoint functionalities** – the tool should be able to develop various types of testing endpoints, which could be used to detect all sorts of interconnectivity and interoperability defects of SUTs;
- **Development productivity** – the tool should ideally have high endpoint development productivity, and less development effort and time;
- **Ease of use** – the tool should be easy to learn and use to specify endpoint interface and behavior at high levels of abstraction rather than implementation details.

3. OUR APPROACH

To identify the common entities and find out their relationships, we conducted our testing environment emulation domain analysis by investigating three typical business applications interacting with their clients. 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 components interaction description framework for testing environment emulation, and identified service request defect types to be detected.

Table 1. Core banking system endpoint description

Signature
All services have a name and consist of one or more parameters for their request and/or response.
All services have a request and response, except for *logout* service, which has a request only.
Request and response parameters can be a string, integer, float, boolean or date data type.
Request and response parameters can be either mandatory or optional.
A *logon* service request has optional username and password fields for authenticating a secured interactive session.
The userid field in *logon* request is five digit integer; amount field in *deposit* and *withdraw* requests ranges from 0.00 to 99999.00; amount field in *moneytransfer* request ranges from 1000.00 to 99999.00
Protocol
A *logon* request transits the endpoint from idle state to home state and an interactive session starts. On the opposite direction, a *logout* request terminates the session.
In a secured session, all the services can be accessed by the SUT. Otherwise, only *logout* and *searchaccount* services can be invoked.
As the minimum money transfer amount is $1000.00, a *moneytransfer* request must follow a *searchaccount* request; and the amount value in the *searchaccount* response will determine whether the *moneytransfer* request can be executed.
Timeout event will automatically change the endpoint state from a "from" state to a "to" state after a certain period of time.
All service requests will be rejected, when endpoint is processing a synchronous service.
All transaction services (*deposit*, *withdraw* and *moneytransfer*) are considered as unsafe services, and multiple requests for a same service are not allowed.
Behavior
To start a secured session, *logon* request must be authenticated by userid, username and password parameters; if only userid parameter is provided, the interactive session will be insecure.
All query and transaction services use userid field to find a bank account record, and retrieve the account balance. If the account record cannot be found, an error code and error message will be generated in response.
For *withdraw* and *moneytransfer* services, the transaction amount must be equal or less than the account balance. Otherwise, a not enough balance error occurs.

3.1 Software Components Interaction Description Framework

Our software components interaction description framework abstracts an interaction into three horizontal and two vertical layers. The horizontal layers include signature, protocol and behavior. The vertical layers include data store (data persistence access) and Quality-of-Service (QoS) (as non-functional requirements). A SUT service request is processed horizontally by an endpoint step by step from signature, protocol, down to interactive behavior 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 validating service request syntax and sequence correctness, before handing it over to the behavior layer for generating response. Vertical layers are not directly involved in request processing, but provide support to horizontal layers. We use modular development approach to model an endpoint – i.e. each module represents a particular interactive layer.

3.2 Integration Testing Environment

A testing endpoint is a server-side application, receiving and processing service requests from a SUT based on Remote Procedure Call (RPC) communication style. Thus, the endpoint should be able to validate the correctness of service requests sent from the SUT. In general, there are two types of service request defects: functional defects, which are directly related to service request processing by endpoint; and non-functional requirement defects, such as non-compliance with security requirement or robustness under different operational conditions. Table 2 lists all the functional defects a SUT service request may cause. Our current version of testing endpoint does not support QoS testing and it will be our future work.

Table 2. Service request defects

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

From Design by Contract (DbC) programming style's perspective, a SUT's obligation is to send correct service requests to an endpoint [6]. The way these requests to be processed is defined in the endpoint's internal implementation. While it may seem as if endpoint behavior modeling is not necessary for emulating an integration testing environment, there are situations where a business process may have several interactions between a SUT and an endpoint. The SUT may send a different subsequent request to its endpoint, depending on what values are returned in the response message it has received from a previous service request (refer to P3 defect type of Table 2). As a result, an emulated endpoint needs to have behavior modeling functionality for capturing some runtime SUT protocol defects.

Testing endpoint functionalities that is one of our tool's objectives, is measured by the coverage of service request defects that can be detected by testing endpoint. In the followings, we discuss how the integration testing defects listed in Table 2 can be detected from our TeeVML's design.

3.3 TeeVML Design

Our TeeVML consists of signature, protocol and behavior DSVLs. Each of the DSVLs includes a collection of visual notations for modeling an endpoint layer and code generators for transforming the layer model to target forms. The design of visual notations is based on a metamodel or a programming paradigm, which covers all endpoint layer modeling aspects and their inter-relationships. We used MetaEdit+ 5.1 [7] as the meta-language to develop the DSVLs. In the following subsections we briefly describe these DSVL designs. More details of TeeVML tool and its visual notations are subject of another publication [8] [1].

3.3.1 Signature DSVL

To improve components reusability and have a concise presentation, we have adopted a hierarchical DSVL architecture design approach (refer to Figure 1). The top-level signature DSVL uses WSDL 1.1 [9] as its metamodel to define the five WSDL entity types and their relationships (refer to Figure 1a). The middle-level operation DSVL is for defining request and/or response message(s) contained in an operation (or call service) (refer to Figure 1b). The bottom-level message DSVL uses W3C XML Schema 1.1 [10] as its metamodel to define complex elements in a message (refer to Figure 1c).

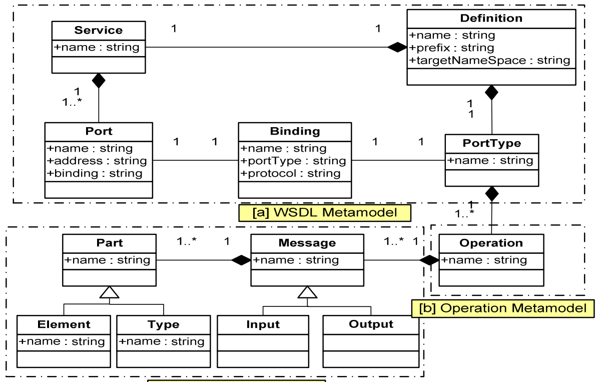


Figure 1. Signature DSVL metamodel

The signature defects S1 to S3 in Table 2 are detected by Axis2 Web Service engine transformed from signature WSDL file. For S4 defect debugging, two fields are added to element type for

[1] The endpoint source codes, a recorded demo video, user study result reports and an in press publication are available at: https://sites.google.com/site/teevmlase/.

specifying the minimum and maximum values of a request parameter.

3.3.2 *Protocol DSVL*

To capture dynamic endpoint protocol behaviors, we used an Extended Finite State Machine (EFSM) metamodel to describe endpoint protocol behaviors (refer to Figure 2). One entity type and two entity properties are added to an operation-driven state transition FSM (marked yellow in Figure 2). The entity type is the *InternalEvent*, which is used to define state transitions triggered by time event. One of the entity properties is the *StateTransitionConstraint* of the transition entity; and it is used for specifying either static or dynamic constraints on state transition function. Another one is the *StateTimeProperty* of the state entity, which is for simulating synchronous and unsafe operations.

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

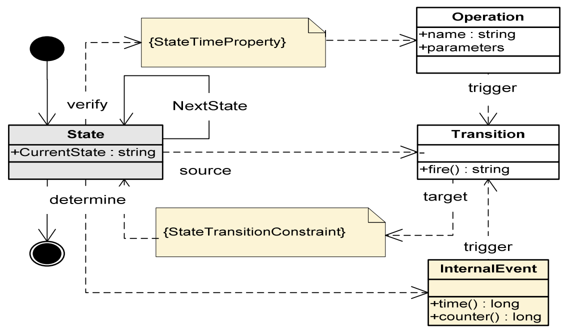


Figure 2. Protocol DSVL metamodel (EFSM)

3.3.3 *Behavior DSVL*

The endpoint behavior DSVL was designed based on dataflow programming paradigm [11]. We chose this metaphor as it allows complex specification of behavior models but is understandable by a wide range of target end users. The dataflow programming execution model is represented by a directed graph. The nodes of the graph are data processing units, and directed arcs between the nodes represent data dependencies. Data flows in each node from its input connector. 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 output connector for the next node(s) in the chain.

4. EXAMPLE USAGE[1]

Here, we use the core banking system from the Motivation section as an example to explain how a testing endpoint is developed. Our testing endpoint development process consists of three steps: (1) modeling endpoint – to model endpoint signature, protocol and behavior layers by using TeeVML, (2) transforming models – to transform endpoint models to WSDL XML file (signature model) and Java class files (protocol and behavior models) by code generators, and (3) integrating the generated codes with domain framework in a Java IDE environment.

4.1 Signature Modeling

Signature modeling starts from specifying endpoint level properties. Then, signature DSVL is used to instantiate the five WSDL entity types (service, port, binding, porttype and operation) by providing their names and relevant information. They are linked together by using either a composition or an association relationship. All the entity types have just one instance, except for the operation. The number of the operation instances depends on the services provided by the endpoint.

We use the operation *deposit* as an example to show how an operation can be modeled. The operation is instantiated by assigning the operation name as *deposit* and pattern as in-out. Then, operation DSVL is used to specify the *deposit_request* and *deposit_response* messages in the operation. The request message label is “in”, and response message label is “out”.

Message elements are defined by using message DSVL. The request message contains *userid* and *amount* elements, and they are placed by their IDs in alphabetic order. The *userid* data type is defined as integer and the element is mandatory. Since a valid *userid* is a five-digit integer, the element’s minimum field is specified as 10000 and maximum field as 99999. Similarly, the *amount* element properties are defined with data type as float, mandatory field, minimum 0 and maximum 99999. The response message consists of three elements: *newaccountbalance*, *errorcode* and *errormessage*. The *newaccountbalance* is a float data type, *errorcode* is integer and *errormessage* is string. The *newaccountbalance* and *errorcode* fields are mandatory with default value of 0.

Figure 3a illustrates the hierarchical signature model of the core banking system endpoint, including the top-level signature model, the middle-level *deposit* operation, and the bottom-level request and response messages.

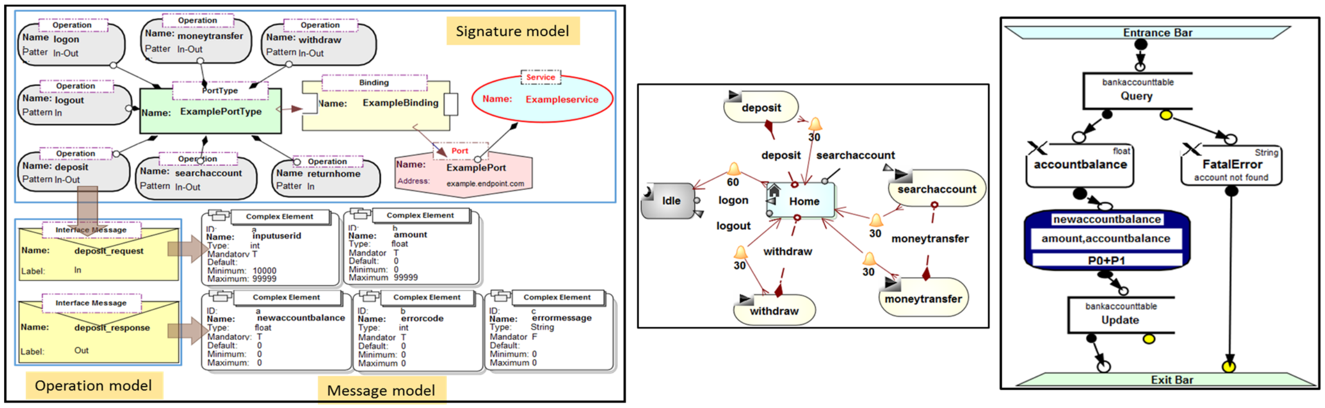
4.2 Protocol Modeling

Endpoint protocol is modeled using protocol DSVL. The first step of protocol modeling is to initiate a session by using a logon transition relationship linking idle state to home state. On the opposite direction, a logout transition relationship ends the session. The session can also be terminated by a timeout event, using a timeout relationship from home state to idle state.

When the endpoint is at its home state, it may accept one query service request – *searchaccount* and two transaction service requests – *deposit* and *withdraw*. The query transaction can be accessed in a secured or an insecured session. Therefore, a standard transition relationship is used to represent the state change from home to the *searchaccount* state. On the other hand, the transaction services are only valid in a secured session, authenticated by *username* and *password* parameters in *logon* service request. Therefore, a constraint transition relationship is needed to represent such a state transition. The constraint condition is defined by specifying the *inputusername* parameter of the *logon* service as not equal to null value. Similarly, as the minimum money transfer amount is $1000.00, the returned bank account balance from a *searchaccount* service determines whether or not a *moneytransfer* service is valid. Figure 3b illustrates the banking system endpoint protocol model.

4.3 Behavior Modeling

Endpoint behavior is modeled using behavior DSVL. We use one service - *deposit* as an example to show how endpoint behavior is modeled. The first step of behavior modeling is to define the service node by assigning its name. The request and response parameters of the service will be imported from the matching signature model automatically.



[a] Signature Model [b] Protocol Model [c] Behavior Model

Figure 3. Example endpoint three interactive layers modeling

The service node behavior implementation is specified by using a node (or call method) sub-graph. The first two constructs to use are a pair of entrance and exit bars. They define inputs and generated outputs to and from the method, and specify where the method execution starts and ends. There are two out ports on the exit bar for normal execution outputs (hollow circle) and exceptions (yellow circle), respectively. The first operation is to retrieve account balance by searching bank account table using *inputuserid* parameter. If the searching record is found, the account balance will be assigned to a variable *accountbalance*. Otherwise, a *FatalError* string variable will be assigned and placed on the exception out port of the exit bar. The next operation is to calculate new account balance by adding input *amount* to the *accountbalance* variable. The calculation is specified by using an evaluator, with assigned *newaccountbalance* variable name on the top, parameters used in the middle, and formula at the bottom. The last operation is to update the same bank account record with the *newaccountbalance*. Figure 3c illustrates the example endpoint *deposit* service node operations and dataflows.

4.4 Testing Environment Creation

Our testing runtime environment is built by transforming the above endpoint layer models into a WSDL XML file and Java class files. We use Eclipse as our Java IDE to build two projects for hosting server and client side Java files separately. The details of testing environment creation process are described as follows:

1. **Testing environment platform creation** – The signature model is transformed to a WSDL file, then the file is transformed to Axis2 Web Service platform by using Axis2 *wsdl2java* utility.

2. **Protocol and behavior models transformation and integration** – The protocol and behavior models are transformed to Java classes, then these Java classes are integrated with Axis2 skeleton class.

3. **Axis2 Web Service generation and deployment** -- An Apache Ant build XML file is used to build endpoint Axis2 Web Service automatically, and the built service aar file is loaded to Tomcat application server.

4. **SUT integration** – A Java API file is provided for integrating a SUT with Axis2 stub file in the client project.

By now, the core banking system testing endpoint is ready to provide integration testing service to its SUT. Figure 4 illustrates the integration testing runtime environment. The SUT is on the top of right-hand side of Axis2 client, communicating with Axis stub class through a Java API. The lower grey areas at both sides are Axis2 SOAP engine for low-level SOAP message exchanges. The behavior and protocol classes are located on the top of left-hand side of Axis2 server, integrated with Axis2 skeleton class.

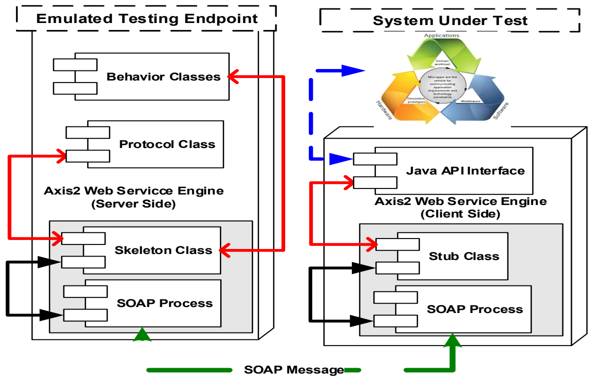


Figure 4. Testing runtime environment

5. EVALUATION AND DISCUSSION

To qualitatively evaluate our tool, we have defined three evaluation criteria, each corresponding to one of the three objectives defined in the Motivation section. Our evaluation process has two parts: In the first part, we have compared our tool versus two other testing environment emulation approaches from a technical point of view. In the second part, we have performed a user study of software testing experts and developers to obtain their opinions on our tool's usefulness and ease of use.

5.1 Technical Comparison

There are two main testing environment emulation approaches being used currently as described in the Introduction section: specification-based by manual coding and interaction trace data record-and-replay. In Table 3, we compare our tool with these existing approaches, and give a three-point ranking (low, medium or high) subject to the level of support they provide for each of the evaluation criteria. Overall our tool compares well with these existing approaches.

5.2 User Evaluation

The user study was conducted in two phases to measure the two variables of the perceived usefulness and perceived ease of use defined by Davis [12], respectively. In the first phase, we conducted interviews with testing experts to examine the usefulness of an emulated testing environment for SUT integration testing. In the second phase, we assessed the ease of use of our tool by asking software developers to perform a modeling task. All the survey participants were asked to fill an online questionnaire for collecting their opinions on each question statement. For this paper evaluation results presentation, we only summarise the overall responses to some of the questions, and the full result reports are available online[1].

Table 3. Emulation approaches comparison

Manual Coding	**Interaction Trace Data**	**Our Tool**
Testing Endpoint Functionalities		
Medium - signature and static protocol behaviors.	**Low** - cannot provide defect information.	**High** – complete signature and protocol behaviors.
Development productivity		
Low – manually coding endpoint.	**High** – interaction trace data recording.	**High** – modeling endpoint.
Ease of use		
Low - programming skill and domain knowledge.	**High** – no special skill requirement.	**Medium** – domain knowledge only.

Regarding the usefulness, we have received 87% in favour response rate as a whole. This is a good indication of the participants' acceptance of our emulated testing environment. In particular, all participants liked the protocol layer testing functionality. We believe the main reason is that many applications do not have a well-documented protocol specification, and protocol related defects can only be found by conducting integration testing. As to what motivates our participants to use testing endpoints, the top reason was early detection of interface errors, rather than savings on cost and effort. In current practice, 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, software testers may have preferred to conduct at least part of integration testing earlier

To evaluate the ease of use, we used the ten questions from Software Usability Scale (SUS) [13]. The SUS questions' responses were quite positive with average 85% in favour. To capture participants' ideas on how much of their time and effort will be reduced through using our toolset comparing with a third generation language, 57% respondents chose "50% - 80%" and "80%+". As a result, we can conclude that most participants believed that our tool could increase endpoint development productivity. Confirming this is the fact that most participants have finished the task of an endpoint service modeling in less than 30 minutes. Based on this result, we can generalize that it is possible to model a relatively complex endpoint with more than ten services within a day through using our tool support.

6. CONCLUSION AND FUTURE WORK

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

In a realistic enterprise environment, endpoint security requirement may put extra constraints on the validity of a service request. Some of the constraints are role related, so that some services are accessible only to a certain group of users. Others are security policy related, such as restriction on available time or specific pattern required for some service parameters. Also, there are some robustness requirements on SUT for handling endpoint malfunctioning situations. These and other non-functional requirements modeling will be our future work.

7. ACKNOWLEDGMENT

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

8. REFERENCE

[1] 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. 79-94.

[2] Prowell, S. J. 2000. TML: A description language for Markov chain usage models. Information and Software Technology. 42:835-44.

[3] Hine, C., Schneider, J-G, Han, J., Versteeg, S. 2009. Scalable emulation of enterprise systems. Software Engineering Conference, Australian: IEEE. 142-51.

[4] 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. 123-32.

[5] Jayasinghe, D. 2008. Quickstart apache axis2. Packt Publishing Ltd.

[6] Dai, G., Bai, X., Wang, Y., Dai, F. 2007. Contract-based testing for web services. Computer Software and Applications Conference, COMPSAC 31st Annual International: IEEE. 517-26.

[7] Kelly, S., Tolvanen, J. P. 2008. Domain-Specific Modeling: Enabling Full Code Generation. Wiley.

[8] 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. In Press. ODI=https://sites.google.com/site/teevmlase/.

[9] W3C. 2001. Web Services Description Language (WSDL) 1.1. World Wide Web Consortium.

[10] Thompson, H. S., Beech, D., Maloney, M., Mendelsohn, N. 2004. XML schema part 1: structures second edition. W3C Recommendation.

[11] Sousa, T. B. 2012. Dataflow Programming Concept, Languages and Applications. Doctoral Symposium on Informatics Engineering.

[12] Davis, F. D. 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS quarterly. 319-40.

[13] Brooke, J. 1996. SUS-A quick and dirty usability scale. Usability evaluation in industry. 189:4-7.