2016 IEEE Symposium on Visual Languages and Human-Centric Computing, Cambridge, UK, Sept 4-8 2016, (c) IEEE 2016.

A Domain-Specific Visual Modeling Language for 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, Iman Avazpour, Mohamed Abdelrazek
School of Information Technology
Deakin University, Burwood, VIC 3125, Australia
j.grundy@deakin.edu.au
iman.avazpour@deakin.edu.au
mohamed.abdelrazek@deakin.edu.au

***Abstract*—Software integration testing plays an increasingly important role as the software industry has experienced a major change from isolated applications to highly distributed computing environments. Conducting integration testing is a challenging task because it is often very difficult to replicate a real enterprise environment. Emulating testing environment is one of the key solutions to this problem. However, existing specification-based emulation techniques require manual coding of their message processing engines, therefore incurring high development cost. In this paper, we present a suite of domain-specific visual modeling languages to describe emulated testing environments at a high abstraction level. Our solution allows domain experts to model a testing environment from abstract interface layers. These layer models are then transformed to runtime environment for application testing. Our user study shows that our visual languages are easy to use, yet with sufficient expressive power to model complex testing applications.**

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

I. INTRODUCTION

Enterprise software systems have become more complicated and interconnected to provide composite services to their consumers. The behavior of these systems are no longer governed by their own system components, but also driven by its increasingly complex interactions with other systems in its operational environment. This means that testing these interconnections in a realistic production environment is critical.

Many approaches have been proposed to provide executable, interactive representations of deployment environments, e.g. method stubs, mock objects and existing emulation solutions [1-5]. These approaches, however, introduce a large implementation overhead for developers, especially for large-scale heterogeneous environments. Consequently, we have developed a novel domain-specific Visual Modeling Language for Testing environment emulation (TeeVML) and tool support to reduce the development cost of testing environment. In this paper, we present a suite of three visual languages used for endpoint signature, protocol and behavior layers modelling.

978-1-5090-0252-8/16/$31.00 ©2016 IEEE

The remainder of this paper is organised as follows: Section II motivates our work with an example case study. It is followed by an introduction of the approach and design of TeeVML in Section III. In Section IV, we show how a testing endpoint can be modeled and then describe the steps to convert endpoint models into testing runtime environment in Section V. In Section VI, we evaluate the tool and discuss the key findings from the results of a user survey. This is followed by a review of related work in section VII. Finally, we conclude this paper and identify some problems for future work in Section VIII.

II. MOTIVATION

We use an example case study of a bank system. Consider a new Internet banking system is to be integrated to a corporate banking environment. The bank has a core system to support in-house daily banking operations, servicing its customers. For the purpose of expanding its customer base and reducing operational costs, the bank plans to introduce an Internet banking system for allowing its customers to do bank account queries and transactions by themselves. Due to operational issues and data security considerations, all bank accounts and customer data will be kept in the core system. The Internet banking system must interact with the core system intensively for data exchange. To ensure interconnectivity and mutual operability between the core system and the new Internet banking system, integration testing must be conducted before putting the new system in place.

For this study, we treat the Internet banking system as System Under Test (SUT), and the core banking system as testing application (or call endpoint) to be emulated. The endpoint must provide interconnectivity and interoperability testing functionalities, which should mimic its real application services. Let's assume the main endpoint characteristics, as described below:

- An endpoint only considers the external behaviors (or call services) of the real system, and all internal implementations will be ignored;
- An endpoint only provides a subset of the real system invoked by the SUT;
- An endpoint should be able to detect all SUT interface defects, identify their types and origins.

Currently, Kaluta system creates testing endpoints by manually coding the endpoint using Java and Haskell languages [5]. A typical testing endpoint includes an endpoint type dependent message processing engine module, and an endpoint type independent network interface module. Thus, cost of emulating a testing environment will be linearly increased along with additions to endpoint types. UML Testing Profile (UTP) is a model-driven testing approach, providing a specification-based means to systemically define tests for static and dynamic aspects of systems modeled in UML [6]. However, the UTP is for server-side system testing, rather than developing a testing environment for client-side application integration testing.

III. Our Approach

Our TeeVML is based on a new layered software interface description framework, and a suite of Domain-Specific Visual Languages (DSVL) are developed for modeling each interface layer of an endpoint. From a high-level point of view, our approach consists of an endpoint modeling environment supported by TeeVML and an Axis2 Web Service runtime environment. Domain experts work on the modeling environment to create endpoint models; and these models are transformed to target source codes automatically by code generators. Fig. 1 depicts TeeVML modeling and runtime testing environments.

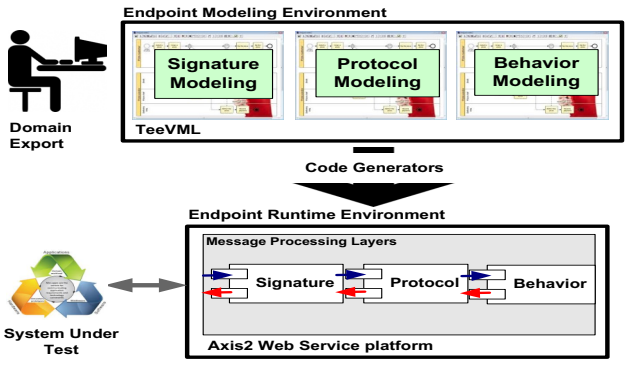


Fig. 1. TeeVML endpoint modeling and runtime testing environments

A. Software Interface Description Framework

To design our framework, we analyzed three popular applications to gather as much domain knowledge as possible, by identifying similar objects and operations. These applications were:

- A public cloud Customer Relationship Management (CRM) application for providing sales process automation; the CRM needs to be integrated with an in-house Enterprise Resource Planning (ERP) system;
- A LDAP server for enterprise resources management; normally, a new application must be integrated with an enterprise LDAP server first, before it can be put to use in production environment;
- An e-commerce sample application jPetStore, originally developed by Sun Microsystems for illustrating the usage of J2EE technology and best practices in system design.

From this domain analysis, we proposed a new layered software interface description framework, which is a modification of Han's comprehensive interface definition framework for software components [7]. Our framework abstracts software interface into three horizontal and two vertical layers. Horizontal layers include signature (message format and structure), protocol (valid service temporal sequence) and behavior (service request process and response generation). Vertical layers include data store (data persistence and manipulation) and Quality-of-Service (QoS) (or call non-functional requirement). A SUT service request is processed horizontally by the endpoint in 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 checking 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 interface layer.

TABLE I. PON PRINCIPLE AND VISUAL NOTATION DESIGN RULE

PoN Principle	Description	TeeVML Visual Notation Design Rule
Semiotic clarity	There should be 1:1 correspondence between semantic constructs and graphical symbols.	All the visual symbols have 1:1 correspondence to their referent concepts.
Perceptual discriminability	Different symbols should be clearly distinguishable from each other.	All symbols use different shapes as their main visual variable, plus redundant coding by colours or textures.
Semantic transparency	Visual representations whose appearance suggests their meaning.	We have used as many icons as possible to represent visual symbols, and minimised the use of abstract geometrical shapes.
Complexity management	There should be some explicit mechanisms for dealing with complexity	Hierarchical visual presentations and information hiding are used to manage diagrammatic complexity.
Cognitive integration	There should be some mechanisms to support integration of information from different diagrams.	Service nodes in behavior model import request and response parameters from signature model.
Visual expressiveness	Use the full range and capacities of visual variables.	We have used various visual variables, such as shape, colour, orientation, texture, etc. when designing visual symbols.
Dual coding	Use text to complement graphics.	Most visual symbols have a textual annotation.
Graphic economy	The number of different graphical symbols should be cognitively manageable	A key design consideration is to minimise the number of visual symbols.
Cognitive fit	Use different visual dialects for different tasks and audiences	Not applicable.

Visual notations form an integral part of a domain-specific visual language, and have a profound effect on the usability and effectiveness of the visual language [8]. To evaluate the "goodness" of visual notations, Larkin et al. defined the cognitive effectiveness [9] as "the speed, ease, and accuracy with which a representation can be processed by the human mind". To achieve the cognitive effectiveness, Moody proposed

the Physics of Notations (PoN) [8], and defined a set of principles to evaluate, compare, and construct visual notations. To improve our DSVL's usability and development productivity, we have applied these PoN principles to our visual notation design. Table I lists the PoN principles and presents our visual notation design rules.

In the following paragraphs, we introduce the visual notation design for a suite of visual modeling languages to model endpoint horizontal layers. The data store layer supports the behavior layer, and QoS will be our future work.

B. Signature DSVL Design

To have a concise presentation view of signature model, we have developed a three-level signature DSVL. The top level signature DSVL uses W3C WSDL 1.1 specification [10] as its metamodel and consists of the five entity types defined in the WSDL specification and two relationships to link them together. The middle level operation DSVL is used to specify request and/or response message(s) in an operation (or call service). The bottom level message DSVL defines all element types used in signature modeling; its metamodel is based on W3C XML Schema 1.1 [11]. Using multi-level modeling approach allows the lower level models to be reused by upper level models. The signature DSVL visual notations are presented in Table II.

TABLE II. SIGNATURE DSVL VISUAL NOTATIONS

Visual Symbol	Description [10]
Service Name: BankService	Service: contains a set of system functions (services) exposed to service consumers through Web-based protocols.
Port Name: BankPort Address: bank.endpoint.com	Port: provides address or connection point to service entity; it has the composite relationship with service entity and associate relationship with binding entity.
Binding Name: BankBinding	Binding: specifies the interface and defines SOAP binding style and transport; it binds portyype entity to port entity through associate relationship.
PortType Name: BankPortType	PortType: contains a set of operations a Web service can perform; it has the composite relationship with operation entity and associate relationship with binding entity.
Operation Name: logon Pattern: In-Out	Operation: is corresponding to a service, and has properties as name and pattern; pattern can be in-only, in-out or out-only.
	Composition relationship: is used to link a main entity to its sub-component entities.
	Association relationship: is used to link two associated entities.
Message Name: logon_request Label: In	Message: specifies messages in operation entity; message has properties as element and label, and label can be in or out.
Complex Element ID: a Name: userid Type: int Mandatory: T	Complex Element: specifies a complex element in a message; it has properties as element name, type and mandatory.

C. Protocol DSVL Design

We used an Extended Finite State Machine (EFSM) to describe endpoint protocol behaviors. The EFSM metamodel is depicted in Fig. 2. We added one entity type and two entity properties (marked yellow in Fig. 2) to a standard operation-driven finite state machine for enriching our protocol modeling with dynamic aspects. The entity type is the *InternalEvent*, which allows users to define state transitions triggered by time event. One of the entity properties is the *StateTransitionConstraint* of transaction entity, and it is used for specifying either static or dynamic constraints on state transition function. Another one is the *StateTimeProperty* of state entity, which is used to simulate endpoint synchronous process and unsafe operation (not an idempotent operation, which will produce the same results if executed once or multiple times). Table III lists all the visual notations used in our protocol DSVL.

The protocol modeling is only applied to statefull applications. This is because endpoint uses its current state to validate the next coming service. If an endpoint is a stateless application, its protocol modeling will be skipped.

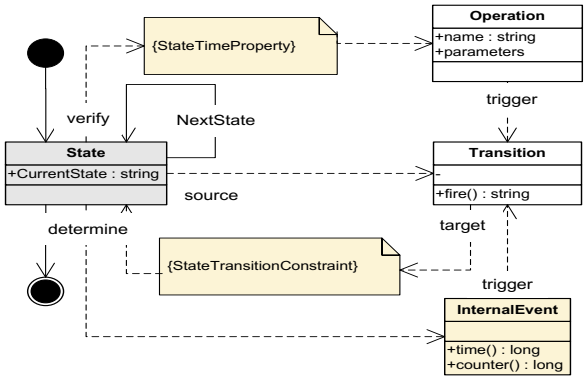


Fig. 2. Protocol DSVL metamodel (EFSM)

TABLE III. PROTOCOL DSVL VISUAL NOTATIONS

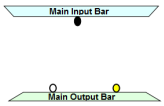
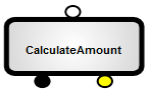
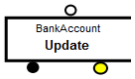
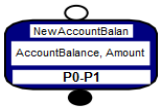
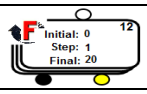
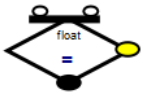
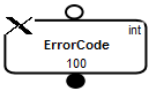
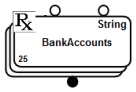
Visual Symbol	Description
logon	State: presents endpoint state, which normally uses service as its default name; for emulating time related scenarios, the state type has a dialog box for allowing users to define state properties.
Home	Home state: is a special endpoint state, representing endpoint in active status.
Idle	Idle state: is a special endpoint state, representing endpoint in inactive status.
logon	Constraint transition relationship: links a *from* state to a *to* state for representing a state transition; it has a dialog box for allowing users to define constraint properties.
logon	Transition relationship: links a *from* state to a *to* state for representing a state transition.
	Loop: is used to define a repeat state transition from a *from* state to a *to* state.
60	Timeout relationship: links two states to represent endpoint state change if no valid service request is received within a defined timeout period.

D. Behavior DSVL Design

Endpoint behavior DSVL was designed based on dataflow programming paradigm [12]. 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 output connector for the next nodes in the chain.

To handle complicated business logics, we designed our behavior DSVL as hierarchical tree structure. Each node may contain several sub nodes (or call methods), and each of the sub nodes implements a specific task. 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 manage diagrammatic complexity problem. At the bottom level, a node consists of some primitive programming constructs for performing operations on data and flow controls for directing execution sequence. We reused the message DSVL of signature DSVL to define data store tables, and a slave table can be defined by specifying the foreign key field data type as undefined.

TABLE IV. BEHAVIOR DSVL VISUAL NOTATIONS

Visual Symbol	Description
	Service node: represents a service provided by an endpoint; it receives request from a SUT, processes it and generates response to the SUT.
	Input and output bars: are used to specify input and output parameters of a service node; all programming constructs will be placed between them to convert input into output; a normal output port (white circle) and an exceptional output port (yellow circle) are on the output bar.
	Data store definition: is used to define data store tables by specifying each table field and field properties.
	Node: is similar to service node, but is used for performing a specific task; a node implementation can either end in successful or failure, this status will decide the next node to be executed in the chain.
	Data store operator: is a primitive construct to retrieve and manipulate data records in data store.
	Evaluator: is a primitive construct to perform an arithmetic operation; the first line is the variable name of the evaluator, the second line lists all variables to be used, and the third line gives the arithmetic formula.
	Loop: is a primitive construct to specify repeated execution of a block of codes for pre-defined times.
	Conditional operator: is a primitive construct to test two input parameters for deciding execution flow; if the testing result is true, the flow will follow the black out port at the bottom; otherwise, the yellow out port at the right will be followed.
	Variable: is a primitive construct to represent a variable with various data types; the variable name is shown at the middle, data type at the upper-right corner and value at the bottom.
	Variable array: is a primitive construct to represent variable array; the array name is shown at the middle, data type at the upper-right corner and array size at the lower-left corner.

At the top level of the node tree structure, discrete service nodes are used to represent the services provided by an endpoint to its SUT. To prevent the data inconsistence between behavior model and signature model, each of the service nodes imports the request and response parameters from the same endpoint signature model. The service nodes can be collapsed to reduce complexity of the diagram. Table IV lists the main visual notations of our behavior DSVL.

IV. EXAMPLE USAGE[1]

A. Business Case

Here, we reuse the banking system of the motivation section and show how a testing endpoint can be modeled by TeeVML. For simplicity, we assume that the core banking system provides only six services to the Internet banking system: session management services *logon* and *logout*, a query service *searchaccount*, and three transaction services *deposit*, *withdraw* and *moneytransfer*. We describe the endpoint three interface abstraction layers as below:

Signature – All the services use in-out operation pattern, except for *logout* service that uses in-only pattern. The *logon* request has a message ID as mandatory field, and two fields for username and password as optional fields. A SUT can logon to the endpoint either in a secured or an insured session, depending on whether the username and password fields are provided or not. All the transaction service requests have a mandatory amount field, which must be equal or greater than zero. All responses contain an optional error code and error message fields for reporting defect types and defect details. For query and transaction services, the response message also includes an optional account balance field to return the account status.

Protocol -- Whenever the endpoint receives a *logon* request from its SUT, it transitions from idle state to home state and an interactive session starts. If the endpoint is in secured session, it changes to the service name state whenever receiving a service request. Otherwise, only the query service request is allowed, and its state will change to *searchaccount* state. The endpoint state transition can also be driven by internal time event. If the endpoint current state is timeout, its state will be changed from a service state to the home state or from the home state to the idle state. In addition, the endpoint processes all the services in synchronous mode, and all the transaction services are considered as unsafe services.

Behavior – To start an interactive session, the *logon* request must be authenticated against stored user account records, and the request will be rejected if the user ID or the pair of username and password does not match any of those records. For all the transaction and query services, account name and account number are used to search for a bank account in data store, and the account balance will be retrieved. For the *withdraw* and *moneytransfer* services, the retrieved balance must be verified to be equal or greater than the transaction amount. For all the transaction services, we must calculate the new balance amount first, and then write the new balance back to the same bank account record.

B. Signature Modelling

As operation (or call service) definition is the main activity of an endpoint signature modeling, we use the operation *logon*

[1] The example application source codes and a recorded demo video are available online: https://sites.google.com/site/teevmlvlhcc/.

as an example to show how such an operation can be modeled by TeeVML. We start to model the operation by assigning the operation name property as logon and selecting in-out from the pattern field drop-down list. Then, operation DSVL is used to specify the both logon_request and logon_response messages in the operation. The message label is "in" for the request message and "out" for the response message.

Message elements are defined by using message DSVL. The request message contains three elements (or call parameters): userid, username, and password, and they are placed in the message by the ID field in alphabetic order. The userid data type is defined as integer and the other two are string, by selecting the corresponding values from the type field drop-down list. An element can either be mandatory or optional by selecting the mandatory field checkbox. Userid is a mandatory element and username and password are optional elements. Similarly, we can define the response message of the *logon* operation with three elements: secure, errorcode and errormessage. Fig. 3 illustrates the hierarchical signature model of the core banking system endpoint, including top-level signature model, *logon* operation, and request and response messages.

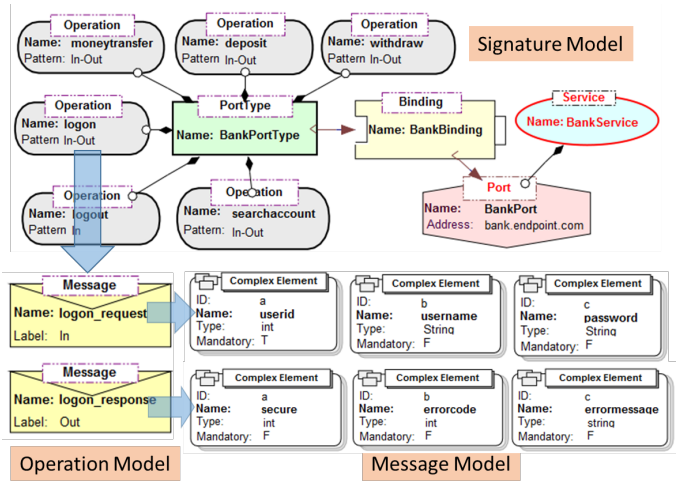


Fig. 3. Example endpoint signature model

C. Protocol Modelling

Endpoint protocol modeling starts from defining its interactive session. A session begins, when the endpoint at idle state receives a *logon* request and changes to home state. This state transition can be represented by using the logon transition relationship to link the idle state to the home state. In opposite direction, a session will end, when the endpoint receives a *logout* request at the home state. A session can also be terminated by timeout relationship, linking the home state to the idle state.

Once the endpoint is in a session, it is ready to receive service requests from its SUT. Since a *searchaccount* request will trigger state transition without any constraint, a standard transition relationship can be used to link the home state to *searchaccount* state. However, a constraint transition relationship must be used to link the home state to all transaction states, as transaction services are only allowed in secured sessions. A constraint transition relationship is defined by setting the constraint transition property to a nonnullable value for the username field in *logon* operation. Fig. 4 illustrates the protocol modeling diagram of the banking system endpoint.

To simulate synchronous services, we open the state property dialog box, select synchronous operation checkbox, and provide a number in seconds to the process time field. Similarly, unsafe services are simulated by selecting the unsafe operation checkbox and filling the transmission time field with a number in seconds.

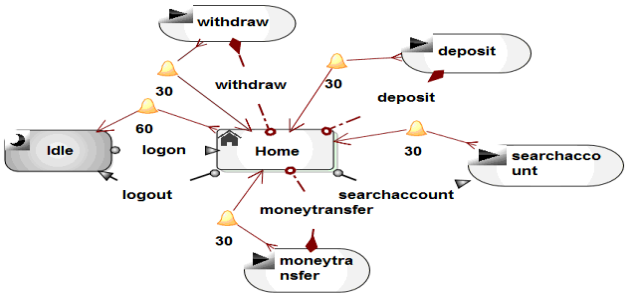
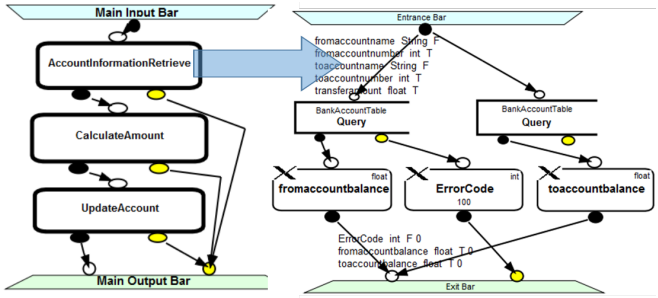


Fig. 4. Example endpoint protocol model

D. Behavior Modeling

We use the *moneytransfer* service node of the banking system endpoint and its sub node *accountinformationretrieve* to explain how the behavior DSVL is used. The *moneytransfer* service node contains three nodes: (1) *accountinformationretrieve* to retrieve the account balance from both "from" and "to" bank accounts, (2) *calculateamount* to calculate the new balances for these two accounts, and (3) *updateaccount* to update the new balance back to persistent data store. Fig. 5a depicts the *moneytransfer* service node structure, execution sequence and dataflow between the input/output bars and nodes.



[a] Moneytransfer service node [b] Accountinformationretrieve node

Fig. 5. Example endpoint behavior model

As the *accountinformationretrieve* is the first node to be executed, it will directly take the service node input parameters: fromaccountname, fromaccountnumber, toaccountname, toaccoutnnumber and transferamount. The output parameters to the next node are: fromaccountbalance, toaccountbalance and ErrorCode. These input and output parameters are defined when we create the input and output bars. We search "from" account by the fromaccountname and fromaccountnumber parameters. If the account is found, the account balance will be retrieved and assigned to the variable fromaccountbalance. Similarly, "to" account balance will be retrieved by using the toaccountname and toaccoutnnumber parameters, and the balance will be assigned to the toaccountbalance variable. If the "from" and/or "to" accounts cannot be found, an error occurs and integer

number 100 is assigned to the ErrorCode variable. If the node is executed successfully, both the fromaccountbalance and toaccountbalance variables will be placed on the normal output port. Otherwise, we will put the ErrorCode variable to the exceptional output port. Fig. 5b shows how the *accountinformationretrieve* node is constructed.

V. Implementation

Our Domain-Specific Modeling (DSM) approach in TeeVML is based on the four abstraction-level architecture defined by the OMG Meta Object Facility (MOF) [13]. DSVL development includes language visual notation design and code generator implementation using MetaEdit+ [14].

As part of our TeeVML tool, we have developed two types of code generators for each DSVL by using MetaEdit Report Language (MERL). The first one is to transform models to target source codes: signature model to WSDL 1.1 XML file, and protocol and behavior models to Java classes. The second one is for converting models to SQL scripts for table creation. The signature WSDL file will be further transformed to Web Service engine by using Axis2 code generator utility wsdl2java, acting as our runtime environment.

A complete testing environment consists of a Tomcat application server for hosting the SOAP service provide by endpoint, Axis2 web service engine for SOAP message processing, a protocol class for protocol logic processing, several behavior model classes for generating services to SUT, and MySQL database for storing static and dynamic persistent data. Once all models have been transformed to source codes, we use Apache ant builder to build the endpoint SOAP service, then load the service to the Tomcat application server.

VI. User Evaluation

The developed TeeVML was evaluated by a two-phase user survey. In the first phase, we conducted a study with testing experts to examine what features of TeeVML they valued in testing endpoint emulation, and what functionalities such a tool should provide. In the second phase, we evaluated TeeVML's usability by collecting software developers' opinions on their experience with the tool. Specially, we wanted them to compare TeeVML with a third generation language they were familiar with.

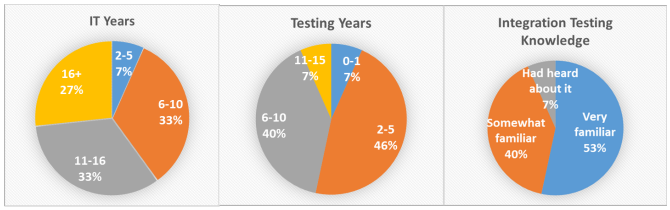
A. Experiment Setup

Phase One survey was conducted by interviewing participants. We used a PowerPoint presentation to introduce TeeVML to them, and explained what testing functionalities were required for a testing endpoint and how such an endpoint could be created by use of earlier versions of TeeVML tool. The interview lasted approximately 40 minutes. After the interview, all participants were asked to do an online survey. Since our target audiences for this phase were required to have extensive experience in software testing, we were only able to get 16 participants to take part in this phase. Fig. 6a summarizes Phase One participants IT and software testing experience. As indicated in the charts, most participants were testing experts with solid experience on software development and testing.

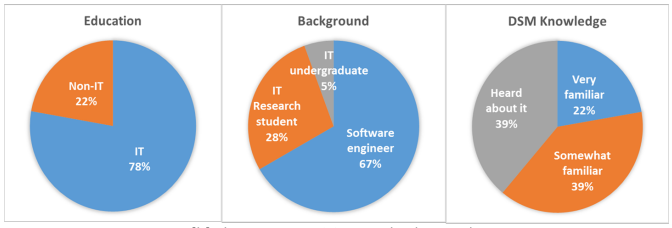
Phase Two survey was a three-step process. First, participants watched a recorded video, to introduce TeeVML and show the steps to model a testing endpoint. Second, the participants were assigned a task to model a simple endpoint example. The task was performed by using TeeVML running on a laptop PC. Finally, all participants were asked to do an online survey. The duration for Phase Two was 60 minutes on average. Overall 21 software developers and IT research students took part in the survey. Two participants could not finish the task, due to their personal reasons. Fig. 6b provides Phase Two participants IT background information.

B. Result Analysis and Discussion

We designed total 58 questions for both Phase One and Phase Two to cover various aspects of our TeeVML, and questions types were 5-point Likert Scale (5 to 1 representing strongly agree to strongly disagree) and multi-choice. Due to space limitation, we only selected some of them for this paper evaluation results presentation, and the full result reports are available online[2]. We have counted frequency of participant responses to measure degree of acceptance to a particular question statement. For the 5-point Likert Scale questions, in favour responses encompass the answers of either 5 or 4 for a positive question, or 1 or 2 for a negative question.



[a] Phase One participant IT and software testing experience



[b] Phase Two participant IT background

Fig. 6. User survey participants' demographics

1) Phase One Evaluation Results

Table V lists the selected questions and participants' responses. Here we analyse the survey results as follows.

Usefulness – Q8 reflects the overall usefulness of testing endpoint for conducting integration testing. The responses to this question are quite positive with 14 out of 16 participants in favour of usefulness of the tool. We can see that the protocol modeling (Q21) has the highest positive response rate and the non-functional requirement (Q30) the lowest. We believe one of the main reasons why most participants want to have protocol testing is that most applications do not have well-documented protocol specification. Therefore, protocol related defects can only be found by conducting integration testing.

[2] Refer to the web site: https://sites.google.com/site/teevmlvlhcc/.

Testing functionalities – Q9 is a multi-choice question for evaluating the usefulness and completeness of functionalities that an endpoint should provide to its SUT. Except for the four features already implemented, we assigned an "Other" item for allowing participants to specify any other useful features, our TeeVML does not support now. Only one participant selected the item "Other", and suggested to provide performance test under different scenarios. From these responses, we can conclude that most participants are satisfied with the functionalities that TeeVML provides.

TABLE V. PHASE ONE SURVEY 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
Q17	It is useful for an emulated testing environment to provide signature testing functionality to its system under test.	7	7	1	1	0
Q21	It is useful for an emulated testing environment to provide interactive protocol testing functionality to its system under test.	12	4	0	0	0
Q25	It is useful for an emulated testing environment to provide interactive behavior testing functionality to its system under test.	6	8	1	1	0
Q30	It is useful for an emulated testing environment to provide non-functional requirement testing features to its system under test.	2	11	3	0	0

No	Question	Statement	Frequency
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 behavior	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

Why or Why not use endpoint – Q13 and Q14 are multi-choice questions, and list four reasons and five concerns why or why not users want to use testing endpoints. Surprisingly, the top reason for users to use endpoints is the early detection of interface errors, rather than savings on investment and development effort. Early interface defects detection is particularly important, when an application is developed by a third party and environment systems are completely inaccessible. Q14 reflects most participants' concern on result reliability. We believe the main reason is that in new endpoint development process endpoints are modeled rather than coded. Therefore, it is important to reliably model emulated testing environments.

2) Phase Two Evaluation Result

For Phase Two, we evaluate the overall usability of TeeVML using Software Usability Scale (SUS) [15]. Table VI presents the SUS survey results by frequencies. We have received quite positive responses from the survey participants, with average 16.2 in favour. Particularly, the Q17 has received in favour response from all participants, followed by Q12, Q14 and Q16. The lowest score is Q15 that has less than half (8 participants) in favour. The assigned task was actually modeling related and a certain level of modeling skill was required. However, the survey participants were not mostly experts in domain specific modeling (refer to Fig. 6b). So, some of them might have needed support for modeling related techniques.

TABLE VI. FREQUENCY TABLE OF SOFTWARE USABILITY SCALE

No	Statement	Frequency				
		5	4	3	2	1
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	11	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

TABLE VII. INTERFACE LAYER USABILITY QUESTIONS AND RESPONSES

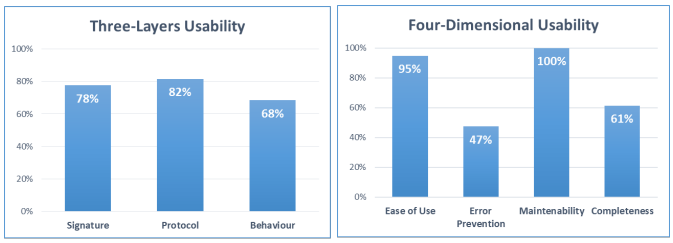
No	Question	Frequency				
		5	4	3	2	1
Q27	Endpoint signature is easily modeled by the tool.	9	9	0	0	1
Q29	It is easy to make changes to message signature model.	13	6	0	0	0
Q30	It is easy to make errors or mistakes during message signature definition.	0	3	7	5	4
Q31	It is capable of defining all types of message signatures you have seen.	2	11	6	0	0
Q33	Endpoint protocol is easily modeled by the tool.	12	7	0	0	0
Q35	It is easy to make changes to interactive protocol model.	13	6	0	0	0
Q36	It is easy to make errors or mistakes during interactive protocol definition.	1	1	5	6	6
Q37	It is capable of defining all interactive protocol scenarios you have seen.	4	8	6	1	0
Q39	Endpoint interactive behavior is easily modeled by the tool.	3	14	1	1	0
Q41	It is easy to make changes to interactive behavior model.	10	9	0	0	0
Q42	It is easy to make errors or mistakes during interactive behavior definition.	1	1	11	6	0
Q43	The tool has sufficient expressive power for creating behavior model with accurate outputs.	1	9	8	1	0

Table VII presents Phase Two survey questions and responses from three interface layers: signature, protocol and behavior, and four usability dimensions of each layer: ease of use, maintainability, error prevention and completeness. Fig. 7

summarizes the in favour responses for each layer and usability dimension from Table VII.

From the layers' viewpoint (refer to Fig. 7a), protocol DSVL has the highest usability and behavior the lowest. This result is in coincidence with what we have expected. Endpoint protocol modeling is simple and easy, and only four relationship types are used to specify various state transitions. In contrast, behavior modeling must deal with complicated logic processing, involving data manipulation, flow control, data store access, etc.

For the usability dimension (refer to Fig. 7b), maintainability has received in favour response from all participants, and is followed by ease of use. High maintainability is one of the key motivations for us to select a DSVL approach, since any changes to endpoint can be done by modifying models only and engaging in coding is not required. More than half of participants were not satisfied with the error prevention mechanism provided by TeeVML. Although TeeVML supports most DSVL specific error prevention mechanisms, it does not currently provide comprehensive error and type checking.



[a] [b]

Fig. 7. Summary of in favour responses for different layers and dimensions

VII. Related Work

There are a wide variety of domain-specific languages. They can be widely used languages for a specific technical domain, such as HTML for web pages, SQL for relational databases, and WebDSL for web applications [16]. Or, they can narrowly focus on a specific business domain, such as MaramaEML for business process modeling [17], SDL for supporting statistical survey process [18], and LabVIEW for electronic circuit testing design [19]. In contrast, we use a set of domain-specific visual modeling languages tailored to modeling signature, protocol and behavior layers of endpoints.

A testing endpoint is developed from its external behavior, communicating with other software components. Han first proposed a rich interface definition framework [7] with layers: signature, configurations, semantics, constraints, and a quality aspect across all these layers. Han's framework defines how to select and reuse a software component, not just based on static component signature, but also on other runtime aspects as well. From a service viewpoint, Beugnard et al. defined a four-level software component contract template with increasingly negotiable properties along with the levels [20]. Our approach on the other hand, focuses on how a request is to be processed by an endpoint in a layered manner.

For the protocol modeling, some researchers used a finite state machine [21, 22] or a formal protocol specification [23, 24] to validate message sequence for different endpoint states. However, Wehrheim et al. argued that the use of service name alone might not be sufficient to trigger a state transition for a realistic endpoint [25]. To deal with the so-called incomplete protocol specification, [25] developed an EFSM-based protocol modelling calculus for specifying service parameters and return values as runtime constraints. Although, various notions for protocol specification have been suggested, there are still some issues to be solved. One is lack of concrete implementation solutions to capture endpoint runtime aspects. Another one is textual form they used for writing state transition rules, and this will make protocol modeling difficult.

Software components interface behaviors can be modeled either externally or internally. Software behavioral interface specification [26] and programming from specification [27] are the external approaches, they model interactive behaviors by defining pre/post conditions to bind both service consumer and service provider. As internal approaches, Business Process Model and Notation (BPMN) [28] and DataFlow Programming (DFP) [12] provide graphical notations for specifying internal data processes and flow controls. In general, external approaches and BPMN require extensive modeling and programming work. While, DFP languages are ease of use with user-friendly interface. But, they are less expressive and only suitable for a specific domain. In contrast to these approaches, our behavior DSVL is ease of use by dragging-and-dropping visual symbols. For handling complicated business logics, hierarchical nodes tree structure is adopted.

UML is a widely used general purpose modeling language, focusing on the definition of system static and dynamic behaviors. Specifically related to our work, UML provides: (1) a testing profile to provide a generic extension mechanism for the automation of test generation processes [6], (2) state charts to simulate finite-state automaton [29], and (3) activity diagrams to graphically represent workflows of stepwise activities and actions [30]. Two main problems with using UML to define new modelling languages [31] are that it is usually hard to remove parts of UML that are not relevant or need to be restricted in a specialized language; and all the diagram types have restrictions based on the UML semantics.

VIII. Conclusion and Future Work

In this paper, we have presented a suite of domain-specific visual languages for testing environment emulation. The language consists of three DSVLs for each interface layer. An endpoint is modeled using these three abstraction levels. By this layered approach, our TeeVML supports partial endpoint development, where a testing endpoint may have only one or two of these layers to meet SUT testing requirement. We have conducted a survey to evaluate the tool's functionality and usability. The survey results demonstrated acceptance of the tool among software testing experts and developers and further improvement areas, such as error prevention.

A fully functional testing endpoint must include testing Quality of Service (QoS) provided to SUT. The QoS DSVL should be able to model performance, reliability, security and other non-functional attributes. We are investigating the benefits of building another DSVL specifically for model syntax checking before transforming models to target sources as our current future work focus.

ACKNOLEDGEMENTS

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

REFERENCES

[1] P. B. Gibbons, "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," IEEE Transactions on Software Engineering, vol. 13, pp. 77-87, 1987.

[2] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, objects," presented at the In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Canada, 2004.

[3] J. Yu, J. Han, J.-G. Schneider, C. Hine, and S. Versteeg, "A virtual deployment testing environment for enterprise software systems," presented at the Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, Italy, 2012.

[4] M. Du, J.-G. Schneider, C. Hine, J. Grundy, and S. Versteeg, "Generating service models by trace subsequence substitution," presented at the Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, Canada, 2013.

[5] C. Hine, J.-G. Schneider, J. Han, and S. Versteeg, "Scalable emulation of enterprise systems," in Software Engineering Conference, Australian, 2009, pp. 142-151.

[6] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch, "The UML 2.0 testing profile and its relation to TTCN-3," in Testing of Communicating Systems, ed: Springer, 2003, pp. 79-94.

[7] J. Han, "Rich Interface Specification for Software Components," Peninsula School of Computing and Information Technology Monash University, McMahons Road Frankston, Australia, 2000.

[8] D. L. Moody, "The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering," Software Engineering, IEEE Transactions on, vol. 35, pp. 756-779, 2009.

[9] J. H. Larkin and H. A. Simon, "Why a Diagram is (Sometimes) Worth Ten Thousand Words," Cognitive Science, vol. 11, pp. 65-100, 1987.

[10] E. Christensen, F. Curbera, and G. Meredith, "Web Services Description Language (WSDL) 1.1. W3C," Note 15, 2001, www. w3. org/TR/wsdl, 2001.

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

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

[13] OMG, "Meta Object Facility (MOF) Specification," ed: The Object Management Group, 2000.

[14] S. Kelly, "Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM," in 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development, 2004.

[15] J. Brooke, "SUS-A quick and dirty usability scale," Usability evaluation in industry, vol. 189, pp. 4-7, 1996.

[16] E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," in Generative and Transformational Techniques in Software Engineering II. vol. 5235, R. Lämmel, J. Visser, and J. Saraiva, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 291-373.

[17] L. Li, J. Grundy, and J. Hosking, "A visual language and environment for enterprise system modelling and automation," Journal of Visual Languages & Computing, vol. 25, pp. 253-277, 8// 2014.

[18] C. H. Kim, J. Grundy, and J. Hosking, "A suite of visual languages for model-driven development of statistical surveys and services," Journal of Visual Languages & Computing, vol. 26, pp. 99-125, 2015.

[19] J. Travis and J. Kring, LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition). Upper Saddle River, NJ, USA Prentice Hall PTR, 2006.

[20] A. Beugnard, J.-M. J, l. Plouzeau, and D. Watkins, "Making Components Contract Aware," Computer, vol. 32, pp. 38-45, 1999.

[21] L. De Alfaro and T. A. Henzinger, "Interface automata," ACM SIGSOFT Software Engineering Notes, vol. 26, pp. 109-120, 2001.

[22] U. Endriss, N. Maudet, F. Sadri, and F. Toni, "Protocol conformance for logic-based agents," in IJCAI, 2003, pp. 679-684.

[23] F. Plasil, S. Visnovsky, and M. Besta, "Bounding component behavior via protocols," in Technology of Object-Oriented Languages and Systems, TOOLS 30 Proceedings, 1999, pp. 387-398.

[24] Y. Jin and J. Han, "Specifying Interaction Constraints of Software Components for Better Understandability and Interoperability," in COTS-Based Software Systems. vol. 3412, X. Franch and D. Port, Eds., ed: Springer Berlin Heidelberg, 2005, pp. 54-64.

[25] H. Wehrheim and R. H. Reussner, "Towards more realistic component protocol modelling with finite state machines," UNU-IIST, p. 27, 2006.

[26] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. MULLER, and M. Parkinson, "Behavioral interface specification languages," ACM Comput. Surv., vol. 44, pp. 1-58, 2012.

[27] C. Morgan, Programming from specifications: Prentice-Hall, Inc., 1990.

[28] M. von Rosing, S. White, F. Cummins, and H. de Man, "Business Process Model and Notation—BPMN," 2015.

[29] S. J. Zhang and Y. Liu, "An Automatic Approach to Model Checking UML State Machines," in Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on, 2010, pp. 1-6.

[30] M. Dumas and A. H. Ter Hofstede, "UML activity diagrams as a workflow specification language," in ≪ UML ≫ 2001−The Unified Modeling Language. Modeling Languages, Concepts, and Tools, ed: Springer, 2001, pp. 76-90.

[31] A. Abouzahra, J. Bézivin, M. D. Del Fabro, and F. Jouault, "A practical approach to bridging domain specific languages with UML profiles," in Proceedings of the Best Practices for Model Driven Software Development at OOPSLA, 2005.