A Domain-Specific Modeling Approach for Testing Environment Emulation

Jian Liu¹(✉), John Grundy², Mohamed Abdelrazek², and Iman Avazpour²

¹ Swinburne University of Technology, Hawthorn, VIC 3122, Australia
jianliu@swin.edu.au
² Deakin University, Burwood, VIC 3125, Australia
{j.grundy,mohamed.abdelrazek,iman.avazpour}@deakin.edu.au

Abstract. Software integration testing is a critical step in the software development lifecycle, as modern software systems often need to interact with many other distributed and heterogeneous systems. However, conducting integration testing is a challenging task because application production environments are generally neither suitable nor available to enable testing services. Additionally, replicating such environments for integration testing is usually very costly. Testing environment emulation is an emerging technique for creating integration testing environments with executable models of server side production-like behaviors. Aiming to achieve high development productivity and ease of use for business users, we propose a novel domain-specific modeling approach for testing environment emulation. Our approach is based on model-driven engineering, and abstracts software service interfaces, or endpoints, into different request message processing layers. Each of these layers represents a modeling problem domain. To model endpoints, we develop a suite of domain-specific visual languages for modeling these interface layers. To build a testing environment, we have created a supporting toolset to transform endpoint models to executable forms automatically. We provide a set of example scenarios to demonstrate the capabilities of our approach. We have also conducted a user study that demonstrates the acceptance of our approach by IT professionals and business users.

Keywords: Model-driven engineering
Domain-specific visual modeling language · Software integration testing
Testing environment emulation

1 Introduction

1.1 Software Integration Testing

Emerging computing strategies, such as cloud computing and social networking, represent an ongoing shift from monolithic applications to highly distributed, heterogeneous and shared computing environments [1]. Most software systems need to interact with other systems to provide composite services to their clients or end users.
Thus, the performance of a software system is no longer determined only by its own internal components, but is also subject to its increasingly complicated interactions with external systems in its operational environment. This means that for effective testing of a software system, testing interconnections (static communication aspects) and interoperability (dynamic communication aspects) with other systems that it communicates with in its realistic production environment is critical.

System Integration Testing (SIT) is a testing process that exercises a software system’s behaviors when interacting with other inter-connected systems. It tests the interactions between different systems and verifies the proper execution of the system in its deployment environment [2]. To test the interactions of a System Under Test (SUT) with the systems (that we call “endpoints”) in an enterprise environment, the testing environment must provide a test-bed, that encompasses all services of the endpoints the SUT will invoke in the environment.

Endpoints deployed to a testing environment have some unique characteristics. First, a SUT often interacts with many different types of applications in its environment. Therefore, it is desirable that each endpoint development cycle should be short and the development approach should have high development productivity. Second, SIT is normally conducted by testing engineers or business users. Most of them have rich business domain knowledge, but may lack programming skills. They prefer to model endpoints using problem domain concepts, rather than code them using a textual language. Last, endpoints, as server-side applications to provide testing services to their SUTs, do not necessarily provide accurate results under all circumstances. Therefore, we may simplify some internal implementations, in return for quick development.

1.2 Testing Environment Emulation

Testing Environment Emulation (TEE) is an emerging technique to develop SIT environments for SUTs that interact with many external systems. The main idea is to model the interactive behaviors of each system in a environment and replace the systems by instances of the corresponding models in the emulation environment [3]. The goal is to make the emulated testing environment rich enough to “fool” SUTs into behaving as though they are talking to the real external systems. Other components that sit underneath or in the background are ignored from the emulated environment perspective whenever possible. Particularly, an emulated endpoint is a simplified version of a real system with three assumptions:

- As an endpoint is used to provide test-bed for SUT integration testing, only the external behaviors of the endpoint application are considered and its internal implementations will be ignored;
- An endpoint is specifically developed for the integration testing of a SUT. Therefore, a subset of the endpoint application operations invoked by the SUT are provided;
- Serving as a defect detection tool for system debugging, an endpoint should be able to capture all SUT interface defects, together with their types, origins and other information.
The key benefits from using TEE include:

- It provides a production-like test-bed for provisioning of testing functionality to SUTs in a much more cost-effective way than application replication;
- Development of such a testing environment could be quick and easy, as some internal logic implementations and auxiliary modules are ignored;
- The test-bed is easily configured and monitored for performing Quality-of-Service (QoS) aspects testing, such as simulating different numbers of instances of a same endpoint type for performance test;
- Software interface defects can be captured and the defect cause information can be reported.

1.3 A Domain-Specific Approach to Testing Environment Emulation

Domain-Specific Modeling (DSM) achieves high development productivity and ease of use by focusing on a narrowed problem domain, so that specific high-level abstraction modeling languages and supporting toolsets can be created. To develop our DSM approach to emulate testing environment, we proposed a new software interfaces description framework, where software interfaces are abstracted into logically separated signature, protocol and behavior layers. We use modular development approach to model endpoints, and each module represents one interface layer.

Our DSM approach consists of an endpoint modeling environment and a runtime environment to provide testing services to SUTs (see Fig. 1). The modeling environment includes a suite of domain-specific Visual Modeling Languages for Testing environment emulation (TeeVML) to model endpoints in interface layers. The runtime environment is hosted in Axis2 SOAP engine [4], and is generated automatically by

![Fig. 1. Endpoint modeling and runtime environment.](image-url)
transforming the endpoint signature model. Testing service is enabled through Web Service provided by Tomcat Servlet Container [5].

This research paper is an extended version of our MODELSWARD 2017 paper [6], and we add the following contents in addition to enriching the covered topics: (1) a brief discussion on SIT characteristics and the benefits from using a model-driven approach to develop endpoints; (2) a description of our visual language notation design for achieving high usability; and (3) an introduction to our Domain-Specific Visual Languages (DSVLs) for modeling endpoints.

This rest of the paper is organized as follows: Sect. 2 motivates our research by a case study, followed by an introduction to our DSM approach in Sect. 3. In Sect. 4, we briefly discuss our visual symbol design and introduce our TeeVML. We show how an endpoint is modeled and then describe the steps to convert endpoint models into testing runtime environment in Sect. 5. In Sect. 6, 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 Sect. 7. Finally, we conclude this paper and identify some key future work in Sect. 8.

2 Motivation

We select a typical business case of a company integrating its legacy system with a public cloud application and use this case to describe the potential interactions between an endpoint and its SUT. The company currently has an in-house Enterprise Resource Planning (ERP) system (such as PeopleSoft Finance [7]) to support its daily operations. For the purpose of streamlining its sales process and improving operational efficiency, the company plans to introduce a public cloud Customer Relationship Management (CRM) service (such as salesforce.com [8]) as its frontend application. From operation and data security considerations, all company data will be kept in-house in the ERP. Therefore, the CRM application must interact with the ERP system intensively for accessing persistent data and processing business logics.

The activity sequence diagram in Fig. 2 illustrates a typical sales process flow among users, the CRM application and the ERP system. Users access the CRM application for handling their client Purchase Order (PO). For every user request, the CRM must invoke a corresponding ERP operation using Remote Procedure Call (RPC) communication style [9]. Our main interest is on the interactions between the client CRM and the server ERP as described below.

Whenever the ERP receives a logon request from the CRM, it transits from idle state to home state and an interactive session starts. The next valid operation is porequest, followed by inventorycheck. The returned value of inventorycheck will determine whether supplier chain related steps will be executed. If the purchase item has enough stock for the PO, the process flow will jump over the supplier purchasing steps and directly go to paymentrequest. Otherwise, we should go through the purchase steps (#4, #5, #6 and #7) to buy the missing quantity of the item. supplierpoapproval and approvalnotification are iteration operations, informing all approvers one-by-one to give their approvals. If all required approvals for the supplier PO are obtained, the rest
of purchasing steps will be executed in the order as in Fig. 2. Otherwise, the sales process will be aborted without success.

To ensure the interconnectivity and mutual interoperability between the ERP and CRM, SIT must be carried out before putting the CRM in production. For this study, we treat the ERP as the endpoint that we need to develop, and the CRM as the SUT. 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? From these assumptions, we defined three key research questions for guiding the development of our approach:

RQ1 – Can we emulate a functioning integration testing environment capable of capturing all interface defects of an existing or a non-existing system under test from an abstract service model?
RQ2 – Would our model-based approach improve testing environment development productivity, compared to using third-generation languages (e.g. Java) to implement endpoints?
RQ3 – Can we develop a user centric approach, easy to learn and use to specify testing endpoints by domain experts?

Fig. 2. The example ERP and CRM interactions process flow diagram [6].
3 Our Approach

To identify software interfaces common concepts and determine their relationships, we conducted our TEE domain analysis by investigating three applications interacting with their clients. These applications were the ERP system introduced in the motivation section, a LDAP server [10], and a core banking system [11]. These applications represent a variety of application domains in a typical enterprise environment. From this domain analysis, we proposed a layered software interfaces description framework for TEE, and defined interface defect types to be captured by endpoints. Consecutively, we then designed our Domain-Specific Languages (DSLs) to model endpoints in interface layers.

3.1 Software Interfaces Description Framework

There are three reasons for having a software interfaces description framework. First, we need to abstract software interfaces into different interactive aspects, so corresponding DSLs can be developed with a clearly defined problem domain boundary. Second, we can adopt a modular development architecture to model endpoints in layers. We may also be able to model a few versions of an endpoint type for different SUTs. Third, some of these interface modules may be shared among endpoints, if they have the exact same functionality.

Our framework abstracts software interfaces into three logically separated layers:

- **Signature** – following RPC communication style specification, this layer specifies the requests and responses of endpoint operations, their parameters and properties;
- **Protocol** – this layer defines the validity of a temporal sequence of endpoint operations, which can depend on either endpoint states (static protocol behavior) or runtime constraint conditions (dynamic protocol behavior), or both;
- **Behavior** – this layer abstractly describes endpoint internal operation request process and response generation, and the returned values in response messages are used to capture dynamic protocol defects.

A SUT operation request is processed by an endpoint step-by-step from signature, protocol, and down to behavior layers. Whenever an error occurs, the request processing will be terminated. Signature and protocol layers act as message pre-processors for checking the correctness of operation request syntax and temporal sequence, before handing the request over to behavior layer for generating a suitable response message.

3.2 Service Request Defects

To develop DSLs for endpoint layers, we must know all the defect types first. Table 1 lists and describes the possible interface defects types that a SUT request may have. The SUT request defects can be grouped into static and dynamic categories, depending on whether they will always cause interactive failures or under certain runtime conditions only. Normally, a software application has an interface specification to specify its provided operations and their parameters. A client SUT must send its requests to the application in accordance with the interface specification. Otherwise, interface fault
will occur due to a static interface defect. On the other hand, a dynamic defect happens under certain business scenarios. An example is the validity of the next request after `inventorycheck`, which is subject to the inventory result returned by the `inventorycheck` operation. In general, static defects can be found by code review against interface specification and SIT; while dynamic defects can only be captured by SIT.

We do not list any behavior defects in Table 1. This is because a SUT’s obligation is to send correct requests to an endpoint and the way these requests are to be processed is defined internally by the endpoint. The reason why we still model the endpoint behavior layer is that the validity of alternative next operation requests may depend on what values are returned in the response message it has received based on a previous request.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signature</strong></td>
<td></td>
</tr>
<tr>
<td>Sig1</td>
<td>An operation request is not an operation provided by endpoint</td>
</tr>
<tr>
<td>Sig2</td>
<td>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</td>
</tr>
<tr>
<td>Sig3</td>
<td>One or more operation request mandatory parameter(s) is (are) missing</td>
</tr>
<tr>
<td>Sig4</td>
<td>One or more parameters in an operation request is (are) beyond the defined value range of the corresponding endpoint operation</td>
</tr>
<tr>
<td><strong>Protocol</strong></td>
<td></td>
</tr>
<tr>
<td>Pro1</td>
<td>An operation request is invalid for the current endpoint state</td>
</tr>
<tr>
<td>Pro2</td>
<td>An operation request is invalid for the current endpoint state, as one or more parameter(s) violate(s) the defined constraint condition(s)</td>
</tr>
<tr>
<td>Pro3</td>
<td>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)</td>
</tr>
<tr>
<td>Pro4</td>
<td>An operation request is invalid, due to endpoint state transition driven by some internal event, such as time out</td>
</tr>
<tr>
<td>Pro5</td>
<td>An operation request is invalid, as endpoint is in processing a synchronous operation request</td>
</tr>
<tr>
<td>Pro6</td>
<td>An operation request is invalid, as one such request for an unsafe operation (i.e. not an idempotent operation that will produce the same results if executed once or multiple times) has been received by endpoint</td>
</tr>
</tbody>
</table>

### 3.3 Endpoint Metamodelling

A Metamodel defines all concepts and their relationships within a specific application domain. The key semantics and constraints associated with these domain concepts are also specified. The main inputs to our software interface metamodels are the software interface description framework and the software interface defects listed in Table 1.
**Signature Metamodel**

Endpoint signature layer models operations provided by endpoint and their parameters. Each parameter has some static properties, such as name, data type, order and mandatoriness. Some parameters with integer, float or date data type may also have upper and lower limits in dynamic nature.

Web Service Description Language (WSDL) specification describes the public interface exposed by a web operation, including what an operation does, where it resides, and how to invoke it [12]. Figure 3 illustrates the signature metamodel that our signature DSL is based on. The metamodel adopts a three-level architecture design. The top-level DSL (see Fig. 3a) uses WSDL 1.1 specification as its metamodel. It specifies the relationships among a root definition entity and other 5 entities: service, port, binding, porttype and operation. The middle-level Operation DSL (see Fig. 3b) is to define request and/or response message(s) in an endpoint operation. The operation communication pattern determines whether it contains a request message only, a response message only or both request and response messages. The bottom-level DSL (see Fig. 3c) is based on W3C XML Schema 1.1 for defining complex elements in a message.

The signature defects Sig1 to Sig3 in Table 1 can be detected by the Axis2 SOAP engine itself, transformed from signature model. To specify the upper and lower limits of a number or a date element (refer to Sig4 defect in Table 1), we add two properties to element type to detect any invalid request parameters beyond defined value limits.

![Fig. 3. Endpoint signature metamodel [6].](image)
Protocol Metamodel
To capture both static and dynamic protocol defects, we designed an Extended Finite State Machine (EFSM) to enrich endpoint protocol modeling capability (refer to Fig. 4). Our EFSM adds one entity type and two entity properties to a standard operation driven endpoint state transition Finite State Machine (FSM) (the items we added are marked yellow in Fig. 4). 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 for specifying either static or dynamic constraints on state transition function. Another one is the StateTimeProperty of the state entity, which allows users to simulate synchronous and unsafe operations. As endpoint protocol modeling is relatively simpler than other two interface layers, we use a flat view presentation structure.

All protocol defects listed in Table 1 can be detected by an endpoint, modeled using the EFSM model: (1) Pro1 – the operation-driven state transition FSM; (2) Pro2 and Pro3 – the StateTransitionConstraint property; (3) Pro4 – the InternalEvent entity; and (4) Pro5 and Pro6 – the StateTimeProperty. Protocol modeling is only applied to stateful applications. This is because an endpoint uses its current state to validate the next coming operation. If an endpoint is a stateless application, its protocol modeling will be skipped, as all requests to the endpoint must necessarily contain the required information.

![Fig. 4. Endpoint protocol metamodel [6] (Color figure online)](image)

Behavior Metamodel
Our behavior metamodel is based on the Data Flow Diagram (DFD) programming paradigm [13]. We chose this metaphor as it allows for complex specification of behavior models but has also been shown to be understandable by a wide range of
software stakeholders. DFD programming execution model is represented by a directed graph; nodes of the graph are data processing units, and directed arcs between the nodes represent data dependencies. A node starts to process and convert the data whenever it has the minimum required parameters available on its input connector. The node then places its execution results onto its output connector for the next nodes in the chain.

To handle complicated business logic, we designed our behavior DSL using a hierarchical tree structure. The benefits of 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 diagram complexity problem. On the top level of node tree structure, discrete service nodes are used to represent the operations provided by an endpoint. At the bottom level, each node consists of some primitive programming constructs for performing operations on data and flow controls for directing execution sequence.

4 Our Domain-Specific Languages

A DSL realizes the concepts and their relationships defined in its domain metamodel by mapping them to corresponding programming constructs. A Visual Programming Language (VPL) lets users create programs by manipulating programming constructs graphically rather than by specifying them textually. Although there is no fundamental difference in expressivity, visual languages are generally easier to learn and use than textual languages [14]. To achieve ease of use for domain experts, we designed our TeeVML as a VPL.

4.1 Visual Symbol Design

Visual symbols have a profound effect on the usability and effectiveness of a visual language. Visual symbols are human thought representations for facilitating communications and problem solving among individuals. To be most effective in doing this, they need to be optimized for processing by human mind. For this reason to evaluate the “goodness” of a visual symbol, Larkin et al. defined the term cognitive effectiveness as “the speed, ease, and accuracy with which a representation can be processed by the human mind” [15]. The cognitive effectiveness determines the ability of visual symbols to communicate among a wide range of software stakeholders.

To establish a scientific foundation for visual symbols’ design, Moody proposed the Physics of Notations (PoN) and defined a set of principles to evaluate, compare, and construct visual symbols by using a synthesis approach based on theory and empirical evidence about the cognitive effectiveness of visual symbols [16]. Some of these principles are related to a visual language as a whole, such as Complexity Management, Cognitive Integration and Graphic Economy. While others focus on individual visual symbol’s properties, such as Semiotic Clarity, Visual Expressiveness and Perceptual Discriminability.

To maximize the cognitive effectiveness of our DSVLs, we applied Moody’s PoN principles to design our visual symbols. Among these principles, we put more emphasis
on those that are subject to DSVL’s characteristics. When multiple entities are to be used, the Perceptual Discriminability principle will be our primary design consideration. This principle is assessed by the visual distance between symbols, measured by the number of visual variables on which they differ and the size of these differences. In contrast, there is no meaning to consider visual distance, if a DSVL contains only one entity. We would focus on the Semantic Transparency principle instead.

In addition to the cognitive effectiveness of visual symbols, there are also some other factors to be considered when designing DSVLs for this research such as reusability. To maximize the reusability, we should make models simple enough to be reused or easily assembled with others as a reusable component. This is the main reason why we have designed some single entity sub-DSVLs.

In the following sections, we introduce the designs of some TeeVML visual constructs. Interested readers can refer to our earlier publications [17, 18] for all the details.

### 4.2 Signature DSVL

Our Signature DSVL consists of three sub DSVLs: WSDL, Operation and Message. Table 2 describes the visual constructs for the five entities used in WSDL sub-DSVL. To provide sufficient visual distance for making them easily distinguishable, we used shape as the primary visual variable, supplemented with color and textual annotation.

### 4.3 Protocol DSVL

Protocol DSVL consists of three state entities for representing endpoint idle, home and working states and four transition relationships for managing endpoint state transitions. Table 3 provides our Protocol DSVL design details of the working state and relationships visual constructs. To distinguish between the relationship visual constructs, their visual variables include shapes at both ends, color, line type and textual annotation.

### 4.4 Behavior DSVL

Our Behavior DSVL has 9 visual constructs used for describing different types of tasks: (1) Service Node and Node for processing an operation or a task, (2) Arc and Entrance/Exit bars for directing dataflow, (3) Variable and Variable Array for holding intermediate results, (4) Conditional Operator and Loop for controlling process flows, and (5) Evaluator for performing arithmetic operations. To provide a general feeling of how Behavior DSVL is designed, we describe some of these constructs below.

**Service Node**

A service node specifies the process of an operation request and generates response. It imports the request and response parameters from the endpoint signature model. Figure 5a shows the visual construct of a service node. Its main design consideration is to display all request and response parameters for helping users to model the operation behavior. To manage behavior model view complexity, service nodes can be collapsed to hide parameters and reduce their symbol size (see Fig. 5b).
Entrance and Exit Bars
Entrance and exit bars (see Fig. 6) specify the input and output parameters and define where execution starts and ends within a service node or a node. The entrance bar has one “out” port underneath, and the exit bar has a normal “in” and an exceptional “in” ports on it. The parameters for both bars can be displayed or hidden by users, depending on whether users need to know these parameters.

Evaluator
Evaluator (see Fig. 7) is use to perform arithmetic operations. The evaluator visual construct has three lines for defining a formula. The first line specifies a result variable to be assigned after the evaluator’s execution. The second line lists all variables to be used by the evaluator, separated by commas. The last line is the arithmetic formula with parameters in as “P” array. The order of the “P” array elements follows the sequence of the parameters in the second line.

Table 2. WSDL sub-DSVL visual constructs.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Visual Symbol</th>
<th>Description</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td><img src="image" alt="Visual Symbol" /></td>
<td>Service is a set of system operations that are exposed to the Web-based protocols.</td>
<td>Name: A Service instance name.</td>
</tr>
<tr>
<td>Port</td>
<td><img src="image" alt="Visual Symbol" /></td>
<td>Port is an address or connection point to a Web Service. It is typically represented by a simple HTTP URL string.</td>
<td>Name: A Port instance name. Address: The network address at which the Service is offered.</td>
</tr>
<tr>
<td>Binding</td>
<td><img src="image" alt="Visual Symbol" /></td>
<td>Binding entity specifies interface, SOAP binding style and transport protocol.</td>
<td>Name: A Binding instance name. Type: To identify the kind of binding details contained in a Binding entity instance.</td>
</tr>
<tr>
<td>PortType</td>
<td><img src="image" alt="Visual Symbol" /></td>
<td>PortType entity defines a Web Service, operations that can be performed, and the messages that are used to perform the operation.</td>
<td>Name: A PortType instance name. Extends: A lists of PortType entities that this PortType derives from.</td>
</tr>
<tr>
<td>Operation</td>
<td><img src="image" alt="Visual Symbol" /></td>
<td>Operation is a Web Service action and the way a message is encoded. An operation is like a method or function call in a traditional programming language.</td>
<td>Name: Operation instance name. Pattern: A template for the exchange of one or more messages.</td>
</tr>
</tbody>
</table>
Table 3. Protocol DSVL visual constructs.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Visual Symbol</th>
<th>Description</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working State</td>
<td><img src="logon" alt="" /></td>
<td>It presents an endpoint state, which normally uses operation as its default name.</td>
<td>Name: State entity instance name; Synchronous Operation: Is the state operation in synchronous mode? Processing Time: Simulated operation processing time in seconds. Safe Operation: Is the state operation safe? Transmission Time: Simulated operation request transmission time in seconds.</td>
</tr>
<tr>
<td>Timeout Relationship</td>
<td>![15]</td>
<td>It links a “from” state to a “to” state for representing endpoint state change, if no valid operation request is received within a defined timeout period.</td>
<td>Time: The time in seconds for an automatic state transition.</td>
</tr>
<tr>
<td>Transition Relation-ship</td>
<td>![logon] to logon</td>
<td>It links a “from” state to a “to” state for representing a state transition.</td>
<td>OperationName: The operation triggers the state transition.</td>
</tr>
<tr>
<td>Constraint Transition Relationship</td>
<td>![logon to logon]</td>
<td>It links a “from” state to a “to” state for representing a state transition; The transition is subject to a constraint condition, defined by its dialog box (see Figure 4.7).</td>
<td>Trigger Operation: The operation triggers the state transition; Operation Name1 + Field Name1: To defines the first state transition condition; Condition Operator: It is used to compare the two conditions; Operation Name2 + Field Name2: To defines the second state transition condition.</td>
</tr>
</tbody>
</table>

![A service node definition](a) ![A collapsed service node](b)

Fig. 5. Behavior DSVL service node’s visual construct.
4.5 Code Generators and a Domain Framework

To make an endpoint executable to provide testing services to its SUT, endpoint models must be transformed to executable code by code generators. This code must then be integrated into a domain framework for providing infrastructure support. Ideally, the transformed code must be complete and in production quality, and should not require manual rewriting, inspection or additions.

**Code Generators**

A code generator accesses models, extracts information from them, and transforms the models into output in a specific form. This process is guided by the concepts, semantics and rules of the modeling language. Our TeeVML includes five code generators to transform endpoint signature, protocol and behavior models to corresponding codes:

- **WSDL (signature)** – To transform an endpoint signature model to WSDL 1.1 specification in XML format;
- **SQL Script (signature)** – To navigate through all operations and search for the parameters of the types of “int”, “float” or “date”. If such parameters are found, their “Minimum” and “Maximum” properties will be stored in an operation parameter table for verifying these parameters’ ranges;
- **Groovy Code Generator (protocol)** – To access and navigate the entire protocol model for extracting endpoint protocol information. The extracted data are stored in a state transition table for validating the operation request and determining state transition;
- **Java (protocol)** – To generate Java code to query protocol information from the state transition table for each operation request and validate the operation request by several “if-else” statements;
- **Java (behavior)** – To define the interdependences among nodes and primitive visual constructs and specify internal implementations within the primitive constructs.
A Domain Framework

A domain framework normally serves for four purposes [19]: (1) to remove duplications from generated code, (2) to provide an interface for code generators, (3) to integrate with existing code; and (4) to hide target environment and execution platform. In addition, our domain framework also plays another important role – to provide a network infrastructure for facilitating message exchange between endpoints and SUTs.

As it is not our research focus, we have not developed our own but used Axis2 SOAP engine [4] instead. Axis2 brings some benefits to our endpoint modeling approach including: (1) Axis2 facilitates Design by Contract (DbC) programming style [20], and the implementations on both the endpoint and SUT sides are bound to a service contract defined by signature WSDL file; (2) Axis2 allows users to modify its SOAP message header by adding some QoS attributes to simulate a variety of business scenarios; and (3) Axis2 is a popular open-source tool, many IT professionals familiar with it.

To implement DbC programming, Axis2 generates linkage codes for both service provider and service client from a signature definition WSDL file. The service provider linkage code takes the form of a service specific implementation skeleton, along with a message receiver class that implements the org.apache.axis2.engine.MessageReceiver interface. The service client linkage code is in the form of a stub class, which always extends the Axis2 org.apache.axis2.client.Stub class. Both the service provider skeleton class and client stub class are generated by the wsdl2java tool.

5 Case Study

We use the motivating example ERP system as a case study to demonstrate how endpoint can be modeled by using our TeeVML in interface layers. We also describe the steps to convert the endpoint models to executable forms and integrate them to the domain framework in a target environment.

5.1 Signature Modeling

Signature modeling starts from specifying endpoint level properties. Then, WSDL sub-DSVL is used to instantiate the five WSDL entities by providing their names and relevant information. They are linked together by using either a composition or an association relationship. All the entities 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 paymentrequest as an example to show how an operation can be modeled. The operation is instantiated by assigning the operation name as paymentrequest and pattern as “in-out”. Then, Operation sub-DSVL is used to specify the paymentrequest_request and paymentrequest_response messages in the operation. The request message label is “in”, and response message label is “out”.

Message sub-DSVL is used to define message elements. The request message contains only one element pono, 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 8 shows the ERP system endpoint signature model. It contains the top-level WSDL model (refer to Fig. 8a, we only show five operations for a better view representation), the middle-level paymentrequest operation model (Fig. 8b), and the bottom-level request and response message models (Fig. 8c).

![Signature Model](image)

![Operation Model](image)

![Message Model](image)

**Fig. 8.** The example endpoint signature model [6].

### 5.2 Protocol Modeling

Figure 9 shows the endpoint protocol model, where a purchase process progresses in clockwise direction. To demonstrate how to model the endpoint protocol layer, we select three typical protocol behaviors 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. To terminate the session, a logout transition relationship is used in the opposite direction. The session can also be terminated by a timeout event, which is specified 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 paymentrequest and the choice of the flows depends on whether the purchase item stock can meet the PO requirement. To specify this type of state transitions, the constraint transition relationship is used and it links the inventorycheck state to the supplierpo state. The constraint condition is specified using 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 greater 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 “to” state of the loop relationship will be executed repeatedly. The approval process of a supplier PO is an iteration, 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.

![Diagram](image)

**Fig. 9.** The example endpoint protocol model [6].

5.3 Behavior Modeling

An endpoint behavior model consists of unrelated service nodes for all provided operations, and we use one operation paymentrequest as example to explain how endpoint behavior is modeled. Figure 10a shows the paymentrequest service node, which consists of two sub nodes: poinformationretrieve to retrieve the PO information from tables and poamountcalculation to work out the total PO amount. These two nodes are placed between an entrance and exit bars.

We select the poinformationretrieve node to show how Behaviour DSVL primitive visual constructs are used to implement business logics. Figure 10b illustrates the data query operations and dataflows within the poinformationretrieve node. The node has one input “pono”, and generates four outputs: “quantity”, “unitprice”, “discount” and “FatalError”. The node contains three data query operations: (1) to retrieve PO “category”, “item”, “quantity” and “client” 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 “client”. If searching records are found, searching results will be placed on the normal output port (black circle) of data store operator.
Otherwise, an “FatalError” variable will be assigned by following the exceptional output port (yellow circle).

5.4 Testing Environment Generation

Our approach provides a very simple and easy way to generate operational endpoints from their models. There are three tasks including: (1) to create two Java project folders for hosting server side and client side codes; (2) to transform models to codes by code generators and copy them to the server project folder; and (3) to run our supporting toolset for packaging Tomcat service and providing testing service to SUTs. To automate endpoint generation process, we have created an Apache Ant build file.

Figure 11 illustrates a deployment view on how an endpoint provides integration testing service to a SUT. The left-hand side is the emulated endpoint hosted in a Tomcat application server, its protocol and behavior classes are integrated into the Axis2 Skeleton class for performing SUT operation requests validation. The grey areas at the bottom of both sides are the Axis2 Web Service engine for encoding and decoding SOAP messages exchanged between the endpoint and the SUT. A SUT is located on the right-hand side at the top, communicating with the Axis2 Stub class through an API class. The SUT invokes the endpoint service through accessing Tomcat Axis2 service URL using SOAP over HTTP application protocol. To capture and see the exchanged messages, we use TCPMon tool [21] to act as an intermediary between the SUT and endpoint. TCPMon accepts connection from the SUT on one port and forwards the incoming traffic to the endpoint running on another port.
6 Evaluation

To assess how well the issues related to the three research questions have been addressed by our approach, we have developed three evaluation criteria:

- **Testing functionality** (addressing RQ1) – the approach should be able to develop a wide variety of endpoints, which could be used to capture all the interface defects of a new or an existing system under test;
- **Development productivity** (addressing RQ2) – the approach should have high endpoint development productivity with less development effort and time;
- **Ease of use** (addressing RQ3) – the approach should be easy to learn and adapt by non-technical background users.

These criteria were first assessed by a technical comparison of our approach with the currently available endpoint emulation approaches. This comparison motivated our new DSM approach to address the shortcomings of the existing approaches. After our approach was ready to use, we also conducted a user survey to evaluate the extent to which our approach was accepted by software testing experts and developers.

6.1 Technical Comparison

Currently, there are two types of TEE approaches to develop integration testing environments: specification-based by manual coding (also called “manual coding”) and interactive trace data record-and-replay (also called “interactive tracing”). The manual coding approaches are used by IT professionals to develop simplified versions of applications with external behavior manually [22, 23]. They perform this using available knowledge of the underlying message syntax, interaction protocol and
behavior. The interactive tracing approaches create endpoint models from recorded request-response pairs between the endpoint system and an earlier version of a SUT automatically [24, 25]. Each endpoint’s simulated response is generated by finding a closely matched request in the recorded trace database.

To compare these two types of approaches with our new TeeVML, we use the three defined evaluation criteria and look into what and how some key techniques these approaches adopt to meet these criteria. Table 4 presents the comparison. From the development productivity and ease of use point of view, the interactive tracing approaches are the highest, as endpoints are created automatically. However, these approaches have two key shortcomings in terms of the 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. In contrast, the manual coding approaches and TeeVML need to develop endpoints by IT professionals. As TeeVML uses higher level abstraction models than code to express design intent, it achieves better endpoint development productivity and ease of use.

<table>
<thead>
<tr>
<th>Manual coding</th>
<th>Interactive tracing</th>
<th>TeeVML</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Testing functionality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The key motivation of these approaches is to provide performance testing by emulating large number of endpoints. To achieve this objective, these approaches adopt a light-weight architecture design and some testing features are deliberately neglected. Dynamic protocol behavior cannot be modeled, as state transition is triggered only by an operation. Unless great effort is made, behavior layer modeling will be limited</td>
<td>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</td>
<td>Endpoints modeled by TeeVML provide integration testing functionality from signature, protocol and behavior abstraction layers. The signature layer model supports all RPC communication styles; the protocol layer can model both static and dynamic protocol behaviors; and the behavior layer uses a hierarchical structure dataflow programming for modeling complicated logic implementations</td>
</tr>
<tr>
<td><strong>Development productivity</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The approaches adopt a modular architecture design, where an endpoint type dependent message engine module is separated from an endpoint type independent network and a system configuration</td>
<td>Endpoint is created by recording the interactive tracing data between the endpoint application and an earlier version of the SUT application. These approaches do not need any endpoint development</td>
<td>An endpoint is modeled by layers, and layer models are transformed to executable code. The key solution to productivity improvement is to maximize components reusability. We have adopted multi-level design for</td>
</tr>
</tbody>
</table>

(continued)
6.2 User Survey

User surveys incorporate a list of questions to extract specific data from a particular group of people. They provide a comprehensive mechanism for collecting information to describe, compare and explain knowledge, attitudes and behaviors of survey participants [26]. Survey results are used to improve products’ quality and functionality by guiding and correcting the design, development and refinement.

Experiment Setup

We conducted our user survey in two phases. In the first phase, we extracted testing experts’ opinions on what testing features they valued in endpoints and what functionality TeeVML should provide to develop endpoints. We introduced our TeeVML and endpoint testing functionality to the participants by using a PowerPoint presentation, then interviewed them and answered their queries. Sixteen testing experts were invited to participate in the survey, and most of them (94%) had more than one year solid testing experience and were knowledgeable about SIT. In the second phase, we assessed TeeVML’s usability by collecting software developers’ experience with the tool to work on an assigned task. We wanted them to compare TeeVML with a third-generation language they were familiar with, as the way the manual coding approaches do. Total of 19 software developers and IT research students took part in the survey. Most of them (95%) had IT background and (63%) were familiar with software modeling.

All the participants were asked to fill an online questionnaire, after finishing their user survey. The questionnaires include 5-point Likert Scale (5 to 1 representing strongly agree to strongly disagree), single-choice, multiple-choice, and open ended questions. For the 5-point Likert Scale questions, in favour responses encompass the answers of either 5 or 4 for a positive question, and 1 or 2 for a negative question. We counted the number of in favour responses to measure the degree of acceptance to a question statement. There were total 58 questions for both Phase One and Two, and we

<table>
<thead>
<tr>
<th>Manual coding</th>
<th>Interactive tracing</th>
<th>TeeVML</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules. However, as the message engine is coded manually, significant amount of development effort is needed for each new endpoint type</td>
<td>work, but some effort on trace data recording</td>
<td>Signature DSVL and node hierarchical structure for Behavior DSVL</td>
</tr>
</tbody>
</table>

*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. But, users must be trained to use the tool | Developers must have business domain knowledge, and some modeling skill is preferred |

Table 4. (continued)

292 J. Liu et al.
only select some of them for this paper results presentation. The full result reports can be accessed at: https://sites.google.com/site/teevmlapsec/.

Table 5. Selected questions and responses from Phase One survey report.

<table>
<thead>
<tr>
<th>No</th>
<th>Likert scale questions</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Statement Frequency</td>
<td>5 4 3 2 1</td>
</tr>
<tr>
<td>Q8</td>
<td>In your opinion, an emulated testing environment is useful for an application inter-connectivity and inter-operability test</td>
<td>8 6 0 1 1</td>
</tr>
<tr>
<td>Q17</td>
<td>It is useful for an emulated testing environment to provide signature testing functionality to its system under test</td>
<td>7 7 1 1 0</td>
</tr>
<tr>
<td>Q21</td>
<td>It is useful for an emulated testing environment to provide interactive protocol testing functionality to its system under test</td>
<td>12 4 0 0 0</td>
</tr>
<tr>
<td>Q25</td>
<td>It is useful for an emulated testing environment to provide interactive behavior testing functionality to its system under test</td>
<td>6 8 1 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No</th>
<th>Multiple choice questions</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q13</td>
<td>Question statement and choices</td>
<td>14 10 5 15</td>
</tr>
<tr>
<td>Q14</td>
<td>Extra development effort on testing endpoints</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Learning a new technology</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Inadequate testing functionality</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Emulation accuracy</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Result reliability</td>
<td>12</td>
</tr>
</tbody>
</table>

Survey Results Analysis – Phase One

We select a few typical questions in Table 5 to analyze them from two different angles: One is about participants’ acceptance of an endpoint as a whole and by each interface layer from functionality point of view. Another is to find out the possible reasons why participants would consider using (or not using) our endpoints in their future projects.

Q8 reflects the overall usefulness of endpoints for conducting SIT. The responses to this question are quite positive with 14 out of 16 (87.5%) participants in favour. This is a good indication of the participants’ acceptance of endpoints modeled by TeeVML.
To further investigate each interface layer, Q17, Q21 and Q25 are used to get participants’ opinion on the usefulness of modeling signature, protocol, behavior layers, respectively. We can see that the protocol layer (Q21) received in favour responses from all participants. We believe one of the main reasons why all participants wanted to have protocol testing is that most applications do not have a well-documented protocol specification. Therefore, protocol related defects can only be found by SIT. On the other hand, the signature layer (Q17) had slightly less in favour response rate compared to the protocol layer. The signature correctness is a must for a client to access operations provided by a server. However, a few participants might have thought that endpoint signature could be easily coded and verified against product interface specification, hence actual testing would be unnecessary.

Q13 is a multiple-choice question, and lists four reasons why users want to use endpoints. Responses to Q13 indicate that the top reason for using endpoints was early detection of interface errors. In current practice, SIT 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 might have preferred to conduct at least part of SIT earlier. Q14 indicates that most participants’ concerns were on the reliability of endpoint testing results. The main reason could be that software developers are used to using real applications for their SIT. However, an endpoint is actually a simplified version of its real application. Often, some implementation aspects of the application are neglected and treated as useless for SIT. This might have some impacts on SUT testing results. Our survey results indicate the importance of conducting endpoint functionality design before modeling it.

Survey Results Analysis – Phase Two

Giving that the participants have used our tool to model an endpoint operation, we want them to provide their opinions on whether the tool is ease of use and how much endpoint development productivity can be improved. The former uses the 10 questions from System Usability Scale (SUS), and the latter is based on two questions of the actual time spending on the task and a subjective comparison with a third-generation language.

SUS is a simple, 10 5-point Likert Scale questions (see Table 6) to give a global view of the subjective assessment of a product’s usability [27]. SUS yields a single number by adding up the score contributions from each question and multiplying by 2.5 to represent a composite measurement of the overall usability of the system being studied. By a statistic study over a large number of products, the overall SUS mean score was 68 [28]. By this survey, TeeVML overall SUS score is calculated as 78.3 out of 100 points, which is equal to 83% from a percentile ranks for raw SUS scores table [28]. From another angle, our SUS score falls between “Good” and “Excellence” in the adjective ranges of the “Acceptability” scoring system proposed by Bangor et al. from a study on numerous products [29].
Table 7 represents survey results for TeeVML’s productivity to model endpoints. For Q9, 79% participants could finish their task within 30 min, which is a typical endpoint operation modeling. Based on this result, we can generalize that it is possible to model a relatively complex endpoint with more than 10 operations within a day through using our TeeVML. From Q22 we can see that more than half of respondents (57.8%) agreed that using TeeVML would reduce “50%–80%” or “80%+” of the time duration they use for endpoint development. No participant voted “Almost the same”. As the results from these two questions, we can conclude that most participants agree that our TeeVML could increase endpoint development productivity significantly comparing with traditional manual coding approaches.

Table 6. System Usability Scale questions [27].

<table>
<thead>
<tr>
<th>No</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q12</td>
<td>You would like to use the tool in your future project</td>
</tr>
<tr>
<td>Q13</td>
<td>You found the tool unnecessarily complex</td>
</tr>
<tr>
<td>Q14</td>
<td>You found the tool was easy to use</td>
</tr>
<tr>
<td>Q15</td>
<td>You would need support to be able to use the tool</td>
</tr>
<tr>
<td>Q16</td>
<td>You found the various features of the tool were well integrated</td>
</tr>
<tr>
<td>Q17</td>
<td>You found there was too much inconsistency in the tool</td>
</tr>
<tr>
<td>Q18</td>
<td>You would image that most people would learn to use the tool very quickly</td>
</tr>
<tr>
<td>Q19</td>
<td>You found the tool very cumbersome to use</td>
</tr>
<tr>
<td>Q20</td>
<td>You felt very confident using the tool</td>
</tr>
<tr>
<td>Q21</td>
<td>You needed to learn a lot of things before you could get going with the tool</td>
</tr>
</tbody>
</table>

Table 7. Phase Two survey results for development productivity.

<table>
<thead>
<tr>
<th>No</th>
<th>Statement</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9</td>
<td>How long did it take you to complete the task?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10–15 min</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>16–20 min</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>21–25 min</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>26–30 min</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>30+ min</td>
<td>4</td>
</tr>
<tr>
<td>Q22</td>
<td>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?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Almost the same</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10–25%</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>26–50%</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>51–80%</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>81%+</td>
<td>2</td>
</tr>
</tbody>
</table>
7 Related Work

Over the years, many approaches have been proposed to develop testing environments. Hardware virtualization tools, such as VMWare [30] and VirtualBox [31], provide management and control over virtual testing servers and they are capable in hosting many endpoint systems in one or a few machines. However, some applications need to be run in shared memory space and they cannot be virtualized. Method stubs [32] and mock objects [33] are programming approaches to mimic testing applications. Their key advantages are low cost and quick deployment. But these approaches abstract away from communication complexities which may significantly impact on the results encountered in the real deployment.

To address both static and dynamic issues related to software components interactions, Han first proposed a rich interface definition framework with logically separated layers [34]: signature, constraints, configurations, and a quality aspect across the three 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 [35]. Our approach on the other hand, focuses on how requests are to be processed in a layered manner and interface defects are captured by endpoints.

For protocol modeling, some researchers used a FSM [36, 37] or a formal protocol specification [38, 39] to validate operations sequence for different endpoint states. However, Wehrheim et al. argued that the use of operation name alone might not be sufficient to trigger a state transition for a realistic endpoint [40]. To deal with the so-called incomplete protocol specification, an EFSM-based protocol modelling calculus were proposed for specifying operation 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 the lack of concrete implementation solutions to capture endpoint runtime aspects. Another one is the textual form they used for writing state transition rules, and this will make protocol modeling difficult.

Software interactive behaviors can be modeled either externally or internally. Software behavioral interface specification [41] and programming from specification [42] are the external approaches, they model interactive behaviors by defining pre/post conditions to bind servers and their clients. As internal approaches, Business Process Model and Notation (BPMN) [43] and DataFlow Programming (DFP) [13] 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. To handle 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 [44]. Specifically related to our work, UML provides: (1) a testing profile to provide a generic extension mechanism for the automation of test generation processes [45], (2) state charts to simulate finite-state
automaton [46], and (3) activity diagrams to graphically represent workflows of stepwise activities and actions [47]. Two main problems with using UML to define new modeling languages [48] 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.

8 Conclusion and Future Work

Aiming to achieve high development productivity and ease of use for domain experts, we have proposed a DSM approach for testing environment emulation. Our approach is based on a new software interfaces description framework to abstract an endpoint into three message processing layers, and a suite of DSVLs have been developed for modeling these layers. Using this layered modeling, our approach supports partial endpoint development, where an endpoint may have only one or two of these layers to meet SUT testing requirements. For a SUT without dynamic interactive aspects, the endpoint behavior layer could be ignored.

A fully functional endpoint should also be able to test SUT’s QoS aspects. These QoS aspects may include security, performance, robustness, etc. For example, applications may put extra security 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. Object-oriented programming has higher expressive power than imperative and procedural programming by supporting inheritance, polymorphism, encapsulation, etc. Making our Behavior DSVL object-oriented can simplify behavior modeling, increase development productivity and output accuracy, and have a better diagrammatic view of behavior model. Furthermore, to reduce modeling overhead in effort and time, some special purpose utility nodes should be provided with Behavior DSVL for common modeling features. These and others could be part of our future work.

Acknowledgements. Support for this work from an Australian Postgraduate Award for the first author and partial support for this work from ARC Discovery Project DP140102185 is greatly acknowledged.

References


34. Han, J.: Rich Interface Specification for Software Components. Peninsula School of Computing and Information Technology Monash University, Mahon Road Frankston, Australia (2000)


