FORMAL METHODS FOR THE SPECIFICATION AND TESTING OF DATA-CENTRIC WEB SERVICES: A CASE STUDY

Iman Saleh*, Gregory Kulczycki†, M. Brian Blake*, Yi Wei‡
*University of Miami, †Virginia Tech, ‡Microsoft
iman@miami.edu, gregwk@vt.edu, m.brian.blake@miami.edu, dawe@microsoft.com

Abstract
Web services allow organizations to capture their human and software-based capabilities as modular software components that are called remotely over a network. In such service-oriented settings, it is important to establish an agreement that sets the obligations of the service provider and the expectations of the service consumer. Since traditional approaches such as Service Level Agreements (SLAs) are loosely defined with respect to data integrity aspects, in previous work, we proposed a formal model for specifying data-centric Web services. The goal is to formally and unambiguously specify the service behavior in terms of its underlying data model and data interactions. In this paper, we present a case study where we use our model to specify and verify data-centric Web services. We demonstrate our proposed methodology using three state-of-the-art specification languages: JML, Dafny and RESOLVE. We also explore the use of our specification to automatically generate test cases for data-centric services. Our goal is to study the feasibility of our proposed model and also to pinpoint the challenges and limitations of current specification and verification tools.

Keywords: Formal methods; Web Services; Verification

1. INTRODUCTION
Unlike software components operating within an enterprise, the Web services model establishes a loosely coupled relationship between a service producer and a service consumer. Service consumers have little control over services that they employ within their applications. A service is hosted on its provider’s server and is invoked remotely by a consumer over the Web. In such settings, it is important to establish a contract between the service provider and the service consumer. The contract establishes a set of obligations and expectations.

Applying design-by-contract principles in the context of Web service addresses many practical challenges in current Web service development, and facilitates future development as formal verification tools mature. From the perspective of a service producer, a Web service must be designed and implemented to be broadly applicable. Quoting (Kaye 2003), “[The concept of broad applicability] suggests that the eventual uses of any given service can’t be predicted at the time the service is created... In other words: Design and implement your Web service interfaces to handle anything that might be thrown at them”.

Accordingly, the service implementation must cover all possible invocation scenarios. One way to achieve this is by adding defensive checks to the service implementation. While these checks may ensure that the service is invoked with the correct input, they can affect the code efficiency in a negative way. As described in (Leavens and Cheon 2006), consider a binary search method that requires its array argument to be sorted. Checking that an array is sorted requires time linear with respect to the length of the array, but the binary search routine is supposed to execute in a logarithmic time. Design-by-contract techniques help avoid these inefficient checks by explicitly adding pre-conditions on the service inputs. It is then the consumer’s responsibility to satisfy these pre-conditions in order to guarantee that the service behaves as expected. Moreover, if these pre-conditions are formally defined, then the consumer can use automatic reasoning tools to check his/her service composition and to guarantee that it satisfies the pre-conditions. Furthermore, these checks are accomplished statically. That is, they are made without having to run the implementation that is based on the composition of services.

While the inclusion of preconditions is a best practice that is supported using current development technologies, the principle of formally specified contracts offer a much greater benefit in the future as tools for verification continue to evolve. Formally specified preconditions, along with postconditions and invariants can potentially be used to completely describe the behavior of a service. At a minimum, these descriptions provide web service consumers with a precise and unambiguous way to understand the service. But as tools mature, they also provide service producers with a way to statically verify the behavior of their services, significantly mitigating the need to perform run-time testing of their services. A few state-of-the-art tools are beginning to emerge that partially support these ambitious verification goals.

In previous work (Saleh et al. 2009a)(Saleh et al. 2009b)(Saleh et al. 2013), we proposed a data modeling and contracting framework for data-centric Web services. Our framework formally specifies a service with respect to its interaction with the data. We also provided pseudo-code of
the model implementation. In this paper, we show an implementation of our model and a case study using three of the state-of-the-art specification languages, namely JML\textsuperscript{1}, Dafny (Leino 2010) and RESOLVE (Ogden et al. 1994). We choose these languages as they represent recent effort for developing specification languages and there’s an ongoing work to improve their verifiers. They also represent interesting examples of different specification approaches that we compare and contrast; JML is an extension of a programming language, Java. Dafny combines both procedural and functional programming and, RESOLVE has a sound mathematical foundation and enables the inspection of verification conditions, as we will be explaining later in detail. Our goal is to show the feasibility of our model implementation and to study the current challenges and limitations of formal languages and verification tools. Our experiment described here show that, despite the limitations of the tools, we could still reason about a service-based implementation and obtain some proofs of correctness. We also show how we can leverage the specifications in order to automatically generate test cases for data-centric services.

2. RELATED WORK

Semantic approaches have gained a lot of attention in Web Services community as a way to specify service capabilities. The W3C OWL-S\textsuperscript{2} standard for semantically describing Web services is an example of such an approach. Semantic techniques are based on description logic, which supports the definition of concepts, their properties and relationships. The reasoning tasks supported by description logic include instance checking, relation checking and subsumption (Baader et al. 2007). This makes techniques based on description logic suitable for solving problems related to the automatic discovery and composition of services, since these problems require matching between a semantically-annotated user query and a semantically-specified Web service. In contrast, our work is based on formal methods, which support verification of correctness of a computer program. For Web services, formal methods are suitable for solving problems related to compositional correctness and verifying that a service complies with its advertised specification. From a software engineering perspective, the semantic techniques and formal methods techniques are complimentary, as they address software validation and verification problems, respectively. While a semantic-based approach can validate that a service or a composition of services match a user query, a formal method approach can verify that the service or composition of services is implemented correctly with respect to that user query.

A related work in (Vaculin et al. 2008) identifies the need to make databases visible to service discovery and composition algorithms. The authors propose an extension to OWL-S standards to support the specification of data-providing services. A data-providing service is a read-only service that provides access to one or more possibly distributed and heterogeneous databases. The service’s main functionality is to retrieve data from a data store based on an input query. Their approach facilitates the automatic discovery of services by applying ontological reasoning. They suggest describing data sources as RDF views over a shared mediated schema. Specifically, the local schema of each data source is mapped to concepts of a shared OWL ontology, and its terms are used to define RDF views describing the data sources in the mediated schema (Vaculin et al. 2008). Their approach is useful in matching a service with a user query based on ontology-based reasoning. As stated before, they use semantic techniques and hence their approach do not address correctness issues or reasoning about the service side-effects. Another related work presented in (Deutsch and Vianu 2008) handles the specification of interactive Web applications and focuses on specifying Web pages and user actions. The proposed data model incorporates temporal constructs to specify browsing paths between pages and application behavior in response to user actions such as clicking a button or browsing through hyperlinks. This approach is useful in verifying properties like page reachability and the occurrence of some events. This approach is working from a process perceptive and an input-boundedness restriction is assumed to guarantee that the verification operation can be done in polynomial time. In our work, we are specifying and reasoning about so-called stateless Web services by modeling the state of the overall system, which necessarily includes the data store. We model the underlying data store as a sophisticated global variable, so that we can reason about how it is modified between a service request and response. We do not attempt to specify all states of a data store, as we focus on correctness and verification of data-related side effects after a service call.

The Tisa language recently proposed in (Rajan et al. 2009) employs a similar methodology by applying formal methods techniques to specify temporal service policies. Policies include non-functional regulations and privacy rules that should be maintained by a service implementation. The language supports reasoning about correctness of a service composition with respect to these policies. In our work, our goal is to reason about correctness with respect to the functional data behavior of a composition of service. In general, our approach enhances the advertised interface of a service by including the specification of its data behavior.

3. A CONTRACTING FRAMEWORK FOR DATA-CENTRIC WEB SERVICES

We propose a framework to support modeling and contracting of data-centric Web services. Our framework enables contract-based composition of services as shown in Fig.1. Based on our framework, the service developer abstracts the data model and annotates the service with a contract that specifies the data requirements and the service’s side effects. The contract is machine-readable and hence automatic reasoning techniques can be used at this stage to ensure the correctness of the service implementation with respect to the contract. Assisted with tools, the developer encodes and publishes this contract along with the service WSDL3 file. At service invocation, a consumer consults the contract to understand the service behavior, ensure correct service invocation, and interpret the service result and effect. Again, automatic reasoning can be used at this stage to ensure the correctness of a service-based composition. Reasoning on the consumer side guarantees that correct inputs are formulated and hence avoiding service requests that will return errors or cause unintentional side effects.

Figure 1. Data Modeling and Contracting Framework for Data-Centric Web Services

By applying design-by-contract techniques, our framework enables the service consumer to ignore the data and service implementation details. It also allows the service providers flexibility in creating or modifying implementations for their services. This separation between specification and implementation promotes software modularity and consequently promotes service reuse.

4. A FORMAL DATA MODEL

We model a data source as a set of entities, where each entity is a set of records. In addition to a unique record identifier (key), a record can have zero or more attributes. This model leverages current data models that we surveyed including the relational and object-oriented modeling of databases, and some earlier efforts for formally specifying databases (Souto 1994)(Fisher 2007).

class GenericDataModel
  attribute entity1: Set(GenericRecord1)
  attribute entity2: Set(GenericRecord2)
  ...
  attribute entityn: Set(GenericRecordn)
  operation GenericRecordi findRecordByKey(key: GenericKeyi) requires (GenericKeyi is the key for GenericRecordi) ensures (result.key = key and result in this.entityi) or result = NIL
  operation Set(GenericRecordi) findRecordByCriteria(values1: Set(Ti1), values2: Set(Ti2), ..., valuesn: Set(Tin)) requires (Tij is the type of the jth attribute of GenericRecordi) ensures  rec in result, rec.attrj in valuesj and result in this.entityi
  operation GenericDataModel createRecord(gr:GenericRecordi) ensures result.entityi = this.entityi U gr
  operation GenericDataModel deleteRecord(key: GenericKeyi) ensures result.entityi = this.entityi - this.findRecordByKey(key)
  operation GenericDataModel updateRecord(gr:GenericRecordi) requires this.findRecordByKey(gr.key) ≠ NIL ensures result.entityi = this.deleteRecord(key).createRecord(gr)
end GenericDataModel

Listing 1. The Data Model Template

We adapt the CRUD (Create-Read-Update-Delete) (Kilov 1990) model to include functional descriptions of the basic data operations. The reader is referred to (Saleh et al. 2009a) and (Saleh et al. 2009b) for a detailed description of our model. The model is used to formally specify a service’s interactions with its underlying source(s). Listing 1 presents a data model class template that we provide to help developers in specifying a data source.

Throughout our work, we use Hoare-style specification (Hoare 1969) to define an operation’s pre/postconditions. A precondition is specified using a ‘requires’ clause and a postcondition is specified using an ‘ensures’ clause. The result keyword is used to denote an operation’s output. The ‘#’ prefix is used to denote the value of a variable before the invocation of the operation. We use these notations in Listing 1 to specify the data operations. These operations are defined as pure mathematical functions that have no side-effect.

http://hipore.com/ijsc
5. The PayPal Express Checkout Case Study

5.1 General Description

The PayPal online payment solutions expose a set of services to facilitate the integration of electronic payment within e-commerce websites. The integrations steps are described within a so-called Express Checkout flow depicted in Figure 2.

![Figure 2. PayPal Express Checkout Workflow](image)

As shown in the figure, in the flow, the buyer:
1. Chooses Express Checkout by clicking the Check out with PayPal button
2. Logs into PayPal to authenticate his or her identity
3. Reviews the transaction on PayPal
4. Confirms the order and pays from your site
5. Receives an order confirmation

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetExpressCheckout</td>
<td>Initiates an Express Checkout transaction with an estimate of the total payment. The service returns a token that uniquely identify the new transaction.</td>
</tr>
<tr>
<td>GetExpressCheckoutDetails</td>
<td>Obtains information about the buyer given a transaction token.</td>
</tr>
<tr>
<td>DoExpressCheckoutPayment</td>
<td>Completes an Express Checkout transaction given a token and the actual total payment amount.</td>
</tr>
</tbody>
</table>

Table 1. PayPal Express Checkout Web Services

Behind the scene, this scenario is implemented by invoking three of PayPal Web services. Table 1 contains the description of these three services. We are using a simplified version of the flow as a composition case study that we formally model and specify using the proposed framework.

5.2 The Data Model

Given the PayPal description and documentation of these services, we infer the data model of the underlying database for the Express Checkout services as shown in Listing 2. The model is based on the template defined earlier in Listing 1.

![Listing 2. The PayPal Data Model](image)

As shown in Listing 2, we define the underlying Express Checkout record model, represented by the TransRecord class, as consisting of a token, a payment transaction amount transAmount and the corresponding payer information captured by the payerInfo attribute. A payment has a status represented by the paymentStatus attribute. The token attribute is a timestamped token that is used by the three Express Checkout services to relate different services calls to one payment transaction. It is unique and hence we choose to use it as the transaction record key. This example shows how our model can reuse Web service data types defined in WSDL files; in the above listing, for example, we are reusing PayerInfoType which is a complex data type used by different PayPal services to hold the payer information such as name, shipping address, email and others. This practice is very useful in minimizing the effort of modeling a service and ensuring that the model complies with the original service design.

---

5.3 Individual Service Contracts

```
PayPalDM ppdm
TransRecord rec

String setExpressCheckout(Float sPaymentAmount)
modifies ppdm, rec
ensures rec.token ≠ Nil
and ppdm.findRecordByKey(rec.token) = Nil
ensures rec.payerInfo ≠ Nil
and rec.transAmount = sPaymentAmount and
result = sRec.token
ensures rec.paymentStatus =InProgress
ensures ppdm = ppdm.createRecord(rec)

PayerInfoType getExpressCheckoutDetails(String gToken)
modifies rec
requires ppdm.findRecordByKey(gToken).payerInfo ≠ Nil
ensures rec = ppdm.findRecordByKey(#gToken) and
result = rec.payerInfo

boolean doExpressCheckout(String dToken, Float dPaymentAmount)
modifies ppdm, rec
requires ppdm.findRecordByKey(dToken).payerInfo ≠ Nil
ensures rec = ppdm.findRecordByKey(#dToken)
ensures rec.transAmount = #dPaymentAmount
ensures (result = TRUE and rec.paymentStatus = Processed)
or (result = FALSE and rec.paymentStatus = Denied)
ensures ppdm = ppdm.updateRecord(rec)
```

Listing 3. The Individual Data Contracts of the PayPal Services

The data model is used to annotate services with formal specifications represented as data contracts. Listing 3 shows how we use the PayPal model to expose a data contract for each individual service in the PayPal flow. Our specification is intended to be complete; i.e., any programming or state variable that is not explicitly specified in the service contract is assumed to be unchanged after the service execution.

In our specification of the PayPal services, we begin by defining a model variable `ppdm` of type `PayPalDM` representing the underlying data store. A model variable is a specification-only variable (Cheon et al. 2005) that is used in conjunction with programming variables to model the state of the system. We include the model variable in each service specification to reflect the fact that all services are reading and updating the same data store and hence capturing dependency and compositional effect of services on that data store. Consequently, the state in our case is represented by service inputs and outputs in addition to the data store model variable. To simplify the specification, we also define a model variable `rec` of type `TransRecord` that is used to specify a transaction record, whenever needed.

As a demonstrative example, consider the contract of the `getExpressCheckoutDetails` service in Listing 3. The contract ‘requires’ clause specifies that, before the service invocation, the underlying data model should have a transaction record, identified by the service input `gToken`, and this record should have a non-null payer information attribute. In other words, the service is called when the payer information has been captured and saved in the data model. The contract’s ensures clause specifies that the service returns the payer information related to that transaction record. The service does not change the data model since the contract does not explicitly define a ‘modifies’ clause.

It should be noted here that our example uses services from the same service provider. However, our proposed approach can be used to specify a composition of heterogeneous services that interact with different sources of data. In that case, we simply need to define a data model variable for each data source used in the composition.

5.4 Specification of Service Composition

```
PayerInfoType ExpressCheckoutFlow(float paymentAmount)
modifies ppdm, rec, token
ensures rec.transAmount = #paymentAmount and ppdm = #ppdm.createRecord(rec)
and (result = rec.payerInfo and result ≠ Nil
and rec.paymentStatus = Processed)
or (result = Nil and rec.paymentStatus = Denied)
Begin
result := Nil;
token := setExpressCheckout(paymentAmount);
PayerInfoType payerInfo := getExpressCheckoutDetails(token);
boolean responseValue := doExpressCheckout(token, payerInfo, paymentAmount);
if(responseValue)
result := payerInfo;
End
```

Listing 4. The pseudocode and the formal specification of the composition of services for the PayPal Express Checkout flow

We use formal specification to annotate the composition of services with a global data contract. The contract describes the intended behavior, from an integrator’s point of view, for the flow of services based on the individual data contracts of each of the participating services. The flow implementation and the global contract are shown in Listing 4. Listing 4 represents this flow in pseudo-code, but we believe this representation is foundational to more specific flow languages such as BPEL5 or OWL-S. We assume a global variable token of type string. The token is the

timed stamped value, described before, that relates different service calls to the same transactions. The contract’s ‘ensures’ clauses specify that the checkout flow creates a new transaction record with payment amount equal to the input payment amount. Also, the flow result is equal to the payer information associated with the newly created record in case the transaction is processed. Otherwise, the result is Nil and the payment transaction is marked as denied.

6. IMPLEMENTATION DETAILS

6.1 REQUIREMENTS

In order to implement the proposed model using a specification language, the language design must support the following constructs:

- Specification-only variables (a.k.a ghost variables): The data model in our framework is defined as a specification-only variable that is used to represent the underlying database and specifying the service-to-data interactions.

- Side-effect free methods: These are used to define the basic data operations supported by the data model. These operations are used in the specification of a service interface. Operations used in the specification must not change the state of the program.

- Specification of interface-only methods: Programmers depend on the APIs signatures in order to call services within their code. A specification language used to specify Web services must hence support a mechanism to specify interface-only methods. A verifier in this case can be used to ensure the consistency of the specification and not to prove its correctness.

In the following sections, we discuss parts of the implementation of the PayPal Express Checkout composition using the Dafny, JML and RESOLVE languages. The complete implementations can be found in Saleh (2012). We have also made an attempt to implement the modules using Spec#, however, Spec# does not provide constructs for user-defined theory types and hence we could not define our data model.

6.2 IMPLEMENTATION MODULES

The specification and verification of the PayPal Express Checkout flow entails the implementation the three modules shown below in Figure 3:

- Module (1): The PayPal data model.
- Module (2): The data contract for each of the three PayPal services.
- Module (3): The implementation and the data contract of the PayPal Express Checkout that composes the three PayPal services.

In the following sections, we discuss parts of the implementation of these three modules using the Dafny, JML and RESOLVE languages.

6.3 DAFNY

Dafny (Leino 2010) is a class-based specification language. A Dafny class can declare variables, methods, and functions. The language supports specification-only variables through ghost variables. It also supports user-defined mathematical functions that can be used in writing specifications. The language has a verifier that translates it to the Boogie intermediate verification language (Barnett et al. 2006). A Boogie tool is then used to generate first-order verification conditions that are passed to the Z3 theorem prover (De Moura and Bjørner 2008). The types supported by Dafny are booleans, mathematical integers, references to instances of user-defined generic classes, sets, sequences, and user-defined algebraic datatypes. Specifications in Dafny include standard pre- and postconditions, framing constructs, and termination metrics.

Listing 5 is an excerpt of the model implementation in Dafny. The implementation includes a definition of the TransRecord class. Due to the abstraction level of the Dafny language, we have simplified the model in terms of the data types used; sequences of integers for example are used to represent strings. Sequences are also used to represent the collection of record representing the transEntity attribute in the model class. The transEntity attribute is defined as a ghost variable as it is declared for specification-only purposes. For the same reason, the methods supported by the model are defined using Dafny’s mathematical functions. In Dafny, mathematical functions are declarative, side-effect free functions that can be used to write specifications. The domain of a function is defined by a requires clause. The reads clause gives a frame for the function, saying which objects the function may depend on. The decreases clause gives a termination and the function’s body defines the value of the function (Leino 2010). We
hence implemented a body for each function in the data model to define its value.

```java
1  class TransRecord {
2      var token: seq<int>; // used to represent a string
3      var transAmount: int;
4      var payerInfo: seq<int>;
5      var paymentStatus: seq<int>;
6  }

7  class PayPalDM{
8      // A 'sequence' is used to define collection of records
9      ghost var transEntity: seq<TransRecord>;

10     function isValid(): bool
11        reads *;
12        { 
13            (0 <= |transEntity|) &&
14                (forall j :: j >= 0 && j < |transEntity| =>
15                    transEntity[j] != null)
16        }

17     function findRecordByKey(key: seq<int>, i: int): TransRecord
18        requires isValid();
19        requires i >= 0 && i < |transEntity|;
20        decreases |transEntity| - i; //specifies that the recursion terminates
21        { 
22            if i >= |transEntity| then null
23                else if transEntity[i].token == key then transEntity[i]
24                else findRecordByKey(key, i+1)
25        }

26  } //end of class PayPalDM

Listing 5. An excerpt of the data model implementation in Dafny

Table 2. Verification Results using the Boogie Verifier

<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Verification Time(sec)</th>
<th>Verification Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class PayPalAPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isValid</td>
<td>0.024</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>expressCheckoutFlow</td>
<td>0.172</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>setExpressCheckout</td>
<td>0.081</td>
<td>Postconditions unverifiable</td>
</tr>
<tr>
<td>getExpressCheckoutDetails</td>
<td>0.057</td>
<td>Postconditions unverifiable</td>
</tr>
<tr>
<td>doExpressCheckout</td>
<td>0.089</td>
<td>Postconditions unverifiable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class PayPalDM</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>isValid</td>
<td>0.012</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>findRecordByKey</td>
<td>0.022</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>findRecordByCriteria</td>
<td>0.030</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>findRecordIndex</td>
<td>0.020</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>deleteRecord</td>
<td>0.035</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>createRecord</td>
<td>0.012</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>updateRecord</td>
<td>0.022</td>
<td>Passed successfully</td>
</tr>
</tbody>
</table>

Program Interpretation | 1.873
Total                  | 2.449

Figure 4. The distribution of the verification time using the Boogie verifier

Dafny does not support the definition of object invariants. Instead, validity functions are used, e.g. the isValid() function in the model implementation specifies the conditions for a valid object. A method that reserves the object validity must reference the validity function in its pre and post conditions. In our example, the validity of a data model implies that the collection of records has a zero or more records and that each record in the collection is not a null object.

Table 2 summarizes the verification results obtained by applying the Boogie verifier to the Dafny code. As can be seen in the table, the verifier fails to prove the postconditions of the three PayPal services since their implementations are not provided. This can be however safely ignored since modular verification can still be applied to the service composition. Figure 4 depicts the distribution of the time among the different verification tasks.

6.4 JML

The Java Modeling Language (JML) is a specification language that is used to specify Java modules. JML annotations are appended to Java code as comments proceeded by the at-sign (@). JML uses a ‘requires’ clause to specify a method’s pre-conditions and an ‘ensures’ clause to specify the post-conditions. The ‘result’ variable denotes the output of a method. The ‘old’ prefix denotes the value of a variable before the method invocation. In JML, side-effect free method labeled as pure can be used within a specification.

Listing 6 is the data model implementation in JML (we have omitted the definition of the TransRecord class to avoid repetition). An array is used to represent the collection of record representing the transEntity attribute in the model class. Similar to Dafny, the transEntity attribute is defined as a ghost variable as it is declared for specification-only purposes. The methods supported by the model are defined as abstract pure methods. In JML, side-effect free method labeled as pure can be used within a specification.

```java
1  abstract class PayPalDM {
```
We use a Concept in RESOLVE to specify the flow composition, and finally a Realization is used to provide the corresponding implementation. The RESOLVE Concept definition is shown in Listing 7. The complete code can be found in (Saleh 2012).

As shown in the code, RESOLVE defines a set of parameter modes. The ones we used include:

- **preserves** – the value of the incoming value will be preserved.
- **replaces** – value will be replaced by some other variable.
- **updates** – the value will be changed in an unspecified way.

To verify that the Express Checkout composition satisfies its specification, we first use the RESOLVE compiler to generate Verification Conditions (VCs). The VCs are a series of logical implications such that proving these implications is necessary and sufficient to demonstrate that the implementation is correct (Smith et al. 2009). The RESOLVE compiler generates the VCs in a user-friendly format that facilitates human inspection. They can also be verified. The verification results are shown in Table 3.

### 6.5 RESOLVE

The authors of (Sitaraman et al. 2011) present the RESOLVE language and its verifying compiler. RESOLVE is used to write specified object-oriented code and provides tools for both generating the verification conditions and proving simple ones. The compiler has been recently implemented as a Web tool (Cook et al. 2011).

We give here a brief description of the language, its structure and tools. The Resolve Tutorial can be consulted for more detail. The language has its built-in specifications that are written using universal mathematical notations. The RESOLVE compiler translates the code and the specification into Java code that can be compiled using Java compiler. The language is based on mathematical theories of programming data types such as integers, strings and booleans. These theories are used in writing the formal specifications and verifying them. In addition to the theories, RESOLVE supports different types of code units. A Concept in RESOLVE defines the mathematical model of a data structure. For example, a stack in RESOLVE is mathematically modeled as a sequence of strings. The stack Concept specifies operations such as pop, push and depth. An Enhancement is used to add custom functionalities. For example, an Enhancement can be used to add a stack reverse operation to the stack Concept. Both Concept and Enhancement units do not provide an implementation; they only provide the specifications. A Realization unit on the other hand provides the implementation of a Concept or an Enhancement. This organization of RESOLVE units enables decoupling the implementation from the specification.

We use a Concept to implement the PayPal data model, an Enhancement to specify the flow composition, and finally a Realization is used to provide the corresponding implementation. The RESOLVE Concept definition is shown in Listing 7. The complete code can be found in (Saleh 2012).

### Table 3. Verification Results using the JML Verifier

<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Verification Time (sec)</th>
<th>Verification Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class PayPalDM</td>
<td>0.016</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>Class TransRecord</td>
<td>0.063</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>ExpressCheckoutFlow</td>
<td>1.327</td>
<td>Passed successfully</td>
</tr>
<tr>
<td>Program Interpretation</td>
<td>0.812</td>
<td></td>
</tr>
</tbody>
</table>

**Total** 2.218

Unlike Dafny, the model can be implemented in JML using abstract methods that require no implementation. An advantage of the JML language is its tight connection with the Java language, which makes it potentially easier for programmers to learn.

We used the ESC/Java2 static checker to verify the JML implementation. The tool fails to verify some assertions due to limitations of the automatic theorem prover. Some of the prover limitations are discussed in (Kiniry et al. 2006). With these assertions commented out and after some code simplifications, other assertions can be verified. The verification results are shown in Table 3.
generated in a syntax accepted by the Isabelle proof assistant (Nipkow et al. 2002) or the RESOLVE integrated prover.

Given an assertion and an implementation, the RESOLVE verifier applies proof rules, replacing code with mathematical assertions and applying some simplifications. Assuming the soundness of the proof system, if the final assertion can be reduced to true, this implies that the first assertion is correct and hence the implementation satisfies the assertion. Since we used the RESOLVE Web tool, we were not able to collect accurate response time data as we did in the case of Dafny and JML. The tool is however relatively fast and the response time is in the order of seconds for program interpretation and VCs generation.

7. ANALYSIS AND DISCUSSION

In this section, we discuss our experience using the three specification languages, we analyze the differences and similarities among them, and their suitability for specifying Web Services.

7.1 Language Constructs

First, to implement the data model, the language that we use must allow defining a specification-only variable representing the model. Both Dafny and JML provide ghost variables for this purpose. Ghost variables are theory-typed variables that are defined only for specification purposes. RESOLVE doesn’t have the explicit notion of global ghost variables; any variable in RESOLVE that is mathematically founded can be used in the specification.

To define and specify the data operations, the language must also support mathematical functions, or side-effect free methods, that can be used in the specifications. Dafny’s mathematical functions are used for that purpose. The body of a mathematical function in Dafny defines its postconditions. Consequently, we had to implement the data operations using Dafny’s functions instead of simply specifying them as originally intended. Once defined however, these implementations can be easily reused across different models. JML on the other hand conveniently provide spec files where interface-only methods can be defined and specified using JML assertions. RESOLVE provide a similar approach through the use of Concepts which also define interface-only methods. It’s worth noting here that we made some trials to implement our model using Spec#, however, the language is lacking the necessary constructs to define theory types and hence we could not use it to define the data model.

7.2 VERIFICATION PROCESS

Out of the three languages, RESOLVE is the only one that provides a human-readable form of the Verification Conditions. This enables a programmer to inspect the VCs and detect any specification error or discover assertions that may be proven true but does not reflect the programmer’s intentions.

Listing 7. A RESOLVE concept specifying the PayPal data model and Express Checkout operations

The VCs generation component of RESOLVE is both sound and complete (Sitaraman et al. 2011)(Wayne Heym 1995). The verification component of RESOLVE is still
evolving and can currently be used to prove some of the simple VCs. The component is both sound and complete. On the other hand, the verification process of both JML and Dafny is neither sound nor complete due to limitations in the current implementations of the verifier. For example, we have intentionally introduced errors in the Dafny implementations and the verifier failed to detect these errors. Similarly, the verification process in JML failed to verify the implementation when a control flow statement is introduced as the verification space grows beyond the verifier capabilities.

7.3 TOOLS

Dafny uses the Boogie verifier, which is a command-line tool. It is also integrated with the Visual Studio IDE to provide a real-time checking of assertions. This enables detecting programming errors while coding. A Web interface is also available to try the language and its verifier on simple examples (Sitaraman et al. 2011). The language does not have yet a compiler and hence programs written in Dafny can only be verified but not executed.

To verify JML code, we use ESC/Java2 as it is relatively matured compared to other tools and has an active community. JML verification tools however have limited capabilities. They support a subset of the JML language and they don’t work with recent versions of Java, specifically with generics. Currently, there’s an effort to develop a new generation of tools, called OpenJML, which is based on OpenSDK and support recent versions of Java.

Finally, RESOLVE provides both a command-line tool and a Web interface compiler/verifier. The Web interface is particularly convenient to use and provide sample Concepts that can be adapted and reused. The RESOLVE compiler transforms the code into Java and can be used to generate the VCs for inspection. Using the Web tool, a realization can be marked with VCs, at approximate places, so that the user can connect the VCs with the code (Cook et al. 2011).

7.4 LEARNING CURVE

We share here our experience in using the three languages in terms of ease of use and learning the language constructs. The Dafny language is relatively easy to learn since it has very limited set of constructs. The language combines both procedural and functional programming and hence familiarity with both is needed in order to use the language effectively. Dafny supports a limited set of data types and hence it’s the programmer’s responsibility to compose these types into complex ones, when needed. JML is easy to learn for Java programmers as it uses similar syntax and is well integrated with the Java language. It defines a set of theory types that the programmer needs to learn.

RESOLVE defines its own syntax and program structure and hence its learning curve is steep relative to Dafny and JML. The language however defines mathematical model and related theories for simple programming types such as integers, floats, booleans and strings that can be easily reused. Many examples are also provided through the Web tool including the implementation and specification of basic data structures such arrays, stacks and queues. Table 4 summarizes the differences and similarities among the three languages.

<table>
<thead>
<tr>
<th></th>
<th>Dafny</th>
<th>JML</th>
<th>RESOLVE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VCs generation</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Soundness</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Inspection</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Variables</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Expressiveness</strong></td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td><strong>Compiler</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Integration</strong></td>
<td>None</td>
<td>Java</td>
<td>None</td>
</tr>
</tbody>
</table>

8. AUTOMATIC TEST GENERATION

In the previous sections, we described how we used the specification of data-centric services in order to apply static analysis of code and prove code correctness. Formal methods research has also recognized the benefits of leveraging the formal specification in order to automatically generate test cases (Tretmans and Belinfante 1999)(Tillmann and De Halleux 2008).

![Figure 5. Test case generation from formal specifications](http://hipore.com/ijsc)

For testing data-centric services, the generation of interesting database states becomes one of the main testing challenges. We propose using our formal model to generate
these states. The general proposed methodology is depicted in Figure 5. As shown in the figure, the service specifications are fed into a constraint solver that uses the specification to generate interesting test values for the code variables as well as the data. These test values are used to populate a test database before running the unit tests.

To evaluate the proposed methodology, we conducted an experiment where we study the effect of leveraging the data specification on the quality of tests generated by an automatic test generation tool. The experiment is conducted as follows:

1. A book rental application is implemented using C# Web services. We modeled the underlying database as a global variable, shared by all services, and we formally specified it using our proposed data model.
2. All services are specified using C# contracts. The contracts define preconditions, postconditions, invariants and the specification of non-null variables. These constructs are used to define how the services interact with the data model variable in terms of reading and/or modifying records.
3. We apply the Pex automatic test generation tool (Tillmann and De Halleux 2008) on the specified code and data model. The tool leverages the specification in order to automatically generate test cases.
4. We measure the code coverage achieved by the generated tests, at the following levels of specifications:
   - L0: No specification, this level acts as a baseline.
   - L1: Specifying only the non-null types.
   - L2: Adding both loops and class invariants to L1 specifications.
   - L3: Specifying preconditions in addition to L1 specifications.

As part of our exploratory experimentation with the tools, we applied our proposed methodology on two services that are part of the book rental application. Namely, the create_user service that creates a new administrator record given a username and password, and return_book, which marks a book as returned given the book’s ISBN. The results of testing these two services are depicted in Figure 6.

The figure shows the code coverage percentage measured at the different levels of sophistication for the specifications; L0 through L3. The results show the increase of code coverage as the specification is made more exhaustive. We actually reach 100% of code coverage for the create_user service at L2, i.e. with partial specification. While we couldn’t run the tool for a significant number of services due to the tools’ limitations, the result we obtained using the case studies are highly promising. By capturing the specification of the underlying data infrastructure, our proposed model enables the generation of interesting data states that exercise the different code branches and consequently achieves higher code coverage.

9. CONCLUSION AND FUTURE WORK

In this paper, we present our effort in implementing a data model and contracting framework for Web services using state-of-the-art specification languages. With the current state of the formal specification languages and tools, it is still not possible to fully specify and verify every property of a large-scale service composition with respect to the requirements. It is however possible to verify some properties, as it is the case with our proposed model, where we focus on specifying and verifying the data aspect. Our experiment shows that we are able to verify some of the correctness properties using the current tools. There has been also a significant progress in this area. Some recent efforts aim at integrating specification techniques into current mainstream programming languages, the C# contracts are an example. While they still lack the necessary constructs for defining abstract data types and tools for verifying complex assertions, they are however useful in detecting some logical errors in the code and enhancing automatic testing (Fähndrich et al. 2010). An interesting extension of our analysis is to study the learning curve and utility of these languages when used by developers of Web service composition. We are also performing more experimentation on automatically generating test cases from the specifications and evaluating the quality of the generated tests in terms of the achieved code coverage and the ability to detect code defects.

10. REFERENCES


Tretmans, J. and Belinfante, A. 1999. Automatic testing with formal methods.


---

**Authors**

**Iman Saleh** is an assistant scientist at the University of Miami. Her research interests include data modeling, Web services, formal methods, big data, and cryptography. Saleh has a PhD in software engineering from Virginia Tech. She’s a member of ACM, IEEE, and the Upsilon Pi Epsilon Honor Society for Computer Science at Virginia Tech. Contact her at iman@miami.edu.

Gregory Kulczycki is an adjunct professor of computer science at Virginia Tech. His research areas include component-based software development, software engineering, and formal methods. Kulczycki has a PhD in computer science from Clemson University. He’s a member of the IEEE Computer Society and ACM. Contact him at gregwk@vt.edu.

M. Brian Blake is a professor of computer science and concurrent professor of electrical and computer engineering, and human genetics at the University of Miami. His research interests include service-oriented computing, workflow systems, and software engineering. Blake has a PhD in information and software engineering from George Mason University. He’s a senior member of IEEE and an ACM Distinguished Scientist. Contact him at m.brian.blake@miami.edu.

http://hipore.com/ijsc  50
Yi Wei is a software engineer at Microsoft. His research interests include service-oriented computing, cloud computing, and agent-based modeling and simulation. He received his PhD in computer science from University of Notre Dame. He is a member of ACM and IEEE. Contact him at dawe@microsoft.com.