A Formal Model for Optimizing Dynamic Service Composition

John McDowall, Alexander Brodsky, Larry Kerschberg
George Mason University, 4400 University Drive, Fairfax, Virginia 21228, USA, jmcdowal@gmu.edu, brodsky@gmu.edu, kersch@gmu.edu

Abstract Run-time service composition has been a goal of the Service-Oriented Architecture paradigm for many years, and several projects have demonstrated means of doing this composition. However, none of these efforts has focused on optimizing the composition of available services to ensure the user is getting a composition that meets his or her needs or falls within constraints that the user must conform to. This paper describes a model for optimizing service composition, beginning with a formal definition of the service composition problem and culminating in an implementation using the Optimization Programming Language.

Keywords optimization; quality of service; modeling; services

1. Introduction

The Business Process Modeling Notation (BPMN) [3] is a graphical language widely used by business analysts to define and document processes. As discussed in [8], BPMN is also useful for providing a graphical means of describing the composition of services within a Service Oriented Architecture (SOA).

However, the use of BPMN is largely confined to the design time phase of system development because there is no standardized way to convert a BPMN model into an executable process. The Business Process Execution Language (BPEL)[6] can be used to define a service composition, but BPEL is a sophisticated language that requires programming skills that are typically outside the expertise of business analysts. Furthermore, even after business process has been converted into an executable process within a system (e.g., an orchestration of services), there is no means for analysts to ensure the service composition is the optimal use of available services. Business analysts would be well served by a means of converting a BPMN model into an optimized composition of services. There has been work on optimizing service composition, including [12, 10, 9]. However, this work did not suggest using the graphical service composition language such as BPMN, nor did it suggest a specific optimization model for BPMN-like service composition.

To enable optimal service composition based on a BPMN model, we extend the BPMN language with semantic annotations to enable mapping activities in the BPMN model to services and we develop a formal optimization model. This paper focuses on describing the optimization model and its application to ensure the optimal service composition is selected form among those available.

This work extends research described in [5, 7, 8] to develop a formal model to describe a business process and the mapping of that process to services. This work also develops a formal model for aggregating the Quality of Service (QoS) metrics of individual services to develop an optimal service composition recommendation based on the QoS metrics.
The principle focus of this paper is the formal definition of a process, the mapping of activities within a process to services, and the means for selecting an optimal service composition from among several candidate compositions. We then define several QoS metrics that can be used to express the utility to be optimized.

The remainder of this paper is organized as follows. Section 2 Provides a short overview of the service composition system, Section 3 describes the formal model on which the optimization is based. Section 4 describes the implementation of that model using the Optimization Programming Language (OPL). Section 5 describes the experimental results, and Section 6 presents conclusions and future work.

2. Service Composition Overview

This work is part of a larger effort that developed a semantic service composition framework suitable for composing both physical and digital services into an executable workflow based on a business process model. The service composition framework includes a service description language suitable for describing both the syntax and semantics of the interfaces to services. A detailed description of this language is beyond the scope of this paper; it is sufficient to state that such a semantic description of service interfaces is available. The framework also includes extensions to the BPMN modeling language necessary to define the semantics of a business process in a way that facilitates automatically matching services to process activities. These BPMN extensions also include the means for specifying the input and output parameters to each activity, necessary to ensure that services matched to sequential activities can be composed together to complete the process. Finally, the framework includes a formal QoS model for specifying QoS parameters of services and processes. The QoS model is founded on a formal optimization model that is the primary focus of this paper. When a business process model, services, and QoS parameters are encoded in this model that information is passed to an optimization execution engine that computes an optimal service composition based on the QoS aspects of each composition.

2.1. Service Composition Process

The service composition process assumes that semantically annotated service interface descriptions are available within a registry, and that these descriptions include information about the inputs and outputs of each service and the type of function that service performs.

Envision a system that enables a typical business analyst with no programming experience to create an executable business process from a BPMN model. The user develops a process model using BPMN and adds semantic annotations to the BPMN model denoting the task type and input and output parameter types of each activity using the BPMN extensions described in [8]. The user then submits the annotated process model to a processing engine that parses the BPMN model and extracts the semantic information from each of the activities in the model, as well as the ordering of the individual activities within the process model. Next, the engine compares the semantic information about the activities in the process model to the service descriptions in the registry to find those services that can perform any of the activities specified in the process model.

Once candidate services are found in the registry, their input and output parameters are compared to ensure that for each of the candidate services for a given activity, there are one or more corresponding services in the immediately preceding and succeeding activities whose inputs and outputs are compatible. After this filtering process is complete, the QoS parameters for the services in each composition are retrieved, and the candidate service compositions are prepared for use in the optimization analysis.

The optimization analysis is based on the formal model described in Section 4. The analysis compares each of the possible compositions that can be assembled from the available services and develops a recommendation for the optimal composition based on the QoS metrics for each of the services.
2.2. Elements of Service Composition

A detailed description of the service composition process is beyond the scope of this paper, but a brief overview overview of the important elements is in order.

2.2.1. Atomic Services

There is no commonly accepted formal definition for an atomic service; for the purposes of this discussion an atomic service is the lowest level to which services are decomposed and is the level at which QoS metrics are assigned to services.¹

2.2.2. Virtual Services

In some cases, multiple atomic services may be composed together and offered through a single interface. This arrangement is known as a virtual service. The simplest virtual service is composed of individual atomic services, but it is also possible to compose a virtual service from combinations of other virtual services and atomic services. Given this definition, every business process also constitutes a virtual service and could be offered as such.

2.2.3. Service Composition

We begin with a process model defined using BPMN such as that depicted in Figure 1 which shows a simple weather forecast process. We use a trivial process as an example to ensure the relationships between elements of the process model and the formal definitions are clear and concise. The example process is composed of two activities: Convert Location and Get Weather. The order of these activities is specified by the arrows in the process model, which BPMN refers to as “sequence flows.”

Each of the activities in this process is assigned a “task type” that provides a reference to the type of work that activity represents. For this example, the Convert Location activity has a task type of “locConvert.” Each activity also has a set of input parameters and a set of output parameters; each of these parameters is identified by the semantic type of the parameter. The set of inputs for the Convert Location activity is shown in Table 1 and the set of outputs for the Convert Location activity is shown in Table 2.

In order to automate this process, we must map services to each of the activities in the process model. An activity to service mapping is only valid if each of the services mapped to the activity has the same task type as the activity and inputs and outputs of the same

¹ As a practical matter, the point at which a service can be defined has been moving lower in the 7-layer ISO stack, to the point where we are now speaking of Infrastructure as a Service. This definition is therefore necessarily arbitrary for the purposes of this discussion.
Some of the selected services may be virtual services. In addition to the inputs, outputs, and task type that the virtual service performs, a virtual service encapsulates a set of activities that comprise the virtual service. Given that a virtual service is a composition of other services, we can envision a virtual service as a process model which can be decomposed into individual activities that are in turn mapped to other services. This decomposition can be recursive, with any given service potentially decomposed into a series of more specialized services until eventually all virtual services have been decomposed down to individual atomic services. This decomposition results in an acyclic tree structure where each leaf of the tree is an atomic service and all other nodes in the tree are virtual services. An illustration of the mapping of a process model to a services to create a virtual service instance is depicted in Figure 2.

Once all available services have been assessed and mapped to activities in the process model, and all virtual services are decomposed into their atomic services, we can determine whether any combination of atomic services can be composed into an executable process that completes all of the activities in the original business process model.

In order to compose services into an executable process, it is necessary that the inputs for each service in the composition be provided by a preceding service in the composition. This is determined by matching the semantic types of each input of each service to the semantic types of the outputs of preceding services. These outputs and inputs need not have the same name, but they must have the same semantic type. For example, if a location service has an output called zip code and a weather service has a single input called postal code, we can see intuitively that these two parameters have the same meaning. But in order for the matchmaker to match them, the parameters must have the same semantic type. An individual service input or output parameter, together with associated semantic and type metadata, is called a semantic parameter. For example, a data element called zipCode would include a semantic annotation that references an ontology and a type annotation indicating it is stored as a string.

Once these assessments are complete for each of the services that have been mapped to an activity, we can calculate a set of candidate service compositions one or more sets of atomic services that can be composed to complete the process originally defined in the BPMN.
model. Calculating the set of candidate service compositions is a straightforward matter if the services are analyzed as a directed graph. First, consider each service that has been mapped to an activity as a node in the graph. For each case in which one service outputs provide the inputs required by another service, assert an edge from the former service to the latter service. The result is a directed graph of the services, where each path from the first activity to the final activity constitutes a candidate service composition. An example of such a graph is shown in Figure 3.

The candidate compositions that can be assembled to complete the process can be discovered by finding all the paths from services matching the first activity to those matching the final activity. In the example depicted in Figure 4, there are four candidate service compositions: convertLocation-getWeather, convertLocation-returnWx-changeFormat, transformLocation-getWeather, and transformLocation-returnWx-changeFormat. Finding these paths through the directed graph of services can be accomplished using common graph analysis algorithms such as Floyd-Warshall [2].

Once all of the candidate service compositions have been calculated, it is possible to calculate the QoS of each composition. The QoS metrics of each of each service in each of these candidate compositions is retrieved from the service description, and the QoS metrics for each candidate composition are passed to the QoS analysis model. The QoS analysis model compares each of the candidate service compositions and recommends the optimal composition based on the QoS parameters of each service and the overall workflow, as determined by the users preferences (e.g., minimize cost).

Upon completion of the QoS analysis, the optimizer returns a recommended optimal service composition based on the QoS parameters.

3. Formal Model

Intuitively, the service composition optimization problem is as follows: given a desired process, a set of services, constraints, and an objective such as minimizing cost, select the set of services that completes the process and best meets the objective within the constraints.

The formal definition of service composition based on a process model is as follows (this discussion mirrors the intuitive discussion in Section 3).
Let $T = \{t_1, \ldots, t_n\}$ denote a set of task types. For example, $t_i \ (1 \leq i \leq n)$ can be the task type “reserve a hotel room.” If a service performs the same function as an activity in the process model, then we say they have the same task type.

Let $SP$ denote the set of all semantic parameters.

**Definition:** Given $T$ and $SP$, a service $s$ is a tuple $(id, I, O, T)$, where

- $id$ is a unique identifier
- $I \subseteq SP$ is the set of input semantic parameters
- $O \subseteq SP$ is the set of output semantic parameters
- $T \in T$ is the task type that describes this service

The above definition includes a unique identifier so that similar services offered by different providers can be distinguished from each other. Such a tuple defines an atomic service sufficiently.

A virtual service, defined below, is needed to enable the recursive composition of services.

**Definition:** A virtual service $s$ (also called a process) is a tuple $s = (id, I, O, T, A, DG, aTask : A \rightarrow T, S)$, where

- $id$ is a unique identifier
- $I \subseteq SP$ is the set of input semantic parameters
- $O \subseteq SP$ is the set of output semantic parameters
- $T \in T$ is a process task type associated with the virtual service $s$
- $A = \{a_1, \ldots, a_n\}$ is a set of activities used in $s$
- $DG \subseteq A \times A$ is an activity precedence graph that must be acyclic. $(a_1, a_2) \in DG$ (also denoted $a_1 \prec a_2$) means that activity $a_1$ must precede activity $a_2$
- $aTask : A \rightarrow T$ is a mapping that associates every activity $a \in A$ to its task type $t = aTask(a)$ in $T$
- $S = \{s_1, \ldots, s_n\}$ is a set of services that can be used by activities in $A$

Note that a virtual service is a service, and any service in $S$ may itself be a virtual service that fits this definition. Therefore, multiple services can be used for each activity in a virtual service. A particular instantiation is formalized in the following definition.

**Definition:** Given a virtual service $s = (id, I, O, T, A, DG, aTask : A \rightarrow T, S)$, an activity-to-service mapping $A2S : A \rightarrow S$ is a mapping that associates each activity in $A$ with a service in $S$, that must satisfy the following properties:

- Let $sTask(s)$ denote the task $T$ associated with the service $s$ in $S$;
- let $SI(s)$ denote the input set $I$ associated with $s$;
- let $SO(s)$ denote the output set $O$ associated with $s$;

The $A2S$ mapping must satisfy:

$(\forall a \in A)aTask(a) = sTask(A2S(a))$

$(\forall a \in A)SI(A2S(a)) \subseteq PrecOut(a)$,

where $PrecOut(a)$ denotes the output of the services preceding service $a$

$PrecOut(a) = \bigcup_{b \prec a} SO(A2S(b))$ (i.e., $b$ is the set of all outputs produced by activities / services that precede $a$)
This definition allows us to map each activity within a virtual service to one or more services that have the same task type as the activity. Each service so mapped must also have all its inputs supplied by the outputs of services mapped to one or more preceding activities.

The notion of a virtual service instance, defined below, describes a recursive mapping of activities to available services for a given virtual service vs.

Definition: Let AS be the set of atomic services and VS be the set of virtual services. A virtual service instance (VSI) over \((AS, VS)\) is a tuple \(V = \langle S, vs, \{A2S_s\}_{s \in S \cap VS} \rangle\), where

- \(S \subseteq AS \cup VS\)
- \(vs \in S \cap VS\)
- \(\{A2S_s\}_{s \in S \cap VS}\) is a set of activity-to-service mappings \(A2S : s.A \rightarrow s.S\) where \(s.A\) and \(s.S\) are the set of activities and services of \(s\), respectively,

such that the following conditions are satisfied:

1. \(s.S \subseteq S\)
2. \((\forall s.S)SI(s) \subseteq PrecOut(s)\), where \(PrecOut(s)\) denotes the outputs of the services preceding service \(s\)
3. \((\exists s_1, s_2 \in S \cap VS)(\exists a_1 \in s_1.A)(\exists a_2 \in s_2.A)(A2S_{s_1}(a_1) = A2S_{s_2}(a_2))\) (i.e. no two activities within services of \(S \cap VS\) can be mapped via \(A2S\) to the same virtual service)

The result of this recursive mapping is a series of atomic services that may be chained together to perform the process specified in the original BPMN model.

We would like to find the “optimal” virtual service instance from among those available. To do this, we establish several quality of service (QoS) factors that can be used to express the utility to be optimized. Formal definitions for each of these factors are provided below. The QoS metrics being considered are cost, duration, satisfaction, and unity. We are given the QoS metrics for atomic services, and we define QoS metrics for a virtual service. The definition of the cost of a virtual service instance is provided below.

Definition: Given a virtual service instance \(V = \langle S, vs, \{A2S_s\}_{s \in S \cap VS} \rangle\) over \((AS, VS)\) the cost of \(s, \forall s \in S\), denoted \(cost(s)\), is defined recursively as follows:

- \(\forall s \in S \cap AS\) \(cost(s) = C(s)\) where \(C(s)\) is the cost of atomic service \(s\)
- \(\forall s \in S \cap VS\) \(cost(s) = \sum_{a \in s.A} cost(A2S(a))\)

The cost of \(V\), denoted \(cost(V)\), is defined as \(cost(V) = cost(vs)\)

The duration of a virtual service instance, which intuitively is the expected time for the entire composition to run from initiation until completion of all services within the virtual service instance, is defined next.
will primarily focus on implementation of the optimization model described in Section 3. This discussion will provide only a brief overview of the service composition framework; it ensures they functioned as expected. A detailed description of the entire service composition framework can be found in [5, 7]; this discussion will provide only a brief overview of the service composition framework; it will primarily focus on implementation of the optimization model described in Section 3.

### Definition: Given a virtual service instance $V = \langle S, vs, \{A2S_s\}_{s \in V S} \rangle$ over $(AS, VS)$ the duration of $s, vs \in S$, denoted $\text{duration}(s)$, is defined recursively as follows:

- $\forall s \in S \cap AS \ \text{duration}(s) = D(s)$ where $D(s)$ is the duration of atomic service $s$
- $\forall s \in S \cap VS \ \text{duration}(s) = \max \{ \text{endtime}(a) | a \in A \}$

  Where $\text{endtime}(a)$ is defined as follows:

  - If $a \in A$ does not have a preceding activity (i.e., $\text{Prec}(a) = \emptyset$): $\text{endtime}(a) \overset{\text{def}}{=} \text{duration}(A2S_a(a))$,
  - otherwise: $\text{endtime}(a) \overset{\text{def}}{=} \max \{ \text{endtime}(b) + \text{duration}(A2S_a(a)) | b \in \text{Prec}(a) \}$

  The duration of $V$, denoted $\text{duration}(V)$ is defined as $\text{duration}(V) \overset{\text{def}}{=} \text{duration}(vs)$

The rating of a service is a measure of users ratings of a service, such as rating a service on a scale of 1 to 10. We assume that each services individual rating has been normalized to the range 0..1. The notion of rating for a virtual service instance is defined below.

### Definition: Given a virtual service instance $V = \langle S, vs, \{A2S_s\}_{s \in V S} \rangle$ over $(AS, VS)$ and the rating of each atomic service, the rating of $s$ is denoted $r(s)$, is defined recursively as follows:

- $\forall s \in S \cap AS \ r(s) = R(s)$ where $R(s)$ is the rating of atomic service $s$
- $\forall s \in S \cap VS \ r(s) = \sum_{s \in s \cap A} r(A2S_s(a)) / |s, A|$ Where $|s, A|$ is the number of activities in $s$

The rating of $V$, denoted $r(V)$ is defined as $r(V) \overset{\text{def}}{=} r(vs)$

Knowing how to calculate each of the QoS parameters across a service composition, we can define the optimal service selection as a utility function over $C$, $D$, and $R$.

### Definition: Given the following input:

- Sets $AS$ and $VS$ of atomic and virtual services respectively
- A root service $rs \in VS$
- An objective expressed as a function $O : D(\text{cost}) \times D(\text{duration}) \times D(\text{rating}) \rightarrow \mathbb{R}$ that gives a value $O(C, D, R)$ for cost $C$, duration $D$, and rating $R$
- Minimum or maximum
- Constraint $C$ is a Boolean expression in terms of $C$, $D$, and $R$ that defines $C : D(\text{cost}) \times D(\text{duration}) \times D(\text{rating}) \rightarrow true, false$

An optimal service instance $vsi$ is defined as $vsi \overset{\text{def}}{=} \arg \min O(C(i), D(i), R(i))$ where $VSI$ is the set of all virtual service instances over $(AS, VS)$ with root service $rs$ subject to $(C(C(i), D(i), R(i)))$ where minimum is required. The definition is similar for the case where a maximum is required.

## 4. OPL Implementation

In order to demonstrate the feasibility and utility of the contributions of this research, a proof of concept prototype was developed to exercise the components of the framework and ensure they functioned as expected.

A detailed description of the entire service composition framework can be found in [5, 7]; this discussion will provide only a brief overview of the service composition framework; it will primarily focus on implementation of the optimization model described in Section 3.
To validate the definitions provided in Section 3 and demonstrate the utility of the optimization model described, we implemented the model in the Optimization Programming Language (OPL) using IBM’s CPLEX optimization engine.

4.1. Agent-Based Framework

The proof of concept prototype uses an agent-based system built on the JADE framework. In prototype, a BPMN model with semantic annotations added to each activity is passed to a parsing agent that reads the task type of each activity and launches a specialized task agent for each activity type.

Each task agent searches the service registry for services with the same task type as the task agent. Each service with a matching task type is further examined to determine the semantic types of its input and output parameters. Each service is loaded into a directed graph structure as a node. Once the directed graph is populated with nodes, each service’s inputs are compared to every other service’s outputs. In each case where one service’s required inputs are supplied by the outputs of another service, an edge in the graph is asserted from the latter service to the former.

After all services have been compared to every other service and edges have been asserted in the graph, the Floyd-Warshall algorithm is used to find all paths from each service corresponding to the first activity in the BPMN model to each service corresponding to the last activity in the BPMN model.

4.2. Optimization Implementation

Once all paths through the service graph have been discovered, the semantic information from the BPMN model and the services that form each path through the graph, together with the QoS information for each atomic service, is written out using the OPL data model syntax. A sample of this syntax that encodes the model shown in Figure 1 follows (for simplicity, this example shows only the cost QoS parameter):

\[
\begin{align*}
SP &= \{\text{wxForecast}, \text{locationZipCode}, \text{locationLatLon}\}; \\
A\text{services} &= \{\text{getWxForecast}, \text{convertLocation}\}; \\
V\text{services} &= \{\text{generalWeatherService}\}; \\
activationCost &= [5.0, 2.5]; \\
Tasks &= \{\text{getWeather}, \text{alterLocation}\}; \\
Inputs &= \{\text{locationLatLon}, \text{locationZipCode}, \text{locationZipCode}\}; \\
Outputs &= \{\text{wxForecast}, \text{locationLatLon}, \text{wxForecast}\}; \\
task &= \{\text{getWeather}, \text{alterLocation}, \text{virtualGetWeather}\}; \\
Activities &= \{\text{ConvertLocationType}, \text{GetWeatherForLatLon}\}; \\
PrecActivities &= \{\}, \{\text{ConvertLocationType}\}; \\
aTask &= \{\text{alterLocation}, \text{getWeather}\}; \\
rootVservice &= \text{generalWeatherService};
\end{align*}
\]

This example is explained as follows. The line

\[
SP = \{\text{wxForecast}, \text{locationZipCode}, \text{locationLatLon}\};
\]

encodes the set of semantic parameters \(SP\) used by all services. The set of atomic services \(AS\) is encoded in the line

\[
A\text{services} = \{\text{getWxForecast}, \text{convertLocation}\};
\]

and the set of virtual services \(VS\) is encoded as

\[
V\text{services} = \{\text{generalWeatherService}\};
\]
(note there is only one virtual service in this example).

The cost QoS parameter is encode in the line

\[
\text{activationCost} = [5.0, 2.5];
\]

denotes that the atomic service getWxForecast has a cost of 5.0 and the atomic service convertLocation has a cost of 2.5.

The set of task types is encoded as

\[
\text{Tasks} = \{\text{"getWeather"}, \text{"alterLocation"}\};
\]

this set of task types includes the task types for all services and activities within the model.

The set of input semantic parameters \(SI\) and the set of output semantic parameters \(SO\) are encoded

\[
\text{Inputs} = \{\{\text{"locationLatLon"}\}, \{\text{"locationZipCode"}\}, \{\text{"locationZipCode"}\}];
\]

\[
\text{Outputs} = \{\{\text{"wxForecast"}\}, \{\text{"locationLatLon"}\}, \{\text{"wxForecast"}\}, \};
\]

Each of the input and output parameters is a member of \(SP\). The ordering of the input and output parameters is matched to the order of the atomic services followed by the virtual services. For example, the input \text{“locationLatLon”} is the input to atomic service \text{“getWxForecast”} and the output \text{“wxForecast”} is the output of virtual service \text{“generalWeatherService”} and so forth.

Each virtual service in VServices has its Activities spelled out in the line

\[
\text{Activities} = \{\{\text{"ConvertLocationType"}, \text{"GetWeatherForLatLon"}\};
\]

where the activities for the virtual service \text{“generalWeatherService”} are listed in the order they appear in the process model associated with the virtual service. This is complemented by the line

\[
\text{PrecActivities} = \{\}, \{\text{"ConvertLocationType"}\};
\]

which lists the preceding activity of each activity in Activities.

The array \text{“aTask,”} encoded as

\[
\text{aTask} = \{\text{"alterLocation"}, \text{"getWeather"}\};
\]

lists the task type for each activity in Activities.

Finally, the line

\[
\text{rootVservice} = \text{"generalWeatherService"};
\]

denotes the “root virtual service” for this optimization problem; that is, it specifies which of the VServices is the original business process that the user wanted to automate.

### 5. Experimental Results

This model has been tested using a variety of data configurations, beginning with a single virtual service containing a single activity that maps to a single atomic service and progressing through a series of increasingly complex test sets to verify the correctness of both the definitions described above and the model implementation.
5.1. Testing

Initial tests were conducted using hand-crafted data sets designed to simplify troubleshooting and verification. These tests identified some issue with the initial model that required modification to some of the definitions and the implementation (these modifications are reflected in the definitions and code snippets shown above).

The tests were made progressively more complex, progressing to a hand-crafted model containing nested virtual services. For example, one activity in the root virtual service mapped to a virtual service with multiple activities; some of these activities mapped to additional virtual services, ultimately mapping to individual atomic services.

Once the model and the OPL implementation were verified using manually created test data, the OPL implementation was integrated into the larger service composition framework to test its behavior in a more realistic environment. After the service composition graph is constructed, a graph analysis agent retrieves the QoS metrics for each service that is part of a complete path through the service graph. These QoS metrics, together with the task type and other information from the service description, is written out in the format described in Section 4. The IBM CPLEX engine is passed to the optimization model and the result (i.e., the optimal service composition) is written out.

5.2. Results

Testing with manually generated data was invaluable in verifying the correctness of the definitions and the implementation. Testing with very simple data sets enabled us to correct errors in the definitions and the model that would have been very difficult to troubleshoot using more complex test data sets.

The progressively more complex testing verified that the recursive definition of an activity to service mapping is correct and that our implementation of those definitions is OPL is also correct. This was proved for cases where virtual services are nested three or more layers deep within the activity-to-service mapping process.

Integrating the model with the agent-based composition framework verified that the entire process, beginning with an extended BPMN model and culminating in an optimal service composition, is a valid approach to automatically generating service compositions from a business process model. This integration provided an additional layer of verification that the definitions and implementation are correct. It also provided further verification that the agent-based composition process is correct by ensuring that the optimal service composition produced by the CPLEX engine is one of those produced as part of the graph analysis.

6. Conclusions and Future Work

We conclude that the implementation described in this paper provides a valid approach for optimizing the composition of services. We provide formal definitions for each aspect of the service composition problem and successfully apply those definitions to the optimization implementation. We verified these definitions and this implementation are correct through the use of both manually-generated test data and by integrating this optimization model with an agent-based semantic service composition framework.

One potentially interesting area for additional work is to extend the OPL model to enable the semantic analysis of service and activity task types as well as input and output parameters. OPL is an inherently syntactic language, and in order to match the task types of an activity and a service it is necessary that they be an exact syntactic match. Our current implementation resolves this issue during the agent-based composition process; when writing out the OPL data file the graph analysis agent resolves all task types and parameters to a canonical format. For example, if two services that perform the same function have task types of “getWeather” and “getForecast,” they are resolved to the single task type “getWeather” when that information is written to the OPL model. Whether it is possible
to develop a model in OPL that enables the sort of semantic analysis that sees those two task types as identical is an open question.

Another interesting area of future research is extending out approach with a more elaborate resource management, for examples in situations where multiple services use or share the same physical resource. Some ideas from [11] may be relevant for the developing of this extension, but it would also require the appropriate extension of the BPMN semantics model.

References


