EVALUATING IMPLEMENTATIONS OF SERVICE ORIENTED ARCHITECTURE FOR SENSOR NETWORK VIA SIMULATION

By

Caleb Isaiah MacDonnell Vincent

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

____
Boleslaw Szymanski, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2011
(For Graduation May 2011)

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

The author acknowledges partial support for the work on this thesis sponsored by US Army Research Laboratory and the UK Ministry of Defense under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government.

The author would also like to acknowledge the supreme patience and love exhibited by his Fiancé, Meredith, when putting up with his recurring and continual lack of presence at their place of residence.

The author would also like to thank Sahin Cem Geyik, whose previous and current research is the starting point for this work.

ABSTRACT

With the use of Service Oriented Architectures in areas such as sensor network, it becomes increasingly necessary to be able simulate the interactions of such a system before actually implementing it. This work explores the viability of two different implementations of a simulation of service composition in dynamic sensor networks. The first simulation is written in C++ with a graphical user interface using Qt and visualizations supplied via Graphviz. This simulation supports distributed and centralized composition with a type hierarchy and multiple-service statically-located nodes in a 2-dimensional space. The second simulation uses the actor model programming language SALSA to support concurrent distributed service composition with a type hierarchy and dynamically located services in spherical space.

1. INTRODUCTION AND HISTORICAL REVIEW

The software was developed with the ability to simulate both centralized and distributed service compositions in dynamic sensor networks as described in [1]. The simulator is designed to work with abstract type and service specifications to allow for simulation of a broad range of possible situations.

1.1 Service Oriented Architecture & Service Composition

A Service Oriented Architecture (SOA) is a system design that provides functionality via a set of services that each perform a specific task. These services can then be combined, or *composed*, to form a higher level function [2]. For example, a hypothetical banking system could contain a set of services one that makes deposits and one that makes withdrawals. To preform a transfer of funds from one account to another, a service can be created that uses the services to preform a withdrawal from one account and deposit it into another account.

Service Composition, within the bounds of this research, is considered the act of determining the set of services that supply the needed inputs for a given service to function properly.

2. SOFTWARE IMPLEMENTATION

2.1 Programming Language

The simulator software is written in C++, using code complaint with the ISO/IEC 2003 specification [3]. Using features of the upcoming release of an updated C++ [4], usually referred to as C++0x was planned, however, this was abandoned due to possible lack of support and inconsistencies between compilers.

C++ was originally chosen for its portability between systems and speed of execution verses its most likely alternatives [5]. It should be noted that at the initial design and conception of this simulator software, there was no plan for a graphical user interface (GUI). This feature was added after a working version was produced, so the availability of GUI libraries was not taken into account at this time.

2.2 Libraries and System Tools

While C++ and it's Standard Template Library (STL) is the basis of the simulator software, the final functionality could not be produced with just this alone, without the creation or use of further libraries or tools.

To provide a GUI the Qt framework was selected. Qt provides all of the GUI functionality used by the simulator software. Furthermore, many of it's other features were used in the simulator's internal workings, such as Signals and Slots, and `foreach`. The selection of Qt was further influenced by it's availability on multiple platforms, thus keeping with the intention of making this software as available as possible.

Visualization of type trees, and node, service and composition graphs is handled using the DOT language and the Graphviz graph visualization software package, and is displayed in the GUI without the use of temporary files through the use of Qt's `QtSvg` module.

2.3 Software Design

The design process of the simulation software can be divided into two distinct categories. The first being the design of the core simulator. The second being the adaptation and integration of the simulator with a GUI.

2.3.1 Core Simulator Early Stages

The core simulator was first designed without a GUI. It would take a text configuration file as a command-line argument and then run the simulation based on the input. It would display the composition information in the standard output and save the node, service and composition graphs as images files using Graphviz. Further discussion of the core simulator is in Section 2.3.3.

2.3.2 GUI

The GUI is made up of a single class: `Gui`, containing a nested class `FileLoad` and having a class `TextStream`.

The class `Gui` is generally responsible for displaying data. To do this, `Gui` inherits from the Qt class `QMainWindow`, this allows on toolbar widget (`FileLoad`), and a central widget. `FileLoad` will be discussed later in the text. The central widget is an instance of Qt's `QTabWidget`, allowing different graphs and other data to be displayed simultaneously in separate tabs. The first tab always displays the `TextStream` class, which will be discussed later. The following tabs can display a graph received by the `NewGraph` message. The graphs received via `NewGraph` are expressed in the DOT language, which are then passed through Graphviz's `dot` command and displayed as a Qt QSvgWidget in the tab.

The class `FileLoad` is a simple container inheriting from Qt's `QStatusBar` allowing several buttons (Load, Start, Pause, Stop) and a textbox to be placed in the toolbar. When one of the buttons in `FileLoad` is pressed, the signal generated is forewarded to `Gui` where the needed GUI states are changed and the needed messages are sent (see Figure 2.3).

The Class `TextStream` inherits from Qt's `QPlainTextEdit`, which implements a multi-line plain-text box. `TextStream` adds additional functionality to QPlainTextEdit, specifically the overloaded insertion operator (`<<`). The insertion operator allows the `TextStream` to act as `std::cout` would, and displays the input stream in a tab as a QPlainTextEdit. A single instance of `TextStream` is made global, so it is available as a drop-in replacement for `std::cout`. There is one limitation upon `TextStream`

Figure 2.1 Messages Passed *Note: Does not include messages generated by Qt classes*

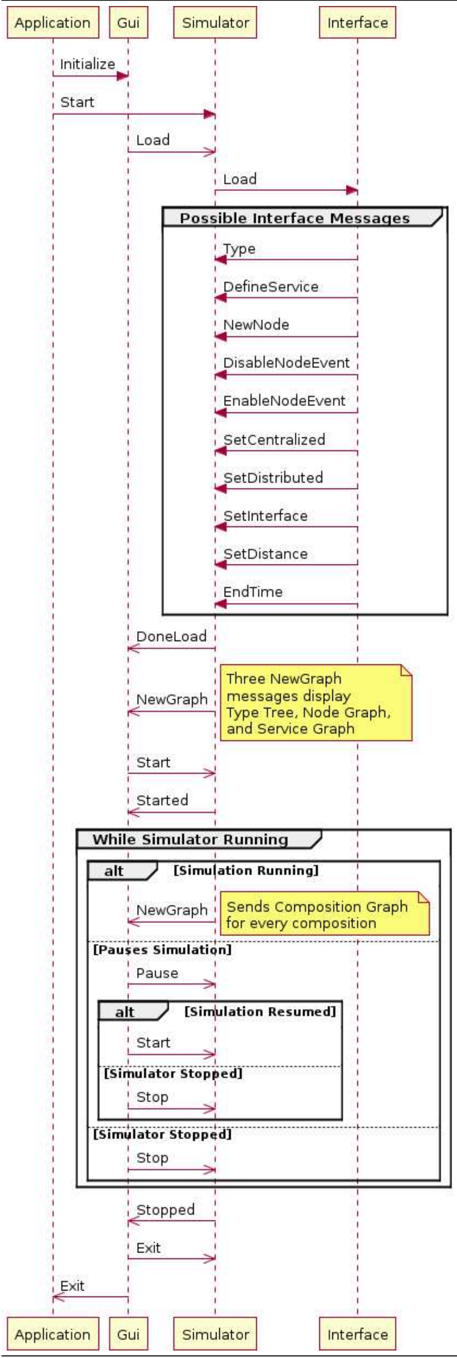


Figure 2.2 Class Diagram

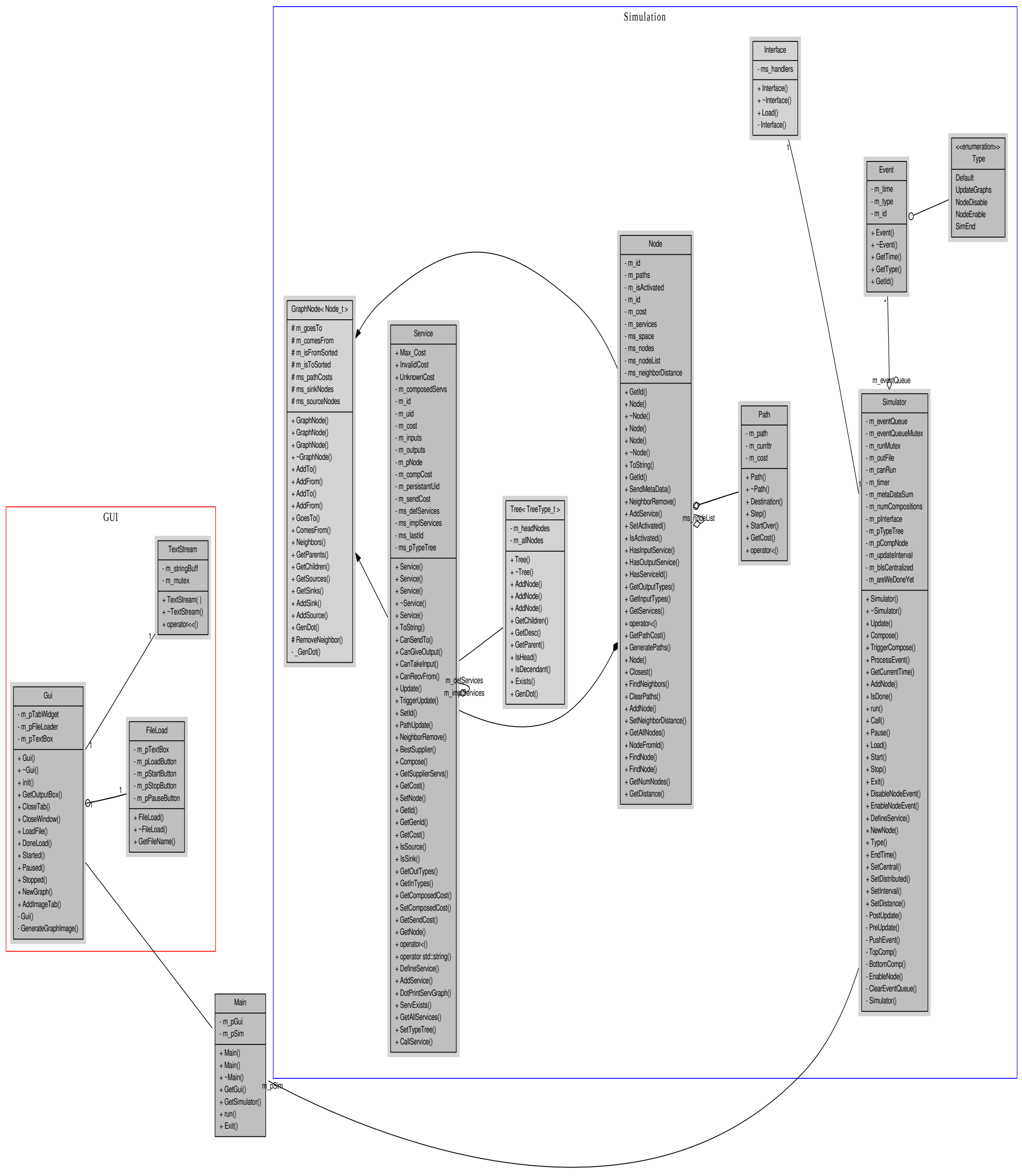
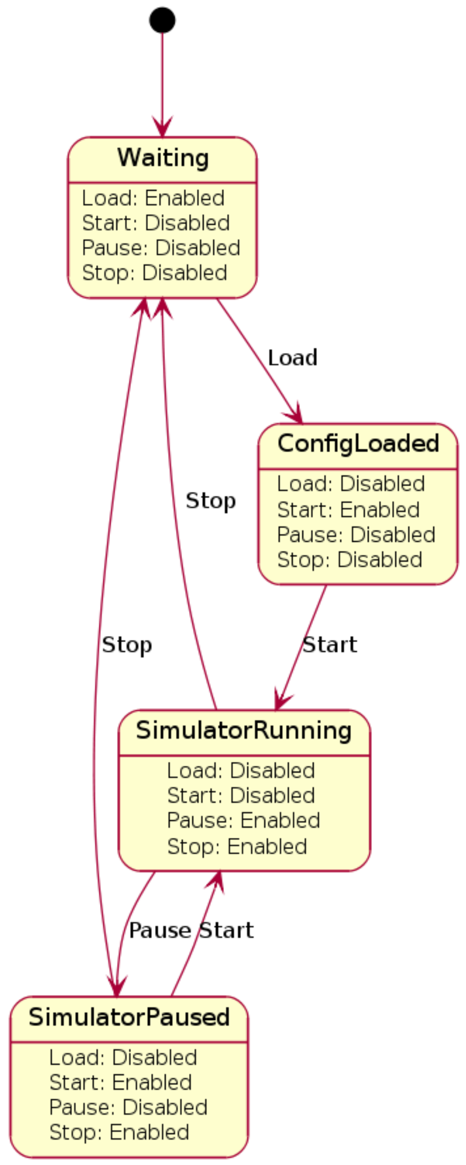


Figure 2.3 States



that differs from `std::cout`. The manipulator function `std::endl` is not available, therefore, the buffer is flushed automatically, and "`\n`" should be used as a newline.

2.3.3 Simulation

The central class of the simulation subsystem in `Simulator`. This class controls the simulation events and configuration of the simulation. `Simulator` has a class `Interface` which is responsible for parsing the configuration file and sending the correct signals with the command parameters. The slots connected to `Interface`'s signals are located in `Simulator`, which then sets member variables or calls the correct functions.

`Simulator` also has a member `m_eventQueue` a queue of class `Event`. `Event` consists of three member variables, a time, an id, and a event type, as defined in the `Type` enumeration.

Simulation data, such as the location, availability, types, and neighbors of nodes and services, are stored in two classes: `Node` and `Service` respectively. Both `Node` and `Service` inherit form the class `GraphNode` that implements the graph aspects of each class.

`GraphNode` has static members that track the source and sink vertices of the graph and static methods to access and modify them. `GraphNode` also provides each vertex with lists of incoming and outgoing neighbors and related useful access and test methods.

The `Node` has static members containing all of the nodes in the simulation along with their locations. Static methods are also included for appropriate access and testing of static members.

2.4 Used Algorithms

2.4.1 Composition

The problem of selecting services to compose a service resembles the cover set problem. The cover set requires that a set of graph vertices be selected such that every grouping of vertices has at least one vertex selected, as shown in Figure 2.4. However the weighted cover set problem is even closer to the composition problem. In the weighted cover set, vertices are selected for having the lowest cost, as shown in 2.5.

The example of service composition in Figure 2.6 shows that this algorithm incorporates the idea of the weighted cover set, trying to achieve the lowest cost while including all of the required types. In this example, types are denoted by the circles labeled *A, B, C,* and *D*, and services are denoted by a rectangle. *Service 4* is a sink service, and *Service 1, 2,* and *3* are sources, thus reliving the burden of distinguishing between input and output types.

As described in [1], there are both top-down and bottom-up versions of the

Figure 2.4 Cover Set Example

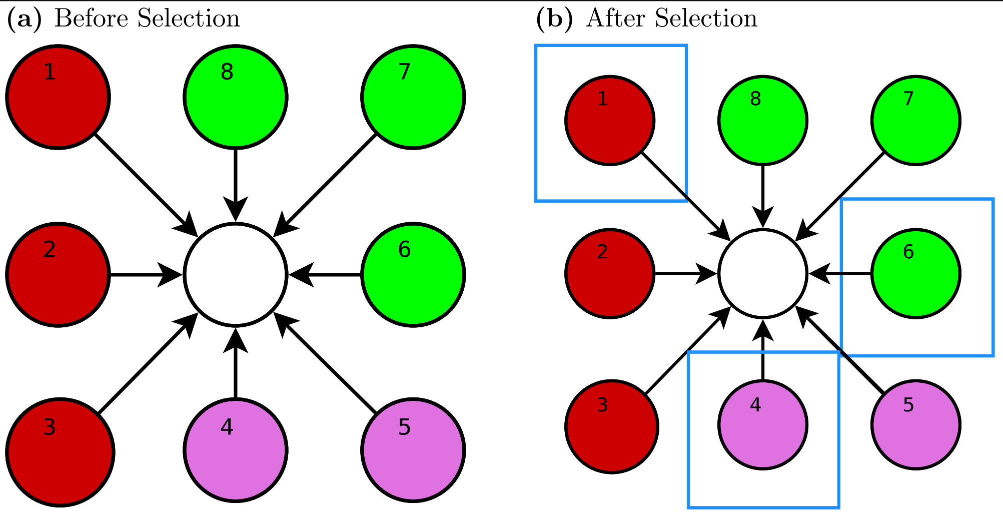
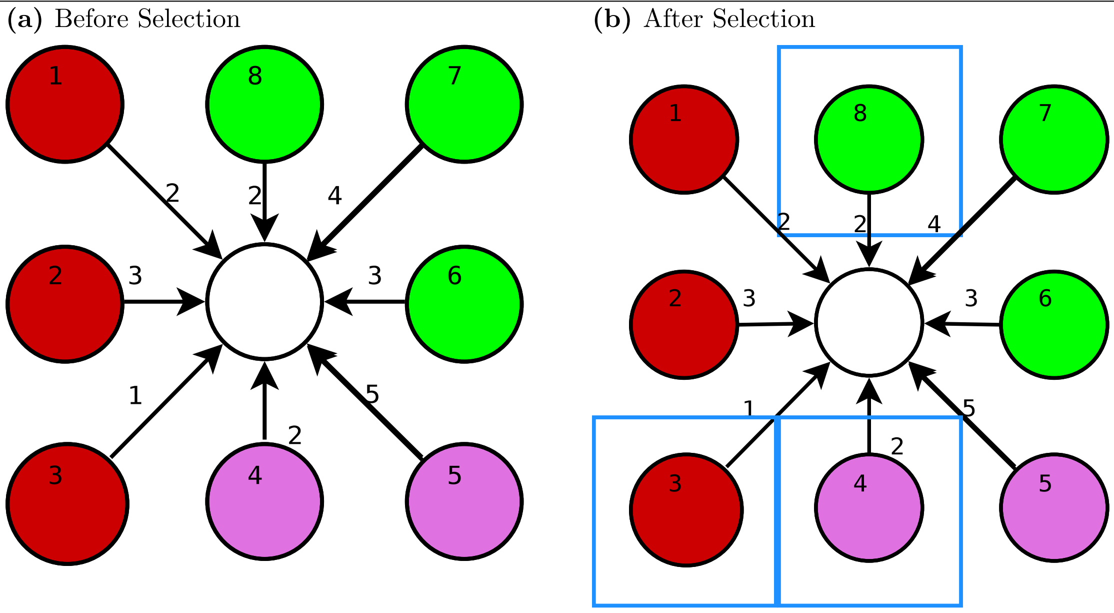


Figure 2.5 Weighted Cover Set Example



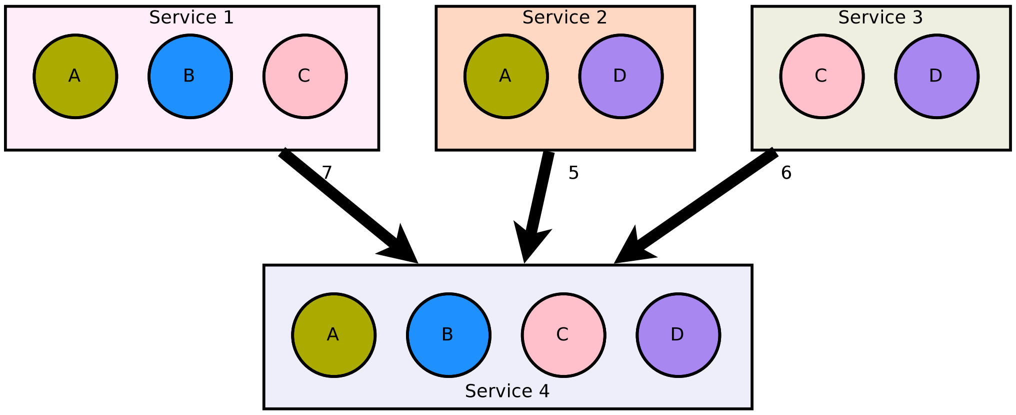
service composition algorithm. Both the bottom-up and top-down algorithms are run during centralized simulations with the lowest cost solution being selected. During a distributed simulation, only the bottom-up algorithm is run.

2.4.1.1 Bottom-Up Composition

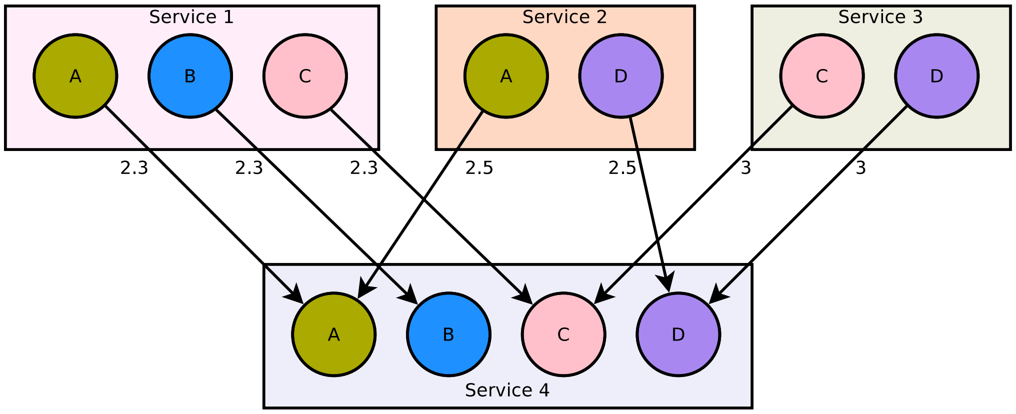
The bottom-up algorithm as implemented in the C++ simulation is shown in Algorithm 2.1 with a subpart in Algorithm 2.2. These correspond to the functions

Figure 2.6 Service Composition Example

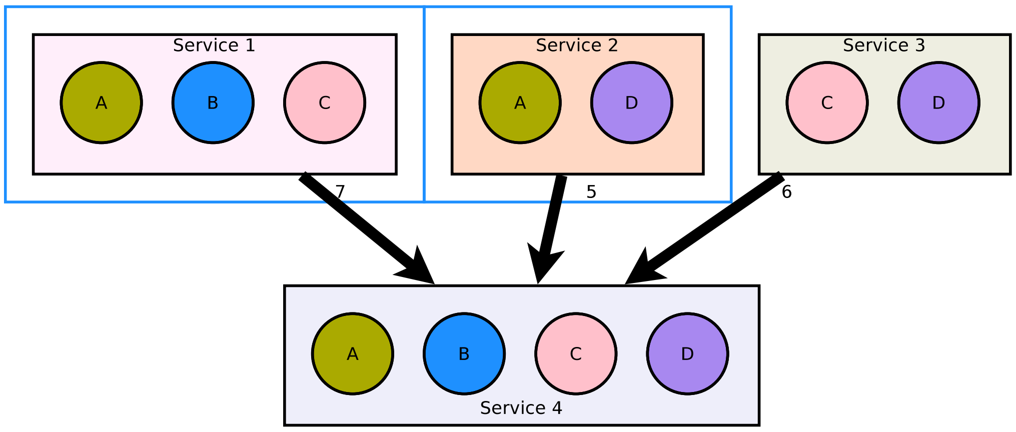
(a) Before Composition



(b) During Composition



(c) After Composition



`Simulator::Compose()`, `Simulator::BottomUp(Service*)`, and
`Service::Compose()`

Algorithm 2.1 starts the bottom-up composition algorithm, it does this by first starting with the source services in a set of the current level of services, and a set of the remaining services to compose, which initialize to all services. As the algorithm progresses services that have been composed are removed from the remaining set and the current level, and their output connected neighbors are added to the next level of services to be processed. If a service cannot compose it is added to the next level to be composed. At the core is of Algorithm 2.1 is Algorithm 2.2, which is run on each service.

Algorithm 2.2 forms the service composition for a given service S. First the algorithm checks to make sure that none of the services that could supply this one are uncomposed, if one or more is, the algorithm returns uncomposed. The main part of the algorithm cycles while there are types left to be supplied, and selects input neighbor services with the lowest per-type cost. All the types supplied by the selected input service are then removed from the remaining types. This process continues until there are no remaining types, or there are no remaining input services. A slightly simplified version of this is shown in 2.6, the difference being the costs. In bottom-up service composition, the cost of a service is its processing cost plus the sum of each of its supplying services' composition costs plus the edge weight times its sending cost. Composition cost for a source service is its processing cost.

These costs are also shown in Equation 2.1 where $C_{c,j}$ =: the composition cost of service S_j, $C_{p,j}$ =: the processing cost of S_j, S =: services supplying S_j, $C_{c,i}$ =: composition cost of S_i, and $C_{s,i}$ =: sending cost of S_i.

$$C_{c,j} = C_{p,j} + \sum_{\forall S_i} C_{c,i} + C_{s,i} \cdot c(e_{i,j}) \tag{2.1}$$

2.4.1.2 Top-Down Composition

The top-down service composition algorithm is shown in Algorithms 2.3 & 2.4. The bottom up algorithm starts with the sink nodes and works its way down the service graph to the sources using much the same methods as the bottom up algorithm.

Algorithm 2.1 Bottom-Up

```
procedure BottomUp(allServices, sources)
    $remaining \leftarrow allServices$
    $currentLevel \leftarrow sources$
    while $remaining \neq \emptyset$ do
        $nextLevel \leftarrow \emptyset$
        for all Service $S_i \in currentLevel$ do
            if $S_i$ is uncomposed then
                $C_i \leftarrow$ BottomUpServiceComp($S_i$)                ▷ Algorithm 2.2
                if $S_i$ is uncomposed then
                    $nextLevel \leftarrow nextLevel + S_i$
                    continue
                end if
            end if
            if $S_i$ is composed then
                $nextLevel \leftarrow nextLevel \cup$ OutputNeighbors($S_i$)
            end if
        end for
    end while
end procedure
```

It can be seen that Algorithms 2.2 & 2.4 are very similar, one of the main differences it that the early return uncomposed has been removed, as in top-down, normally the supplying services would not be composed. The second difference is in the definition of cost, for top-down the cost is merely the processing cost plus the sending cost times the edge cost. This is shown in Equation 2.2, where $C_{c,j}$ =: the composition cost of service S_j, $C_{p,j}$ =: the processing cost of S_j, $C_{c,i}$ =:, and $C_{s,i}$ =: sending cost of S_i.

$$C_{c,j} = C_{p,j} + C_{s,j} + c(e_i, j) \tag{2.2}$$

Algorithm 2.2 Bottom-Up Service Composition

```
procedure BottomUpServiceComp(S)
    $composingServices \leftarrow \emptyset$
    $typeSuppliers \leftarrow$ InputNeighbors($S$)
    if $typeSuppliers \ni \hat{S}$ :COST($\hat{S}$) = uncomposed then
        return uncomposed
    end if
    $remaining \leftarrow$ InputTypes(S)
    while $|remaining| \geq 0 \wedge |typeSuppliers| \geq 0$ do
        $S' \leftarrow$ min($typeSuppliers$)  ▷ finds the service with lowest $cost\frac{S}{u}ppliedTypes$
        $typeSuppliers = typeSuppliers - S'$
        for all $T_k \in$ OutputTypes($S'$) do
            for all $T_m \in remaining$ do
                for all $T_n \in$ InputTypes(S') do
                    if $T_m$ canSupply $T_n$ then
                        $composingServices = composingServices + S'$
                        for all $T_l \in$ OutputTypes($S'$) do
                            if $T_l \in remaining$ then
                                $remaining = remaining - T_l$
                            end if
                        end for
                    end if
                    break
                end for
            end for
        end for
    end while
    if $|remaining| \geq 0$ then
        return uncomposed
    end if
    return $\sum_{\forall S_h \in composingServices}$ cost($S_h$)
end procedure
```

Algorithm 2.3 Top-Down

```
procedure TopDown(allServices, sinks)
    $remaining \leftarrow allServices$
    $currentLevel \leftarrow sinks$
    while $remaining \neq \emptyset$ do
        $nextLevel \leftarrow \emptyset$
        for all Service $S_i \in currentLevel$ do
            if $S_i$ is uncomposed then
                $C_i \leftarrow$ TopDownServiceComp($S_i$)            ▷ Algorithm 2.4
                if $S_i$ is uncomposed then
                    $nextLevel \leftarrow nextLevel + S_i$
                    continue
                end if
            end if
            if $S_i$ is composed then
                $nextLevel \leftarrow nextLevel \cup$ InputNeighbors($S_i$)
            end if
        end for
    end while
end procedure
```

Algorithm 2.4 Top-Down Service Composition

```
procedure TopDown(S)
    $composingServices \leftarrow \emptyset$
    $typeSuppliers \leftarrow$ InputNeighbors($S$)
    $remaining \leftarrow$ InputTypes(S)
    while $|remaining| \geq 0 \wedge |typeSuppliers| \geq 0$ do
        $S' \leftarrow$ min($typeSuppliers$)  ▷ finds the service with lowest $cost\frac{S}{u}ppliedTypes$
        $typeSuppliers = typeSuppliers - S'$
        for all $T_k \in$ OutputTypes($S'$) do
            for all $T_m \in remaining$ do
                for all $T_n \in$ InputTypes(S') do
                    if $T_m$ canSupply $T_n$ then
                        $composingServices = composingServices + S'$
                        for all $T_l \in$ OutputTypes($S'$) do
                            $T_l \in remaining \Rightarrow remaining = remaining - T_l$
                        end for
                    end if
                    break
                end for
            end for
        end for
    end while
    if $|remaining| \geq 0$ then
        return uncomposed
    end if
    return $\sum_{\forall S_h \in composingServices}$ cost($S_h$)
end procedure
```

3. SOFTWARE OPERATION

3.1 Installation

The available package contains the code and can be extracted into a directory. Once the simulator is extracted, then the Qt pre-compiler `qmake` should be run on the file `DSCSim.pro`. Following this, the standard GNU-make utility `make` will run the necessary compilation tasks. This procedure is exampled in Figure 3.1.

3.2 Simulation Configuration

The simulator is configured using plain-text input files constructed in a specific format.

3.2.1 General Configuration

The first entries in the configuration file should be the options describing the parameters of the simulations.

Composition Interval: `INTERVAL`

Neighbor Distance: `DISTANCE`

Simulation Type: `DISTRIBUTED` or `CENTRALIZED`

The *Composition Interval* is the length of simulation time between the initiation of a service composition. This time is in abstract simulation units that can represent whatever the user desires. Neighbor Distance is the maximum distance between nodes

Figure 3.1 Example Installation

```
name@host:~/src$ unzip DSCSim.zip
name@host:~/src$ cd DSCDSim
name@host:~/src/DSCDSim$ qmake DSCSim.pro
name@host:~/src/DSCDSim$ make
.
.
.
name@host:~/src/DSCDSim$
```

that are considered neighbors, the units are, once again, abstract. Simulation Type is the a selection of whether a distributed, or a centralized composition will be used. If a centralized composition is desired, the Composition Node will have to be designated. This will be covered further later on. Another configuration setting listed is the `END`, for which the argument is a number specifying the length of time the simulation should run. It should be noted that this is an arbitrary simulation time, not a real time. Figure 3.2 contains an example of these settings in use.

3.2.2 Type Configuration

The next section in the configuration file consists of defining the data types used in the simulation. A type is specified on a line beginning with the identifier `T` followed by the Type Identifier with is followed any Parent Types that may exist. Figure 3.2 is an example of two types, where `Vehicle` is the Parent Type of `Bus`.

Following the types in the configuration file are the service definitions. Each service definition is a line beginning with the identifier `T` followed by a unique type name. This can be optionally followed by a space separated list of type names that inherit from the first type. Examples of the service type definitions can be found is Figure 3.2. Please note that types can either be defined on their own, or can combined into a combined parent-children statement.

3.2.3 Service Configuration

The next section in the configuration file is the Service definitions. The service definition begins with an identifier `S`. This is followed by a unique service name. Following that is a list of service input types that are ended by a `|` symbol. Then there is a list of service output types, also ended by a `|` symbol. After the types and their ending symbols comes the service processing cost, or the cost associated with producing the output from the inputs. The final entry in the service definition is the cost associated with sending the output, i.e. a size of the message to be sent. an example of a service definition can be seen in Figure 3.2. It is also important to note that `|` symbols must be separated from spaces on either side, otherwise the simulator parses it as part of the type name, resulting in an invalid instruction.

3.2.4 Node Configuration

Once the services have been defined, they can be used in the next configuration step, the node definitions. Node definitions consist of, as usual an identifier, `W`, followed by a unique name. This is then followed by a space-delineated list of service names that are contained by this node. Note: if a service is listed for more than one node, it will be considered that there are two separate implementations of this service, one on each node. At the end of the node definition is the nodes location. Listed as a space separated set of Cartesian coordinates. Examples of node definitions can be found in Figure 3.2.

3.2.5 Event Configuration

Once the definitions of all the required components for the simulation are listed, then follow the definitions of events. Events consist of a node either being deactivated or activated. An event definition consists of a identifier, either `D` for deactivate or `E` for activate, followed by a node name, followed by a simulation time. This can be seen in Figure 3.2.

3.3 Use

3.3.1 Load Configuration

To load the configuration into the simulation; type the configuration file name into the text box in the upper left corner Figure 3.3. Start the loading process by pressing the button labeled `Load`. This will read in the configuration file, create the type tree, the graph of the node, and the graph of the services. In the node graph, shown in Figure 3.5, the edge weights displayed is the distance between the two nodes.

The type tree generated for the example can be seen in Figure 3.6, note that the direction of the edges points in the same direction as type satisfaction, meaning: the type at the origination of the edge can be used as a replacement for the type at the receiving edge.

In the service graph, the edge weights displayed is the path distance between the nodes upon which the services reside, multiplied by the sending cost of the service. The computation cost of each service is shown by a number in the vertex. The label

Figure 3.2 Example Configuration

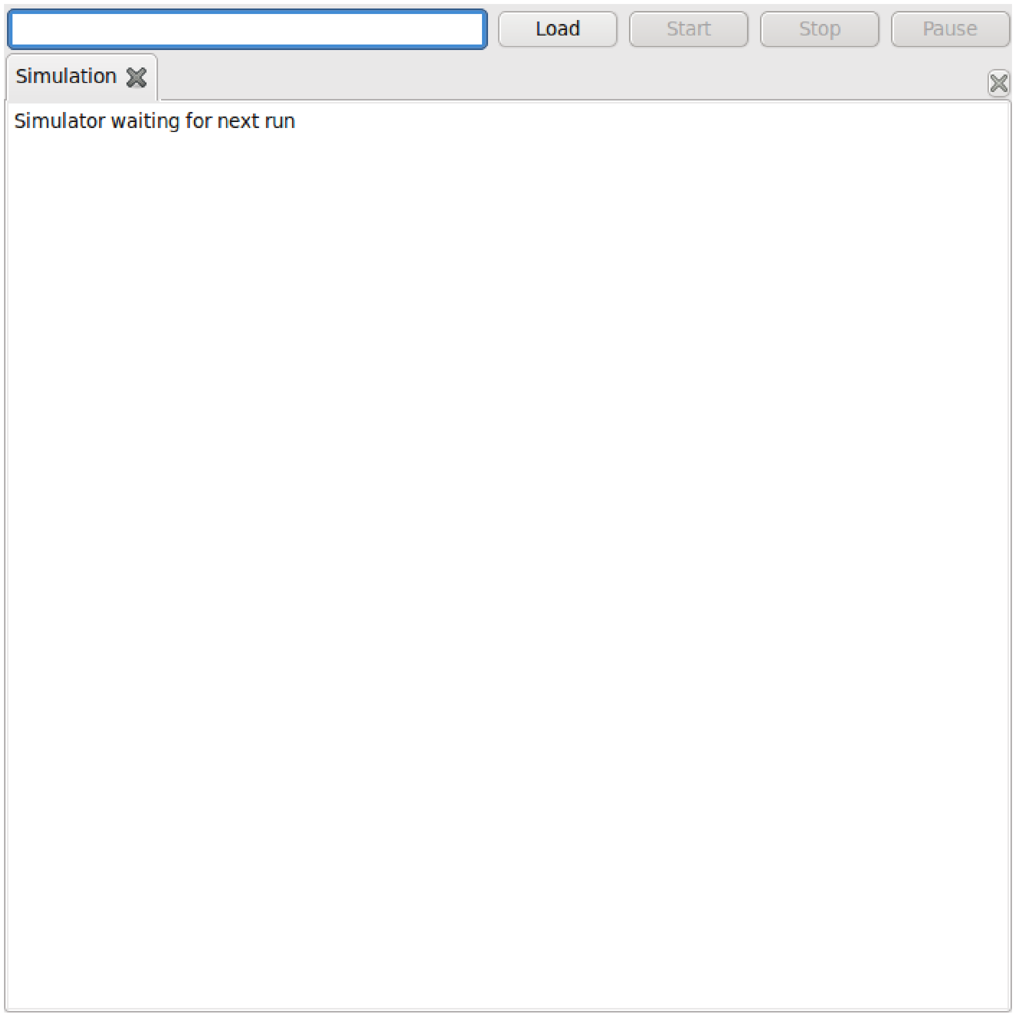
```
INTERVAL 2
DISTANCE 2.5
DISTRIBUTED


T SmallImage
T SmallImage MediumImage
T MediumImage LargeImage
T LargeImage HugeImage
T Vehicles Trucks Cars
T Objects Vehicles Faces

S FaceDetect LargeImage MediumImage | Faces | 4.3 2
S VehicleDetect LargeImage MediumImage SmallImage | Trucks Cars | 3 1
S SmallSupply | SmallImage | 1.1 1
S LargeSupply | LargeImage | 2.1 2
S MediumSupply | MediumImage | 3.2 2.6
S HugeSupply | HugeImage | 2.2 3
S DataCollection Objects | | 1.7 0

N ImageProc FaceDetect VehicleDetect | 1 2
N ImageSupply SmallSupply MediumSupply LargeSupply HugeSupply | 1 1
N DataStorage DataCollection | 1 3

D ImageSupply 2
E ImageSupply 4
D ImageProc 5
E ImageProc 7

END 9
```

shown in each vertex is the unique id of the service. This is made of the id specified in the configuration file, a number, and the id of the node it is part of.

3.3.2 Start Simulation

To start the loaded simulation, press the `Start` button in the top center of the window as shown in 3.4.

Figure 3.3 Simulator Screenshot: Before Configuration Loaded

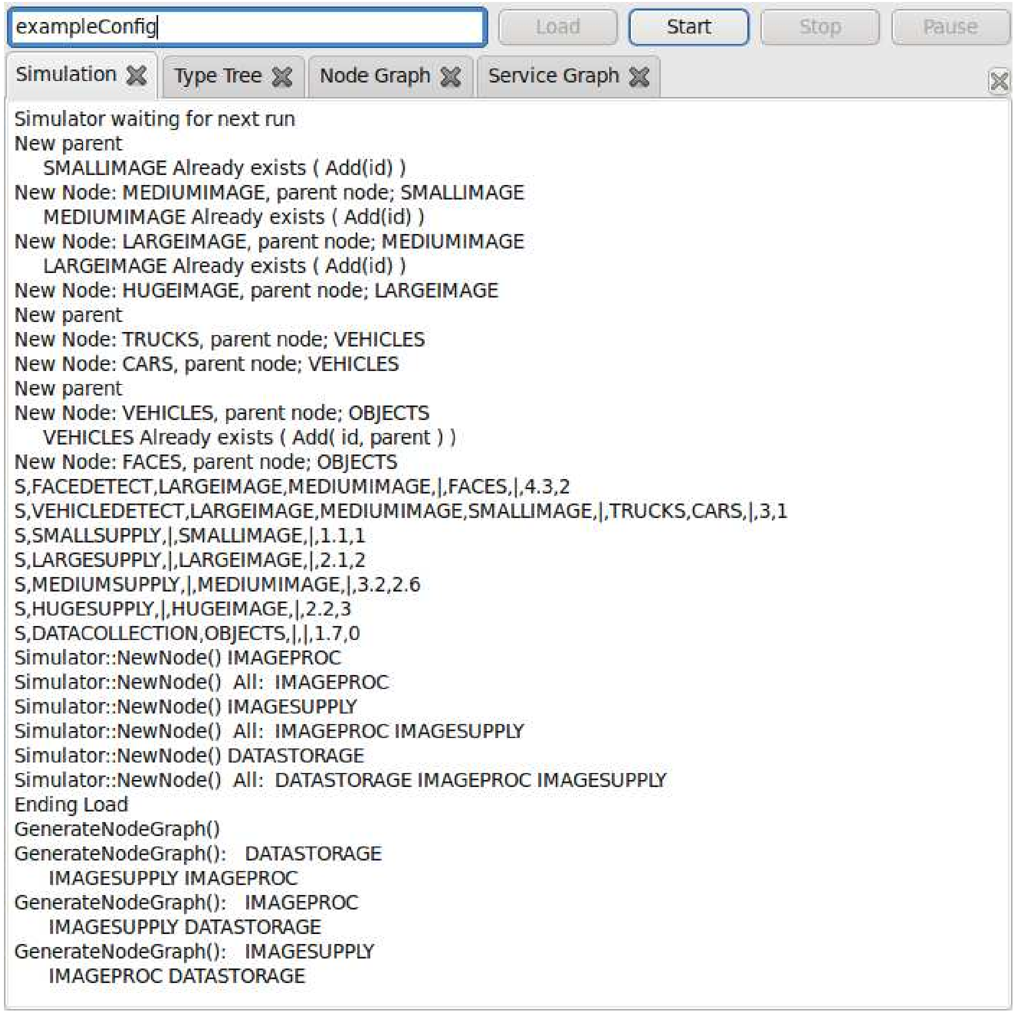


Simulator waiting for next run

3.3.3 Running Simulation

Once the simulation is running, the `Pause` button will temporarily halt the running simulation and the `Start` button will restart it will where it was halted. The `Stop` button will halt the simulation permanently. While the simulation is running, a new tab will be created showing the service composition for every new composition.

Figure 3.4 Simulator Screenshot: After Configuration Loaded



```
Simulator waiting for next run
New parent
    SMALLIMAGE Already exists ( Add(id) )
New Node: MEDIUMIMAGE, parent node; SMALLIMAGE
    MEDIUMIMAGE Already exists ( Add(id) )
New Node: LARGEIMAGE, parent node; MEDIUMIMAGE
    LARGEIMAGE Already exists ( Add(id) )
New Node: HUGEIMAGE, parent node; LARGEIMAGE
New parent
New Node: TRUCKS, parent node; VEHICLES
New Node: CARS, parent node; VEHICLES
New parent
New Node: VEHICLES, parent node; OBJECTS
    VEHICLES Already exists ( Add( id, parent ) )
New Node: FACES, parent node; OBJECTS
S,FACEDETECT,LARGEIMAGE,MEDIUMIMAGE,|,FACES,|,4.3,2
S,VEHICLEDETECT,LARGEIMAGE,MEDIUMIMAGE,SMALLIMAGE,|,TRUCKS,CARS,|,3,1
S,SMALLSUPPLY,|,SMALLIMAGE,|,1.1,1
S,LARGESUPPLY,|,LARGEIMAGE,|,2.1,2
S,MEDIUMSUPPLY,|,MEDIUMIMAGE,|,3.2,2.6
S,HUGESUPPLY,|,HUGEIMAGE,|,2.2,3
S,DATACOLLECTION,OBJECTS,|,|,1.7,0
Simulator::NewNode() IMAGEPROC
Simulator::NewNode()  All:  IMAGEPROC
Simulator::NewNode() IMAGESUPPLY
Simulator::NewNode()  All:  IMAGEPROC IMAGESUPPLY
Simulator::NewNode() DATASTORAGE
Simulator::NewNode()  All:  DATASTORAGE IMAGEPROC IMAGESUPPLY
Ending Load
GenerateNodeGraph()
GenerateNodeGraph():   DATASTORAGE
     IMAGESUPPLY IMAGEPROC
GenerateNodeGraph():   IMAGEPROC
     IMAGESUPPLY DATASTORAGE
GenerateNodeGraph():   IMAGESUPPLY
     IMAGEPROC DATASTORAGE
```

Figure 3.5 Simulator Screenshot: Displaying Node Graph

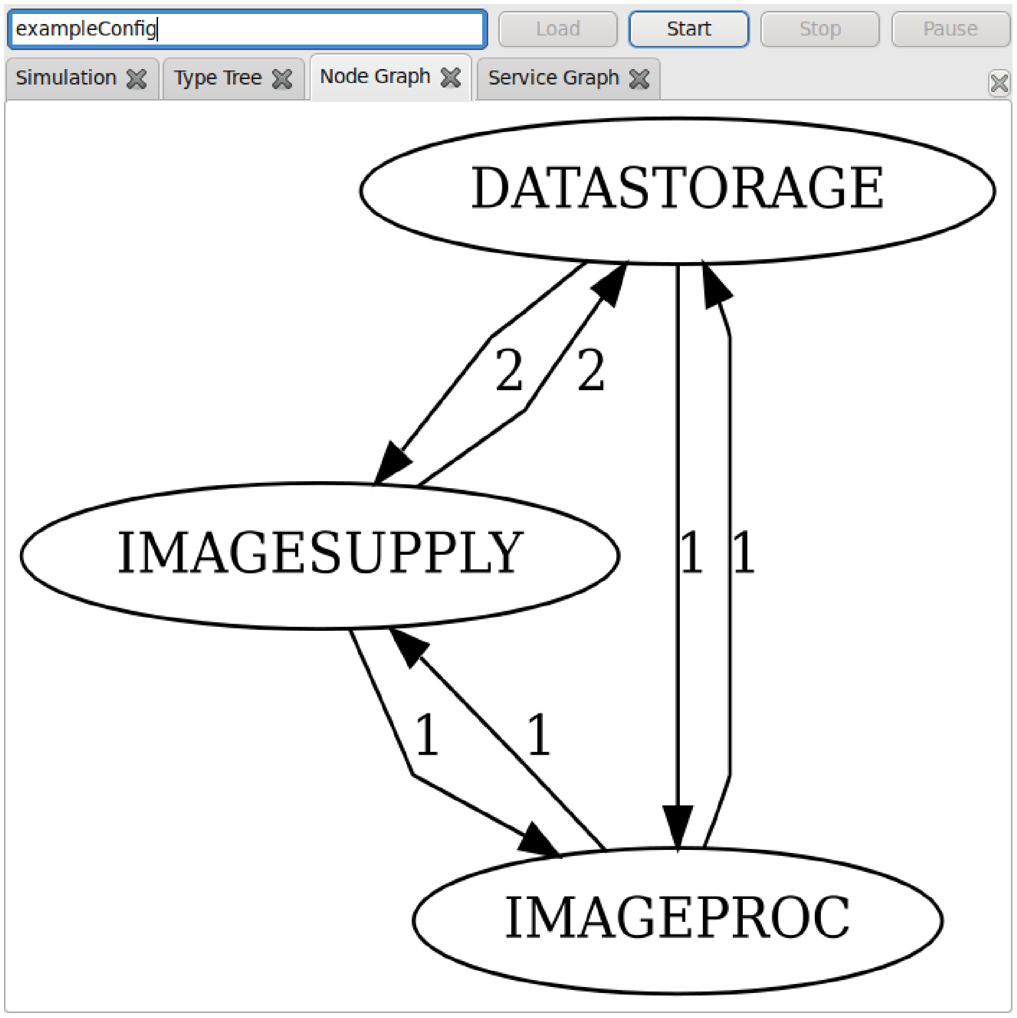


Figure 3.6 Simulator Screenshot: Displaying Type Tree

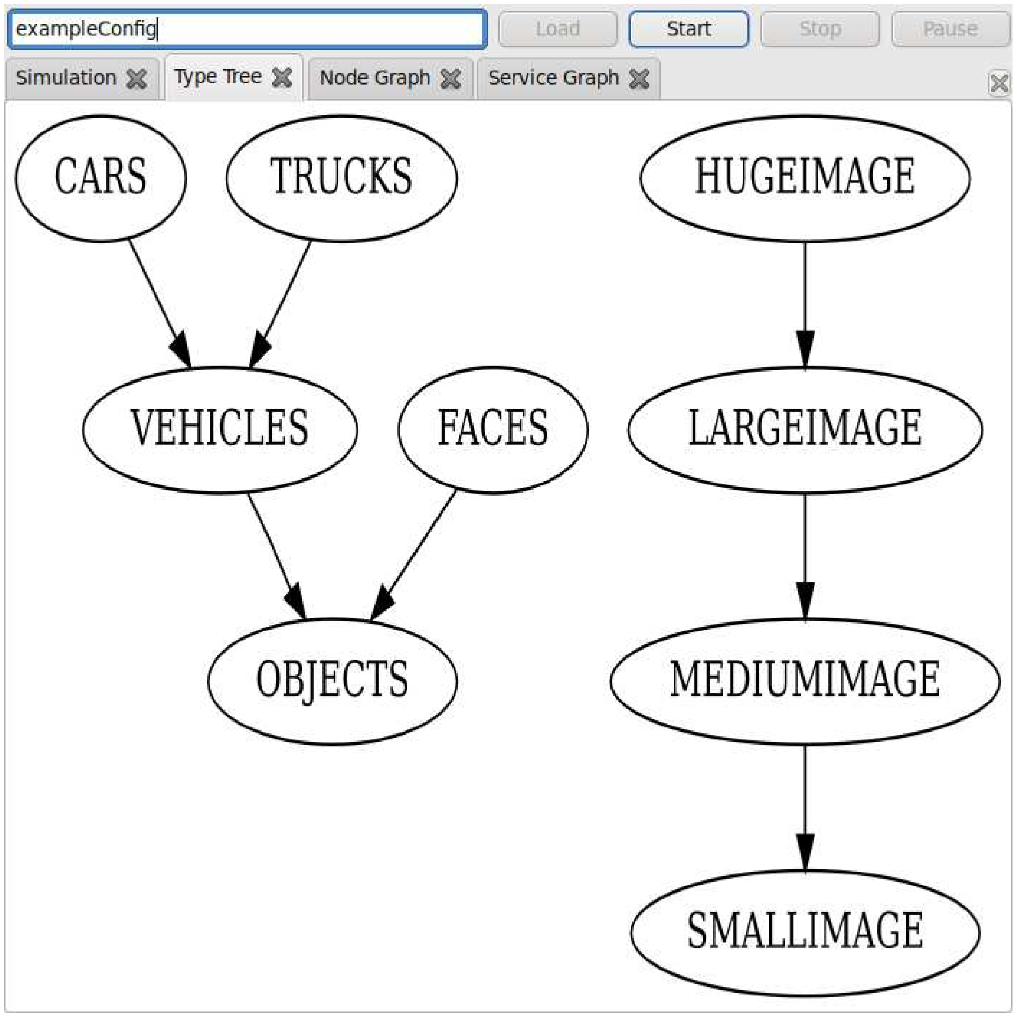
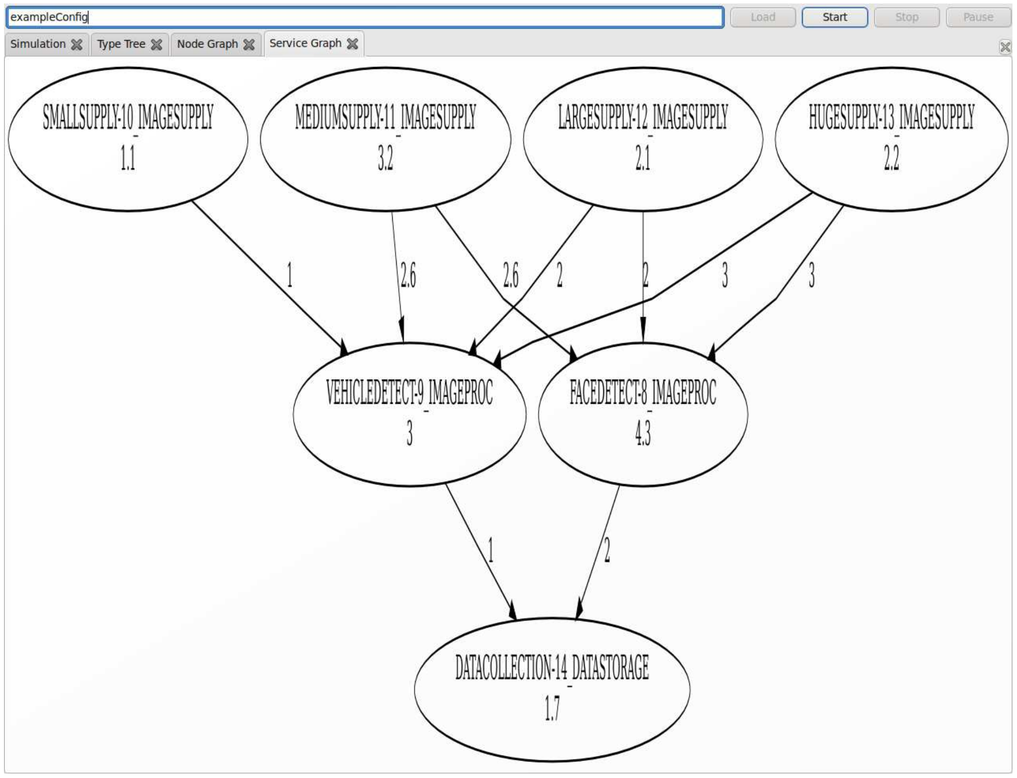


Figure 3.7 Simulator Screenshot: Displaying Service Graph


exampleConfig
Load
Start
Stop
Pause
Simulation
Type Tree
Node Graph
Service Graph
SMALLSUPPLY-10_IMAGESUPPLY
1.1
MEDIUMSUPPLY-11_IMAGESUPPLY
3.2
LARGESUPPLY-12_IMAGESUPPLY
2.1
HUGESUPPLY-13_IMAGESUPPLY
2.2
1
2.6
2.6
2
2
3
3
VEHICLEDETECT-9_IMAGEPROC
3
FACEDETECT-8_IMAGEPROC
4.3
1
2
DATACOLLECTION-14_DATASTORAGE
1.7

4. SALSA VARIATION

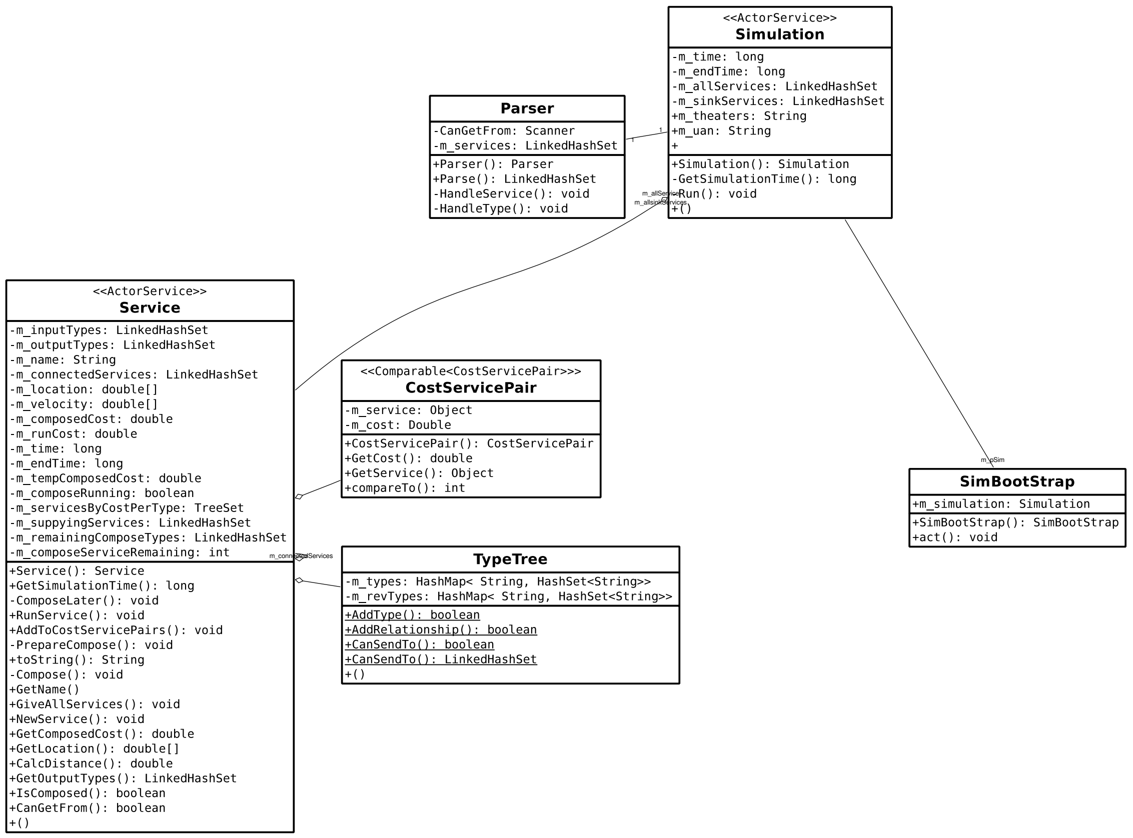
4.1 Idea

A variation of the original service composition simulation was created to attempt to utilize the concurrent nature of the distributed composition implementation. This simulation was written in SALSA, an extension upon Java that implements an actor model programming language [6]. The actor model suits modeling of services in a dynamic sensor network. This is due to all the communication between actors in the actor model relying on message passing, which is quite natural for a services since they would normally be separate entities passing messages over a network.

Further difference lies in what was being simulated. While the majority of this research simulates services on nodes that can be enabled or disabled with static locations, in the SALSA simulation each service had static availability and dynamic location. The world modeled by the locations in the SALSA simulation further differed from the C++ simulation. While the C++ simulation used 2-dimensional space, the space in the SALSA simulation is 3-dimensional. The three dimensions are specified by spherical coordinates in a vector consisting of a radial distance, an inclination angle, and an azimuth angle. The change in location was modeled assuming a constant velocity specified at service creation. This velocity is given as a vector specifying a radial velocity and two angular velocities, allowing the simulation to roughly model an approximation of satellites in orbit around a body. Services are considered to be connectible if they were within a specific given Cartesian distance and one can supply a type required by the other.

Another notable difference between this simulation and the C++ simulation is that, while in the previous simulation the services are located on a node at a location; in this version the services are not attached to a node and they occupy a position in space directly.

Figure 4.1 SALSA Variation Class Diagram



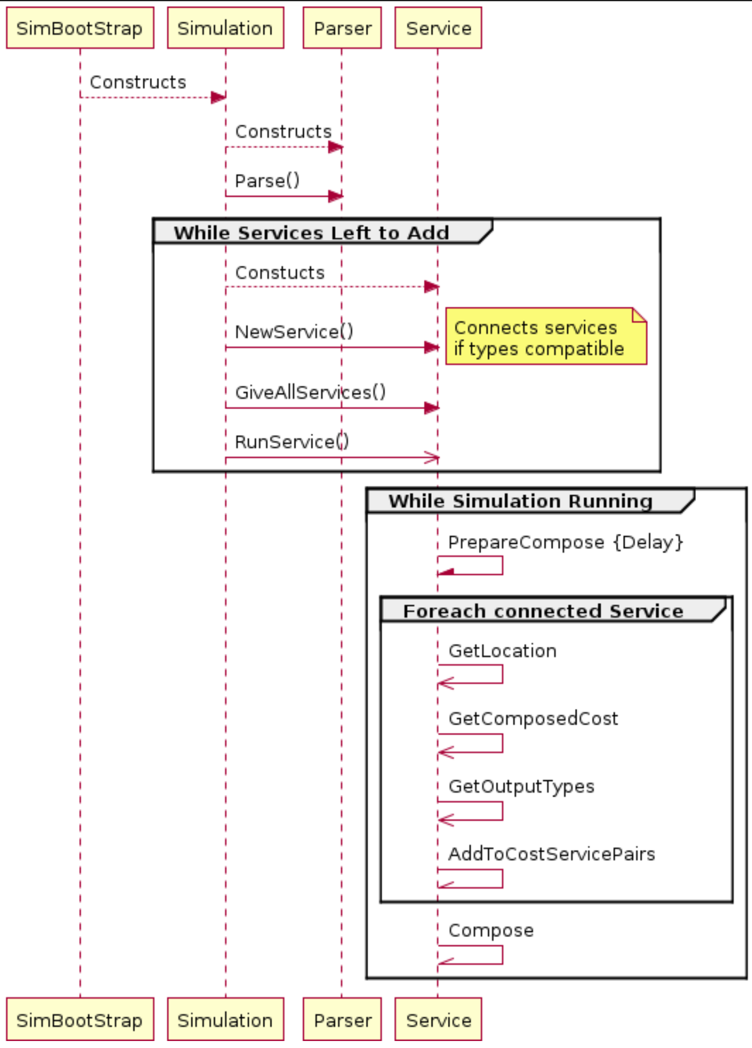
4.2 Design

The design of the SALSA simulator is very much like that of the C++ simulator, only simplified. The running of the simulation is controlled by a class `Simulation`. Unlike the C++ version, no event queue must be maintained, since all composition is based on the position of the services rather than their respective enabled status. Note: most of the use of the event queue for the C++ simulation was to enable and disable nodes at specific times.

The class `Parser` is the SALSA simulation's equivalent of the `Interface` class. It interprets the text configuration file from the format described in Section 4.3.1. From this, it generates a type tree of the same style as the C++ version's and it generates services with an initial position and an angular velocity.

After each service is constructed, it is connected to all other services that have compatible types and then passed an asynchronous message `RunService` to start it running its composition updates. `RunService` chooses a random time uniformly

Figure 4.2 SALSA Variation Messages Passed



distributed between 0 and 500 and sends an asynchronous `PrepareCompose` message to itself with a after that delay, reducing the chance of all services composing at the same time and competing for communication resources. The composition algorithm itself works in much the same way as the C++ version's bottom-up composition, as described in Section 2.4.1.1

4.3 Operation

4.3.1 Configuration

It can be seen from Figure 4.3 that the configuration file for the SALSA simulation is different from the C++ simulation's configuration. The primary operational difference is the lack of general simulation parameters such as the composition interval, neighbor distance, and simulation length. These parameters are compiled into the code due to the simplicity of the simulation. Another difference easily spotted is the abandonment of the single-line approach of the C++ simulation's configuration file. This was done as an experiment to improve upon readability.

4.3.1.1 Type Configuration

The Type configuration is begun with an identifier `Type` on it's own line. The first line after the identifier should contain the Type's ID. The following line should contain a list of all Types that this Type can supply, this list should be whitespace delimited and followed by whitespace and a `#` symbol.

4.3.1.2 Service Configuration

The Service configuration begins with the identifier `Service`, followed by the service ID on the next line. The line following contains a whitespace delimited list of all output Types of this Service. This is followed by whitespace and a `#` symbol. The next line is a whitespace delimited list of all input Types of this Service, followed by whitespace and a `#` symbol. The next line is a space delimited list of the three initial coordinates, followed by a line with the space delimited velocities, as described in Section 4.1.

4.3.2 Compilation

The compilation of SALSA code is described in [6], Section 3.6.3. This simulation's code can be compiled using GNU Make and the included Makefile, *i.e.* `make`.

4.3.3 Running

As this simulation does not properly utilize the distributed computing aspects of SALSA, the process for running SALSA code is described in [6] Section 3.6.3. This simulation is run using the `SimBootStrap` class, and the configuration file name as an argument.

Figure 4.3 Example SALSA Variation Configuration

```
Type
  LowQuality
  #
Type
  MedQuality
  LowQuality #
Type
  HighQuality
  MedQuality #
Service
  HighCamera
  HighQuality #
  #
  1 1 1
  0 0 1
Service
  LowRecorder
  #
  LowQuality #
  1 1 1
  0 0 2
```

5. DISCUSSIONS AND CONCLUSION

5.1 C++ Simulator

5.1.1 Performance

The C++ simulator can be very slow for simulations of a thousand or more services. This varies greatly depending on activation rates of nodes and the number of types each service has for inputs and outputs. Profiling has shown that a majority of the time consumed by the service composition is spent in the function `Service::CanRecvFrom()`. This function has to cycle trough every type supplied and every type received by each Service, to determine if there are any compatible Types in the type tree. This composition time was found be reduced by a few percent by changing the method of storing each service's types from a `std::set` to a `std::vector`, and no perceptible increase was found in the creation or other accesses of the types. This shows that if additional speed-up is desired, the types of other "lists" may want to be investigated.

Initially the simulator used Pthreads to allow the services to compose concurrently, but this was abandoned due to seeming to be unnecessary and for portability issues. Larger simulations have shown that this speed-up may have proven to be useful, assuming the overhead of that many threads would be manageable. It would be recommended to investigate a concurrent version of this simulator. However Qt's `QThread` would be advised for portability.

5.1.2 Capabilities

One difficulty that can be encountered with the GUI is that if larger simulations are desired, Graphviz may have trouble displaying a neat and clear graph of the services, nodes, and composition. It has been noticed, that particularly with the service graph, Graphviz tends to produce a result that starts expanding rapidly horizontally, but not vertically, making the text illegible.

There are many other visualization softwares available, *e.g.* VTK [7] and [8] (though it should be noted that [8] also utilizes the DOT language), if large complex simulations are desired, switching to one of those would be recommended. Investigating further visualization options at the planning stages is also highly recommended.

5.1.3 Desired Enhancements

A lacking feature that would prove useful would be some built-in method of saving the generated graphs without having to take a screenshot. This feature was left out due to time constraints and priorities.

5.2 SALSA Simulator

The SALSA simulation was never intended to act as a comparison to the C++ version, hence the great discrepancies between capabilities and functionality. Therefore to compare it against the C++ simulator would be unjust to its design and function.

The current implementation of the SALSA simulation preformed its task well with small numbers of services (*i.e.* a couple dozen). However even at these numbers, the Java process would not utilize greater than a single CPU core. Given the overhead inherent in implementing an actor model on top of Java, it can be assumed that this would preform better if it ran with multiple cores.

As a further hindrance to the SALSA simulation's viability, when scaled up to larger numbers of services (in this case: more than a couple dozen), it would slow down and cease to function. It has not yet been determined whether this is due to deadlock or some other problem, as it does not occur with low enough service numbers to practically trace the simulation's operation.

LITERATURE CITED

[1] S. C. Geyik, B. K. Szymanski, P. Zerfos, and D. Verma, "Dynamic Composition of Services in Sensor Networks," *Services Computing, IEEE International Conference*, pp. 242–249, 2010.

[2] SOA CoE Core Team 2008/2009, "Service Oriented Architecture (SOA) Governance Model," tech. rep., State of California Franchise Tax Board, May 2010. Version 1.4.

[3] "ISO/IEC 14882." ISO/IEC, October 2003. Programming Languages – C++.

[4] ISO/IEC JTC1/SC22/WG21 - C++ Standards Committee, "Working Draft, Standard for Programming Language C++." N3242=11-0012, February 2011.

[5] "Algorithmic Performance Comparison Between C, C++, Java and C# Programming Languages," tech. rep., Cherrystone Software Labs, August 2010. Version 1.6.02.

[6] C. A. Varela, G. Agha, W.-J. Wang, T. Desell, K. E. Maghraoui, J. LaPorte, and A. Stephens, *The SALSA Programming Language 1.1.2 Release Tutorial*. Rensselaer Polytechnic Institute, 110 8th St Troy, NY, February 2007.

[7] W. Schroeder, L. Avila, and W. Hoffman, "Visualizing with VTK: a tutorial," *Computer Graphics and Applications, IEEE*, vol. 20, pp. 20 –27, Sep/Oct 2000.

[8] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.