Service Composition Implementation Details

Composition Request

The concept of service composition request used in this implementation has been directly borrowed from <Touraj's thesis/paper title/citation>. It is defined as a quadruple R = <I, O, QoS, C>. From the implementation perspective:

- The inputs, outputs and QoS features requested by the user are represented as lists of String objects whereas the constraints are represented as a list of Constraint objects. The Constraint class is defined in the service repository implementation.

- Each of the requested inputs and outputs is a String consisting of two parts: data type and name. The two parts are separated by a colon preceded and succeeded by a space character. E.g., "string : Student Name", "int : NumberOfCourses", etc.

- The data types currently handled by the implementation are int, float, char, boolean and string.

- Input/output names are allowed to have spaces. E.g., both "string : Student Name" and "string : StudentName" are valid parameters.

- A user-requested QoS feature consists of just the feature name, which must be one of the four acceptable values: COST, RESPONSE_TIME, RELIABILITY and AVAILABILITY.

- Comparison of input, output and QoS parameters during validation consider the case of the parameter, i.e., the comparison is case-sensitive. Therefore, "string : StudentName" is considered to be different from "String : StudentName" and "string : studentname". Also, while "COST" is an acceptable QoS feature, "Cost" and "cost" are not.

- A user-requested constraint is composed of exactly three elements in the given sequence: feature, operator and literal value. The three elements are separated by a pipe symbol preceded and succeeded by a space character. E.g., "float : Price | < | 100.0", "boolean : InvoiceRequired | = | true", etc.

- The feature (also called type) in a user constraint must be a parameter from the requested inputs, outputs or QoS features. The feature must follow the same format and naming convention as defined for the parameters. E.g., while "COST | < | 10" is a valid user constraint, "cost | < | 10" is not. Similarly, while "float : Price | < | 100.0" is a valid user constraint, "Price | < | 100.0" is not.

- The operator in a user constraint must be one of the following acceptable operators: <, >, =, <= and >=.

A sample composition request for a composite service that accepts a student's unique ID as an input and returns his marks percentage and GPA as output would appear as follows:

- **Inputs:** {int : StudentID}
- **Outputs:** {float : MarksPercentage, float : GPA}
- **QoS:** {RESPONSE_TIME}
- **Constraints:** {RESPONSE_TIME < 5, StudentID > 0}

Validation Checks

The following validation checks are performed on the service composition request submitted by the user. If any of these checks fail, the service composition process is aborted immediately.

- A composition request constraint must have exactly 3 elements (separated by the pipe symbol) in the sequence: type, operator and literal value. E.g. "float : Price | < | 100.0".
- Operators that can be used in a constraint currently include <, >, =, <= and >=. These are validated against the Operator enumeration defined in the service repository implementation.
- A composition request must provide at least 1 input.
- A composition request must request at least 1 output.
- Quality of Service features that can be used in a composition request currently include COST, RESPONSE_TIME, RELIABILITY and AVAILABILITY. These are validated against the QualityOfService enumeration defined in this service composition implementation.
- A composition request constraint must be applied only on (i.e., should have as type) the parameters provided as input, output or QoS in the request.

Below are some additional validation checks performed once a valid composition request is created. If any of these checks fail, the service composition process also fails.

- The service repository must contain at least 1 service.
- The composition problem should be solvable based on the given request, repository and algorithms, i.e. at least 1 solution should be obtained at each step of the composition process.
- The composition problem should not be solvable by a single service present in the given service repository, thereby rendering service composition unnecessary.

Differences between Algorithm and Implementation

- In the algorithm, a composition request and a set of available services are provided as input. However, in the implementation, a composition request configuration object and a logger object are provided as arguments.
  Reason: The configuration object contains the inputs, outputs, QoS features and constraints requested by the user which are used to create a composition request object before triggering the various composition phases. It also contains the location of a service repository file which can be parsed to extract the available services. Besides these, the configuration also contains a flag,

which, when set to “Y”, causes the composition plans created as solutions to the given request to be stored as composite services in the given repository. This is an additional feature of the implementation.
To accommodate the additional information required and to be able to fetch the required input from the user through the console or a file, creation of the configuration object was necessary.
The logger object provides another additional functionality of recording error messages in a text file for reference.

- The algorithm did not include any specific validation checks. However, in the implementation, several checks are performed on the composition request, service repository and results of the various stages of the service composition process.
  Reason: These validation checks ensure that the composition process continues after completing each step only if all the execution parameters are available and valid and if it is worth triggering the next step so as to minimize the effort involved in case of a failure.

- The loops around certain algorithms that are triggered by the service composition algorithm are not implemented. Instead they are included in the respective triggered algorithms themselves.
  Reason: This has been done for better modularity and lower coupling. Since the complete implementation of the subsequent algorithms is now contained within their own classes, any future modifications in those processes (if required) would not affect the service composition implementation.

Notes

- As per the Constraint class defined in the service repository implementation, each constraint object must have the service name data member populated. However, the constraints provided in a composition request are not associated with any individual service. Therefore, in the implementation, a dummy service name “CompositeService” is associated with the requested constraints.
  Since, user constraints are not used in the composition process as of now, the dummy name does not affect the implementation. However, once they are included in the process, this dummy name would be replaced by the dynamically generated composite service name.

- The input and output names for any service (atomic or composite) have 2 parts separated by a colon preceded and succeeded by a space character: data type and name. E.g., “float : Price”, “string : ProductName”, etc.

- Since there is no processing being done on the QoS features as of now, no data types have been associated with them.

Additional Features

- **Logger:** A simple message logging utility has been added to the implementation which allows all the error and status messages generated during the service composition process to be recorded in a text file.
  Each logger object is associated with a specific text file, and the file is opened in "append" mode. Therefore, if the same logger object is passed across the different methods called during the service composition process, all the messages generated can be recorded in the same text file. This helps in generating a persistent error record of a composition run and also assists in performing automated unit testing (since it eliminates the use of the console).

- **Service Parser Alternatives:** Depending on the type (file extension) of the service repository file whose details are provided in the request configuration by the user, a suitable constrained service parser can be employed. Currently, a serialized Java object parser is used to parse .txt files and an XML parser is used to parse .xml files. This functionality can be extended to include as many repository file types as there are parsers available, thereby making the composition process more versatile.

- **Composite Service Storage and Reuse:** A layered composite service decorator has been added to the service repository implementation. A utility class has also been defined which uses the decorator to create a layered composite service object for each of the constraint-aware composition plans created as a solution to a composition request. This utility also enables storing this composite service object back in the service repository from which the atomic services for its creation were extracted. The stored composite service can then be used as a component service for future compositions.
  The limitation to this feature is that, at present, the storage and reuse of composite services is restricted to serialized Java object repositories only.

Future Work/Limitations

- Although the user is allowed to specify constraints as part of a composition request, the current service composition model does not support user constraints nor are there are any mechanisms implemented as of now that would include the user constraints within a composition plan.
  The forward expansion algorithm does include a "CheckUserConstraints" statement that should trigger verification of user constraints, however, currently, it acts only as a placeholder for a more elaborate user constraint verification solution, which is presently out of scope of this thesis. However, since we plan to support user constraints in future, we accept them as part of a composition request.

- Similarly, although QoS features are accepted as part of a composition request, they are neither used in the service composition model nor the implementation presently. However, they are planned to be included in future works.

- Currently, the operators that can be used in constraints include only <, >, =, <= and >=. More operators can be added to the Operator enumeration and means to validate and evaluate them can be implemented.

- Currently, the QoS features that can be used in composition requests include only COST, RESPONSE_TIME, RELIABILITY and AVAILABILITY. More features can be added to the QualityOfService enumeration and means to validate and process them can be implemented.

- Currently, int, char, float, string and boolean data types are being handled for input and output parameters. The functionality can be extended in future to handle more data types. Also, if QoS features are processed in future, data types would need to be associated with each of them.

- Currently, a parameter is represented as a String with two parts separated by a colon: data type and name. This makes the parsing of parameters untidy. To make the process cleaner, a new data structure, such as a Parameter class with datatype and name data members, could be defined to better represent and process service inputs, outputs and effects.

- The storage and reuse of layered composite services is, at present, restricted to serialized Java object repositories. This functionality could be extended to other repository formats, such as, XML, JSON, WSDL, etc., by designing the proper representations and developing the parsers and writers for them.