Service Composition Implementation Additional Features

To make the service composition implementation more flexible and to enable its use as a tool/application, some additional functionality has been introduced in it that is not found in the algorithms on which the implementation is based. Here, those additional features and their architecture have been described.

Service Repository 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 within the service composition process to read constrained service repositories. 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 easily extended to include as many repository file types as there are parsers available, thereby making the composition process more versatile.

Service Composition Driver

The service composition driver is responsible for prompting the user to provide the information required for the service composition process, to trigger the various phases involved in the process in the proper sequence and to display the final status (success/failure) of the process on the console. Specifically, the driver performs the following tasks:

- Prompt the user on the console to select a mode of input for providing the composition request configuration. At present, the user can choose between console and XML file modes, although the architecture in place allows these options to be extended to other modes as well.

- If the XML file mode is selected, prompt the user to provide the XML configuration file name and location.

- Depending on the selected mode of input, invoke the corresponding request configuration reader to receive the composition request and other necessary information from the user.

- Create a logger object which would be passed across methods for recording error messages generated throughout the process.

- Trigger the service composition process, passing it the request configuration and logger objects created earlier.

- If the composition process fails at any point, display a failure message and invite the user to check the log file for error details.

- If the composition process is successful, write the constraint-aware composition plans generated into a text file and invite the user to check the plans file.

- The log and plans file, if generated, are stored in the same folder that contains the service repository offered by the user. Sample log (log.txt) and plans (plans.txt) files have been placed in the documents folder along with this document.

Plans File Description:

- Multiple constraint-aware composition plans can be generated for a composition request. All the generated plans will be numbered and listed in the file.

- Each plan consists of layers, with their indices starting from 0. A plan description lists all the layer descriptions from top to bottom in increasing order of their layer index. A separate line is allotted to each layer.

- Each layer description describes the service nodes that constitute that layer. Multiple node descriptions are separated by a comma. The order of service nodes (or their description) within a layer is not important.

- A service node description consists of four parts: predecessor names, constraints, service name and successor names. They are written as follows:
  {predecessor_names} [constraints] service_name {successor_names}

- Predecessor names for a service node are a comma-separated list of service names that belong to the service nodes that act as predecessors to the given service node. Predecessors are the nodes that must be executed before the given node can be executed. The order of names in the list is not important.

- Constraints for a service node are a comma-separated list of constraints that belong to the given service node and must be satisfied before the service contained by the node could be executed. These may include the constraints that belong to the service contained by the given node or the constraints that were transferred to the node from other nodes in the plan during the constraint adjustment process. The order of constraints in the list is not important.
  Each constraint is described as “feature_datatype : feature_name operator_name literal_value”, where:
  - feature_datatype can be one of int, float, char, boolean and string.
  - feature_name is the name of an input or output parameter.
  - operator_name can be one of the following depending on the intended operator:
    - **< :** LESS_THAN
    - **> :** GREATER_THAN
    - **= :** EQUALS

        - ❖ **<= :** LESS_THAN_OR_EQUAL_TO
        - ❖ **>= :** GREATER_THAN_OR_EQUAL_TO
    - literal_value is the value to which the feature's value will be compared during constraint verification.

- Service name is the name of the service contained by the given node.

- Successor names for a service node are a comma-separated list of service names that belong to the service nodes that act as successors to the given service node. Successors are the nodes to which the given node acts as a predecessor. The order of names in the list is not important.

- Service nodes in the first layer do not have any predecessors while those in the last layer do not have any successors. In such cases, the curly braces that enclose the predecessor/successor list are left empty ({}). Similarly, service nodes that do not have any constraints have the enclosing square brackets empty ([]).

- Since every service node must have a service, and every service must have a name, therefore, a service node description must always have a service name.

Composition Request Configuration and its Readers

- **Request Configuration**

  A Request Configuration object contains all the information provided by the user that is required to execute the service composition process. It consists of the following data members:

    - **inputs:** Comma-separated list of requested inputs
    - **outputs:** Comma-separated list of requested outputs
    - **qos:** Comma-separated list of requested QoS features
    - **constraints:** Comma-separated list of requested constraints
    - **repoFileName:** Complete name and path of the file containing the available services
    - **storeCSFlag:** Single-character flag, which, when set to "Y", causes the constraint-aware composition plans created as solutions to the given request to be translated into layered composite services and stored back in the given service repository file. When set to "N", this flag prevents the composition solutions to be added to the repository. Presently, this flag works only for serialized Java object repositories.

  Once created and passed to the service composition trigger, the inputs, outputs, qos and constraints members of a Request Configuration object are used to create a Composition Request object while the repoFileName is used to parse and extract a list of services available for composition. These objects are then used for performing composition of services. Finally, if any solutions are found to the given composition problem, value of the storeCSFlag is inspected to decide whether or not to store the solutions in the repository.

- **Request Configuration Readers**

  The user can opt for different modes for supplying the composition request configuration details. An architecture has been designed and implemented to allow modular addition and removal of such readers. Presently, this implementation can support console and XML file readers. The architecture and implementation specifications are as follows:

  - **RequestConfigReader:** The interface to be implemented by all concrete composition request configuration readers. It declares the readReqConfig method which should be defined to accept request configuration details from the user based on the mode of input being handled by each concrete reader.

  - **ConsoleReqConfigReader:** Concrete composition request configuration reader for interacting with the user through the console to obtain request configuration details. It implements the RequestConfigReader interface and defines the readReqConfig method. After reading all the required information, the method creates a Request Configuration object to be used for service composition.

  - **FileReqConfigReader:** Abstract class to be extended by all concrete composition request configuration file readers. It implements the RequestConfigReader interface but does not define the readReqConfig method; it is expected to be done by the concrete file readers. This class contains a configFileName data member and a mutator method for assigning a value to it. The configFileName member is inherited by all concrete file readers and stores the complete name and location of the file to be read by them.

  - **XMLFileReqConfigReader:** Concrete composition request configuration reader for extracting request configuration details from a user-specified XML file. It extends the FileReqConfigReader class and defines the readReqConfig method. After reading all the required information, the method creates a Request Configuration object to be used for service composition.

  Readers for other file formats can be easily added to the existing architecture by extending the FileReqConfigReader class and defining the readReqConfig method to do the file-specific parsing. To include readers for other modes of input, such as, databases, a new class can be added to this structure and made to implement the RequestConfigReader interface while defining the required behavior in the readReqConfig method.

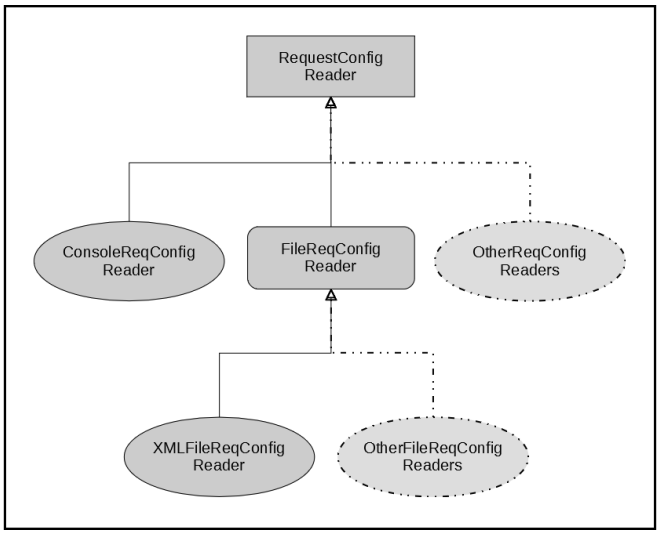

Figure 1: Composition request configuration reader architecture

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 **appending** this composite service object to 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.

Elements of the composite service object created by this utility are listed below:

- **Composite Service Name:** "CompSvc_" concatenated with the current system expressed in nanoseconds.
- **Composite Service Inputs:** Same as the composition request inputs.
- **Composite Service Outputs:** Same as the composition request outputs.
- **Composite Service Effects:** Set of all the effects of all the services contained within the service nodes that constitute the constraint-aware plan.
- **Composite Service Constraints:** Set of all the constraints of the service nodes that constitute the constraint-aware plan.
- **Constraint-aware Plan:** The constraint-aware composition plan for which this composite service is being created. It is one of the resultant plans generated by the service composition process for the given composition request.

The constrained service decorator from the service repository implementation is first used to create a simple constrained service using the composite service name, inputs, outputs, effects and constraints.

Then, this constrained service and the constraint-aware plan are combined by the newly-created layered composite service decorator to form a layered composite service object.

Logger

A simple message logging utility has been added to the implementation, which is shared by the service composition and translation processes. It allows all the error and status messages generated during a 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 a process, all the messages generated can be recorded in the same text file. This helps in generating a persistent error record of a composition/translation run and also assists in performing automated unit testing (since it eliminates the use of the console).