Service Translation Implementation Additional Features

Service Translation Driver

The service translation driver, similar to the composition driver, is responsible for prompting the user to provide the information required for the service translation 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 composite service 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 composite service configuration reader to receive the composite service, target language 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.
- Depending on the target language indicated by the user in the given configuration, trigger the translation process, passing it the configuration and logger objects created earlier. At present, the user can choose between Lucid (more specifically, Objective Lucid) and XML languages, although the architecture in place allows these options to be extended to other target languages as well.
- If the translation process fails at any point, display a failure message and invite the user to check the log file for error details.
- If the translation process is successful, invite the user to check the file to which the translation has been written.
- The log file, if generated, is stored in "*testoutput/servicetranslationruns*" folder under the parent directory. The file with the translation, on the other hand, is stored in the same folder that contains the service repository offered by the user. Sample log (log.txt) and translation (CSLucid_CompSvc_965847426844964.ipl and CSXML_CompSvc_123.xml) files have been placed in the same folder as this document.

Composite Service (CS) and its Readers

- **Composite Service**

  The layered composite service objects parsed and created by these readers are instances of the same class whose objects have been described in the "Layered Composite Service Storage and Reuse" section in the "WSC Additional Features" document. They are composed of the following elements:

    - **Composite Service Name:** Name of the target composite service.

    - **Composite Service Inputs:** Inputs required by the composite service to complete its processing.

    - **Composite Service Outputs:** Outputs generated by the composite service.

    - **Composite Service Effects:** Ideally, in accordance with the service composition model that we follow, the set of all the effects of all the services that together constitute the composite service. However, it is an optional field and no validation is performed to enforce this definition. Also, the current implementation does not make a significant use of service effects.

    - **Composite Service Constraints:** Ideally, in accordance with the service composition model that we follow, it is a union of two sets of constraints: the set of all the constraints of all the service nodes that together constitute the composition plan of the composite service and the set of user constraints imposed on the composite service. However, it is, again, an optional field and no validation is performed to enforce this definition.
      Note: Having all these constraints in the composite service definition is useful when the composite service is used as a component for another larger constraint-aware composition (as per the underlying composition model). In that case, all the known constraints associated with a service could be adjusted optimally across the constraint-aware plan being created, thereby, reducing the number of rollbacks required during the execution of that plan.

    - **Constraint-aware Plan:** The composition plan which describes the service nodes that constitute the composite service and the relationships between them. Even if there are no constraints associated with any node in the plan, the structure must be the same as that of the constraint-aware plan defined in the underlying composition model.

  While the serialized composite service reader directly reads a complete service object from the repository file, the XML reader needs to extract all the elements mentioned above from the file and then create a layered composite service object using them. For that, it first uses the constrained service decorator from the service repository implementation to create a simple constrained service using the composite service name, inputs, outputs, effects and constraints. Then, it invokes the newly-created layered composite service decorator to combine this constrained service and the constraint-aware plan to form a layered composite service object.

- **Composite Service Readers**

  The user can offer one of several types of composite service repositories as part of the CS configuration from which the intended layered composite service can be extracted. Presently, the implementation can parse serialized Java object and XML file repositories. However, the architecture in place allows modular extension and modification of these options. Specifications of the architecture and implementation are as follows:

- **CompositeServiceReader:** The interface to be implemented by all concrete composite service readers. It declares the readCompositeService method which should be defined by each concrete reader. The method is responsible for finding a specific composite service by its name in the given composite service repository file, parse it and create a composite service object for it. The search and parse activities depend on the type of repository being handled by each concrete reader.
  The repository file name and location, target service's name and a logger object for recording error messages generated while reading are provided as input parameters to the readCompositeService method.

- **SerializedCSReader:** Concrete reader for extracting a specific composite service from a repository file of serialized composite service Java objects. It implements the CompositeServiceReader interface and defines the readCompositeService method. The method first uses the ServiceSerializedParser defined in the service repository implementation to read all the services contained within the repository. Then, the extracted service list is searched for the target service by its name. If found, the target service object is returned.
  In case the target service is not found in the repository, a suitable error message is recorded in the log file using the input logger object, and null is returned.

- **XMLCSReader:** Concrete reader for extracting a specific composite service from an XML composite service repository file. It implements the CompositeServiceReader interface and defines the readCompositeService method. The method first fetches a list of all the "compositeservice" XML nodes from the file. Then, the list is searched for the target service by its name. If found, the "compositeservice" node is completely parsed to obtain all the elements required to create the layered composite service object to be returned. The other nodes in the list are ignored.
  In case the target service is not found in the repository, a suitable error message is recorded in the log file using the input logger object, and null is returned.

Readers for other repository formats can be easily added to the existing architecture by implementing the CompositeServiceReader interface and defining the readCompositeService method to do the format-specific parsing. Concrete readers for modes other than files, such as, databases, can use the repository filename parameter of the readCompositeService method for accepting the information required to access the collection of composite services stored by them.

Assumption: There are no validation errors in the given composite service repository files. If a composite service exists in the repository, then it can be parsed successfully into a layered composite service.

Put svc validation checks in a common file, perform checks on cs elements

Assumption is that xml file will have all the xml elements and tags and attributes correctly. No checks are performed for those errors. Add these checks as a future work.

Diagram

Xml file rep

Cs constraints are ignored in lucid translation

Future Work

- At present, no validation checks are performed while parsing composite service repositories to extract a specific service from them for translation. To improve the reliability and robustness of the translation process, checks similar to the ones performed during the service composition process can be performed on the composite service elements extracted from the files.