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 valid layered composite service.

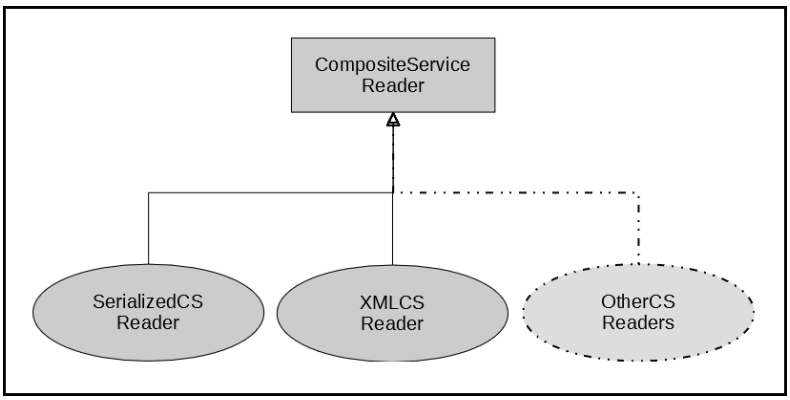

Figure 1: Composite service reader architecture

- **Composite Service Repository File Descriptions**
  - **Serialized Java Object File**

    It should be a text file containing a serialized Java ArrayList of Service objects. "Service" class refers to the one defined in the service repository implementation. Even if the repository is meant to hold a single composite service, it should be stored as part of an ArrayList.

    Although the composite service reader will be able to extract objects of any sub-class of Service class from the file, in order to ensure correct translation, the target object must belong to the LayeredCompositeService class.

    There is no restriction on the name or location of the file other than its extension, which must be "txt".

  - **XML File**

    It contains the following elements:

    - **compositeservices:** It is the root element of the XML file.
    - **compositeservice:** It appears as a sub-element of the root, once for every composite service that resides in the repository. All the information about a composite service is stored within the sub-elements of its corresponding "compositeservice" element.
    - **csname:** It appears once as a sub-element of every "compositeservice" element. The value assigned to its "value" attribute is the corresponding composite service's name.
    - **csinputs:** It appears once as a sub-element of every "compositeservice" element. For each input of the corresponding composite service, an "instance" sub-element is added to it. The value assigned to the "name" attribute of the "instance" element is *"<input data type> : <input name>"*.
    - **csoutputs:** It appears once as a sub-element of every "compositeservice" element. For each output of the corresponding composite service, an "instance" sub-element is added to it. The value assigned to the "name" attribute of the "instance" element is *"<output data type> : <output name>"*.

- **cseffects:** It appears once as a sub-element of every "compositeservice" element. For each effect of the corresponding composite service, an "instance" sub-element is added to it. The value assigned to the "name" attribute of the "instance" element is *"<effect data type> : <effect name>"*.
- **csconstraints:** It appears once as a sub-element of every "compositeservice" element. For each constraint of the corresponding composite service, an "instance" sub-element is added to it.

  Each "instance" element must have four sub-elements: "servicename", "literalvalue", "type" and "operator". Each of these sub-elements appears once and has an attribute called "name". The values assigned to this attribute are as follows:
  - **servicename:** Name of the atomic service to which the constraint belongs. If the constraint is user-defined for the composite service, name should be of the composite service.
  - **literalvalue:** Literal value component of the constraint
  - **type:** Feature component of the constraint, specified as *"<feature data type> : <feature name>"* for non-QoS features and as *"<QoS feature name>"* for QoS features
  - **operator:** Operator component of the constraint
- **csplan:** It appears once as a sub-element of every "compositeservice" element. It describes the constraint-aware service nodes that constitute the composition plan of the composite service and contains the following sub-elements:
  - **servicelayer:** It appears as a sub-element of "csplan" element, once for each service layer in the plan. The value assigned to its "index" attribute is an integer indicating the index of the layer, starting from 0 and increasing by 1 for each subsequent layer.
  - **servicenode:** It appears as a sub-element of a "servicelayer" element, once for each service node belonging to that layer.
  - **service:** It appears once as a sub-element of each "servicenode" element. The value assigned to its "name" attribute is the name of the service represented by the service node.
  - **constraints:** It appears once as a sub-element of each "servicenode" element. It describes the constraints attached to the service node. It follows the same format as that of the "csconstraints" element. Each constraint gets assigned an "instance" sub-element, which in turn is composed of sub-elements called "servicename", "literalvalue", "type" and "operator".
  - **predecessors:** It appears once as a sub-element of each "servicenode" element. For each predecessor node of the current service node, an "instance" sub-element gets added to it. The values assigned to the "name" and "layerindex" attributes of the "instance" element are the predecessor's service name and container service layer index respectively.

- **csatomicservices:** It appears once as a sub-element of every “compositeservice” element and lists the descriptions of all the atomic services that constitute the composite service. It contains the following sub-elements:
  - **service:** It appears as a sub-element of a “csatomicservices” element, once for each atomic service. The value assigned to its “name” attribute is the name of the service.
  - **inputs:** It appears once as a sub-element of each “service” element. It follows the same format as that of the “csinputs” element. Each input gets assigned an “instance” sub-element.
  - **outputs:** It appears once as a sub-element of each “service” element. It follows the same format as that of the “csoutputs” element. Each output gets assigned an “instance” sub-element.
  - **constraints:** It appears once as a sub-element of each “service” element. It describes the service constraints and follows the same format as that of the “csconstraints” element. Each constraint gets assigned an “instance” sub-element, which in turn is composed of sub-elements called “servicename”, “literalvalue”, “type” and “operator”.
  - **effects:** It appears once as a sub-element of each “service” element. It follows the same format as that of the “cseffects” element. Each effect gets assigned an “instance” sub-element.

For excluding the optional properties, such as constraints and predecessors, from a service description, whether atomic or composite, the main element of the property must be included without any sub-elements. E.g., for a composite service that does not have any constraints, the “csconstraints” tags must be included in the XML file without any “instance” sub-elements.

There is no restriction on the name or location of the repository file other than its extension, which must be “xml”. A sample XML repository file (XML_Repository.xml) has been placed with this document.

- **Expected Composite Service Specifications**

The composite services to be translated must follow certain specifications for correct translation. Since, at present, there are no validation checks in place to enforce these conditions, it is the responsibility of the composite service creator to ensure that all the conditions listed below are met.

- Each composite service in a repository must have a unique name.
- Each atomic service listed as part of a composite service must have a unique name. Atomic services belonging to different composite services may have the same names. Also, a composite service may be listed as an atomic service for another composite service but not for itself.
- Each service, whether composite or atomic, must have at least one input and one output.

- The effects of a composite service must be a set of all the outputs of all its component atomic services.
- The effects of an atomic service must be the same as its outputs.
- The acceptable data types for service parameters (inputs, outputs, effects and constraint features) are int, char, float, boolean and string.
- The constraints of a composite service must be a union of two sets: a set of all the constraints of all its component atomic services and a set of user constraints imposed on the composite service as a whole.
- User constraints for the composite service must be applied only on (i.e., must have as feature) the composite service's inputs, outputs and QoS features.
- The acceptable QoS features are COST, RESPONSE_TIME, RELIABILITY and AVAILABILITY. QoS features do not have a data type.
- Feature of a constraint must follow the same format and naming convention as the parameter being used as the feature.
- The acceptable operators for constraints are <, >, =, <= and >=.
- The operators and literal values defined for a constraint must be compatible with the type of its feature.
- Parameter names and types are case-sensitive.
- Although the number of component services does not affect the translation process, conceptually, a composite service should be composed of at least two services.
- For translation to Objective Lucid, service parameter names and constraint literal values must not have any spaces. This restriction does not stand for translation to XML.

Composite Service (CS) Configuration and its Readers

- **Composite Service Configuration**

  A CS configuration object contains all the information provided by the user that is required to execute the service translation process. It consists of the following elements:

  - **Composite Service:** Layered composite service object to be translated.
  - **CS Input Details:** List of records, one for each input of the composite service. Every record consists of a specific input's name, data type and value.
  - **Target Language:** The language to which the composite service needs to be translated. At present, the implementation supports Objective Lucid and XML languages.
  - **Destination Folder:** Complete name and path of the folder where the file containing the target language code will be placed.

  Once created, the target language of a CS configuration object is used to invoke the appropriate translator, which accepts the configuration object as an argument. While the Objective Lucid translator uses the composite service and input details from the configuration for generating the target language code, the XML translator only needs the composite service object to complete the translation. Both translators use the destination folder information to store the translated code files that they generate.

- **Composite Service Configuration Readers**

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

  - **CSConfigReader:** The interface to be implemented by all concrete CS configuration readers. It declares the readCSConfig method which should be defined to accept composite service configuration details from the user based on the mode of input being handled by each concrete reader.
    A logger object for recording error messages generated while reading is provided as an input parameter to the readCSConfig method.

  - **ConsoleCSConfigReader:** Concrete CS configuration reader for interacting with the user through the console to obtain composite service configuration details. It implements the CSConfigReader interface and defines the readCSConfig method.
    The readCSConfig method is responsible for performing the following tasks:
    - Reading the CS repository name and location, CS name, target language and CS input values from the console.
    - Invoking a specific CS reader based on the repository type for creating an object of the composite service to be translated using the repository and service name.
    - Using the repository file location as the destination folder for the translation.
    - Creating a collection of input records using the input details from the composite service object and the values read from the user. This is required only for Lucid translation.
    - Creating a CS configuration object using all the information so collected.
    - In case of any failures during the creation of the configuration object, recording proper error messages in the log file using the input logger object and aborting the process.

  - **FileCSConfigReader:** Abstract class to be extended by all concrete CS configuration file readers. It implements the CSConfigReader interface but does not define the readCSConfig 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.

  - **XMLFileCSConfigReader:** Concrete CS configuration reader for extracting composite service configuration details from a user-specified XML file. It extends the FileCSConfigReader class and defines the readCSConfig method.
    The readCSConfig method is responsible for performing the following tasks:
    - Reading the CS repository name and location, CS name, target language and CS input details from the configuration XML file.

- ❖ Invoking a specific CS reader based on the repository type for creating an object of the composite service to be translated using the repository and service name.
- ❖ Using the repository file location as the destination folder for the translation.
- ❖ Creating a collection of input records using the input details from the composite service object and the values read from the configuration file. This is required only for Lucid translation.
- ❖ Creating a CS configuration object using all the information so collected.
- ❖ In case of any failures during the creation of the configuration object, recording proper error messages in the log file using the input logger object and aborting the process.

Readers for other file formats can be easily added to the existing architecture by extending the FileCSConfigReader class and defining the readCSConfig 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 CSConfigReader interface while defining the required behavior in the readCSConfig method.

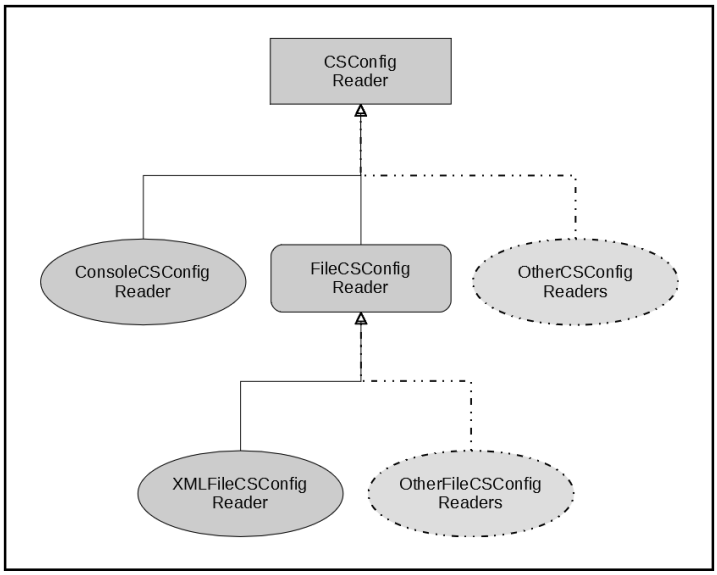

Figure 2: Composite service configuration reader architecture

- **Composite Service Configuration Input Format**

  The format in which the readers expect to receive the CS configuration details have been described below for each acceptable mode of input:

  - **Console**

    The user will be prompted to provide each of the required elements' values. The elements and their expected input formats are as follows:

- **CS Repository Filename:** Complete name (with extension) and path of the composite service repository file. For now, only “txt” and “xml” are recognized as acceptable file extensions.
- **Composite Service Name:** Name of the composite service to be extracted from the given repository and later translated into the target language code.
- **Target Language Name:** Name of the formal language to which the given composite service needs to be translated. For now, only “Lucid” and “XML” are recognized as acceptable target languages.
- **CS Input Values:** Value to be assigned to each composite service input according to the name and data type mentioned in the prompt for each specific input. User will not be prompted for input values if the target language is XML.

- **XML File**

  The elements of the CS configuration XML file have been described below. Each of these, except for “input”, is mandatory.

  - **csconfig:** It is the root element of the XML file.
  - **csrepofilename:** It appears only once, as a sub-element of the root. The value assigned to its “value” attribute is the complete name (with extension) and path of the composite service repository file. For now, only “txt” and “xml” are recognized as acceptable file extensions.
  - **csname:** It appears only once, as a sub-element of the root. The value assigned to its “value” attribute is the name of the composite service to be extracted from the given repository and later translated into the target language code.
  - **targetlang:** It appears only once, as a sub-element of the root. The value assigned to its “value” attribute is the name of the formal language to which the given composite service needs to be translated. For now, only “Lucid” and “XML” are recognized as acceptable target languages.
  - **input:** It appears once for each composite service input, as a sub-element of the root. It has three sub-elements called “name”, “type” and “value”. The values assigned to the “value” attributes of these sub-elements are, respectively, the corresponding input’s name, data type and value.

    Input details need to be provided only for translation to Lucid. For XML translation, the “input” elements are not required.

  A sample configuration XML file (CS_Configuration.xml) has been placed with this document for reference. There is no restriction on the name or location of the configuration file other than its extension, which must be “xml”.

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.