Case study 1: Creating and configuring a reengineering method based on MLSACA metamodel

Suppose a development team aims at deploying a legacy application stack on EC2 Amazon cloud servers to reach flexible fees based on the amount of computing resources. This scenario is classified as migration type V, i.e. a virtual-machine-based application migration to the cloud using IaaS model. In this scenario, the method engineer merely focuses on enabling phase of the reengineering process and does not concern other phases. Figure 1 depicts the four-step procedure of the metamodel tailoring towards creating a bespoke method for this scenario. The following elaborates on the way MLSAC implements these steps.

Step 1. The inputs to the MLSAC are three parameters including a method name/description and the choice of migration types and phases.

Step 2. Inputs parameters are used to perform a vertical transformation where the metamodel as the source model at M2-layer is instantiated to a new method as a target model at M1-layer. In the current example, relevant elements to the migration type V and *enable* phase are selected by MLSAC for inclusion in the new method. These elements are *resolve incompatibilities* (including subclasses *refactor codes*, *develop integrator*, and *adapt data*), *encrypt/decrypt entities* (including subclasses *encrypt database*, *obfuscate codes*, and *encrypt/decrypt messages*), *enable elasticity*, *test application* (including all test subclasses), *configure network*, and *deploy system components*.

MLSAC also specifies elements that are mandatory, situational, or unnecessary for incorporating into the method based on the information in the knowledge source (See the knowledge source file in GitHub main directory). For instance, *deploy application components* is a mandatory task, which is denoted by the symbol √. On the other hand, the incorporation of tasks related to incompatibility resolution is subjected to the choice of a cloud platform, which is denoted by symbol (√). That is, MLSAC recommends to developers that if legacy system components are to be bound to specific cloud services, potential incompatibilities between these two platforms should be identified and subsequently resolved through adaptation mechanisms. At the end of this step, the new method containing a set of method fragments and their definitions is created. Figure 2 shows a snapshot of this method. The graphical user interface in Figure 2 has three main sections. The upper section contains the method name, the migration type for which the method is created, and a general description of the method. The bottom-left section shows the method elements reused from the metamodel where the method engineer can browse through the method elements and their definitions. The bottom-right section has the information about an element once the method engineer clicks on it in the tree view. Different icons are used in MLSAC to illustrate the classification of a method's elements such as phases, tasks, and work-products.

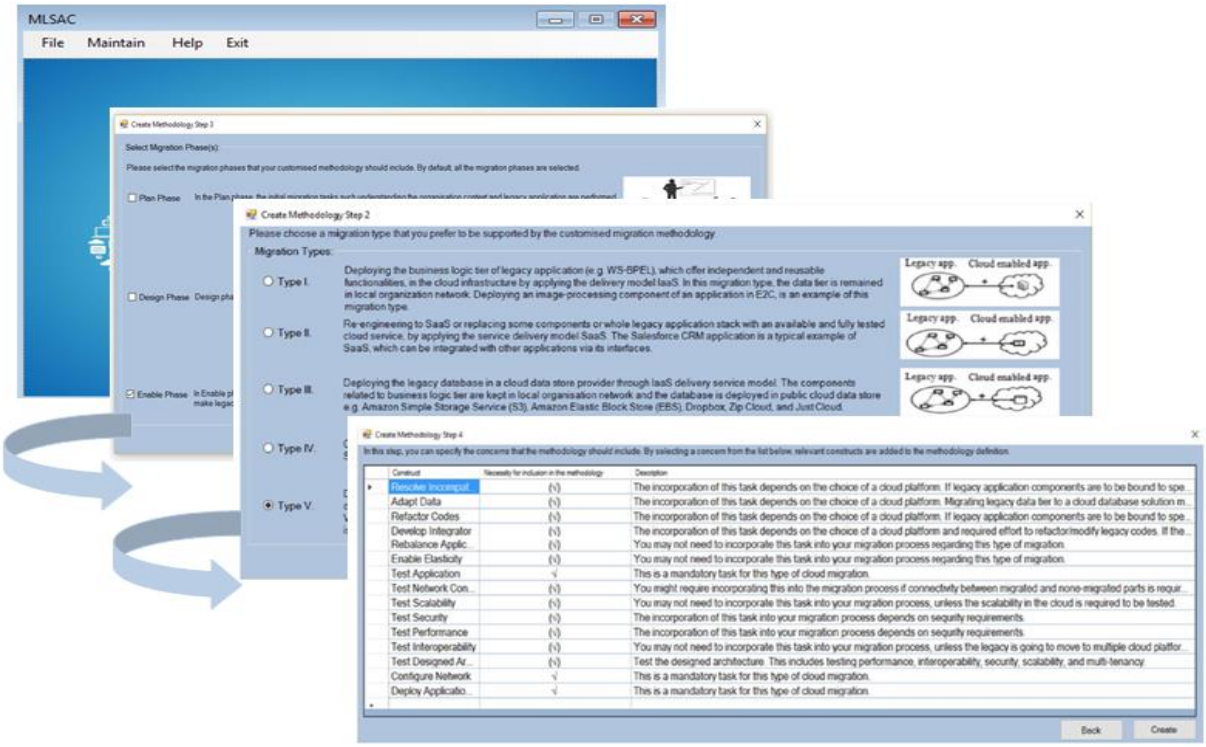

Figure 1. steps for method creation

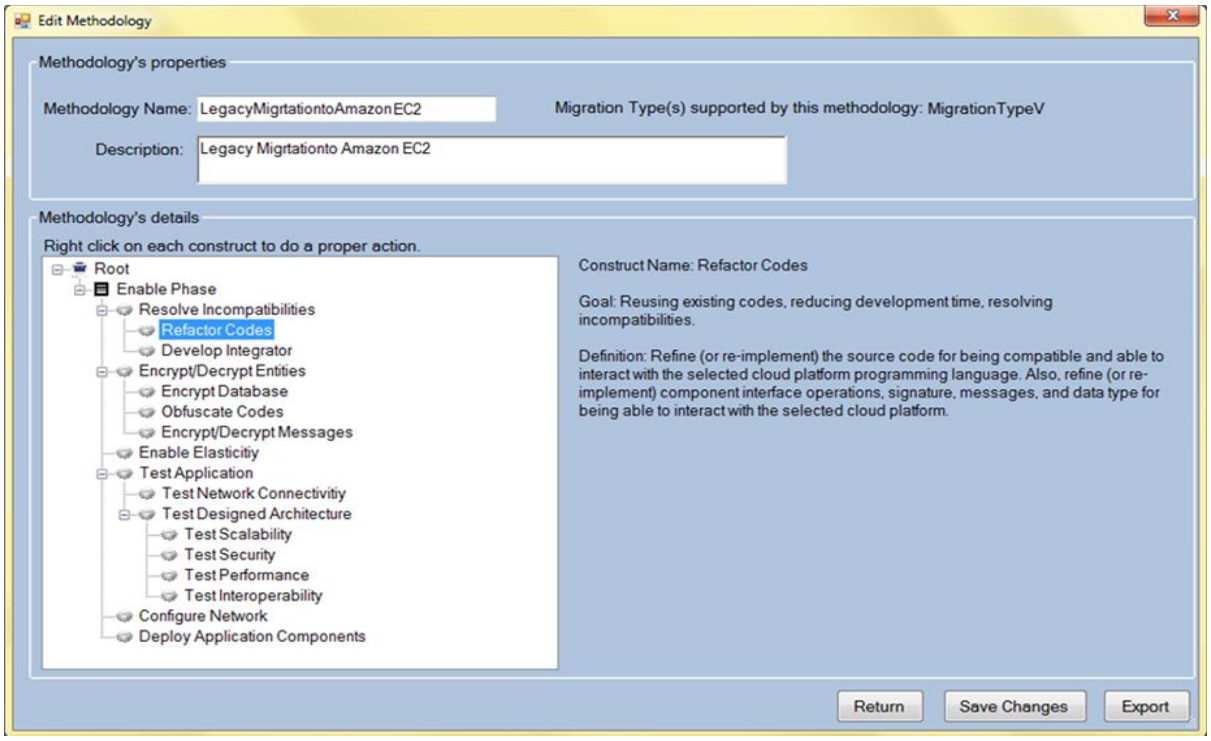

Figure 2. the created method

Step3.1 The method engineer may believe in excluding some default elements in the method that are not necessary. For example, the method engineer recognizes that there are strong data security mechanisms in Amazon servers. In addition, the database includes non-critical data for example temporary tables. Thus, there is no longer need for encrypting the application database. Then she removes the element *encrypt database* from the method to avoid application performance degradation.

Step3.2 The tailoring procedure allows adding of new elements such as phase, tasks, and work-products, which might not have been already supported by the meta-model. These elements can be

added from various sources such as past developer experience or existing methods. For example, the method engineer may define a particular legacy system code refactoring to inform developers of updating the legacy system APIs to make them compatible with Amazon EC2 APIs to prevent semantical or syntactical incompatibilities. The method engineer adds a new element, as a subclass and named *update APIs and framework*, under the element *refactor codes* in enable phase (Figures 3 and 4).

Step3.3 MLSAC gives the ability to define arbitrary implementation techniques to operationalize method elements. Suppose, the organisation wants to move the legacy database to Amazon Aurora database analytics engine. It is likely that the meaning of some legacy database table columns is the same as target cloud database solution database table columns, however, the unit of measurement is different which raises a semantic change in the interpretation of the table data field. For example, both platforms may use different currencies or precision in saving decimal points. In this scenario, it is important to not lose or change the original legacy application data after integration with Amazon Aurora. To assure this, as shown in Figure 3, the method engineer firstly prescribes the following technique including two steps (1) Interoperability test should cover test cases for identifying semantic inconsistencies and incompatibilities between the legacy database and migrated data to Amazon Aurora database. The important columns that should be examined are those that contain currency and decimal points, (2) developers should manually compare columns in both legacy and Amazon Aurora databases by looking at the main screen of the system. This technique is then assigned to the element *test interoperability* (Figure 5 and 6).

Step3.4 The method engineer can override existing sequences defined by the MLSAC to define a specific sequence of method elements. Figure 7, shows she defines that *recover legacy application knowledge* should be performed before a cloud platform is chosen via task *choose cloud provider*.

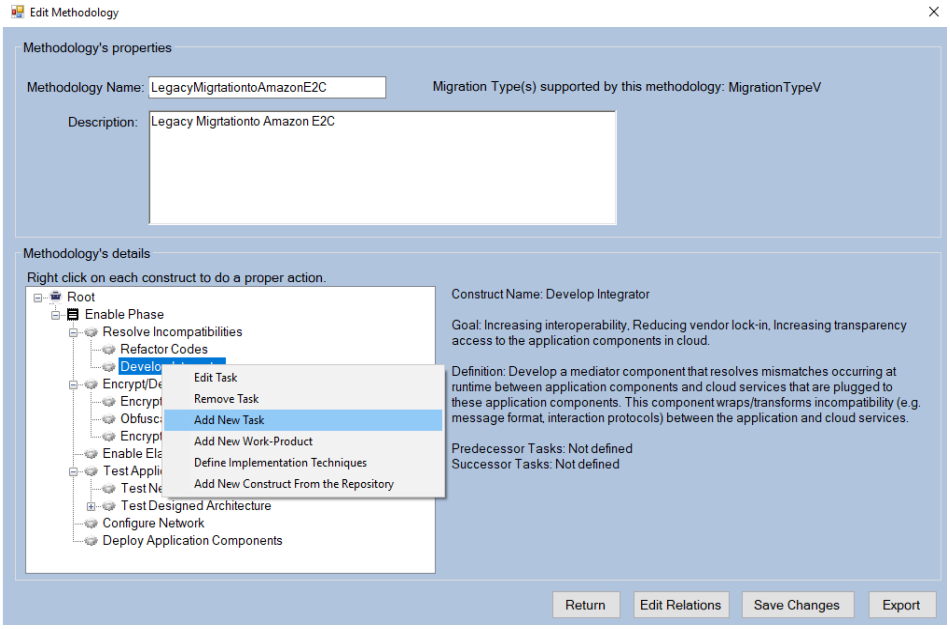

Figure 3. Adding new task element to the method

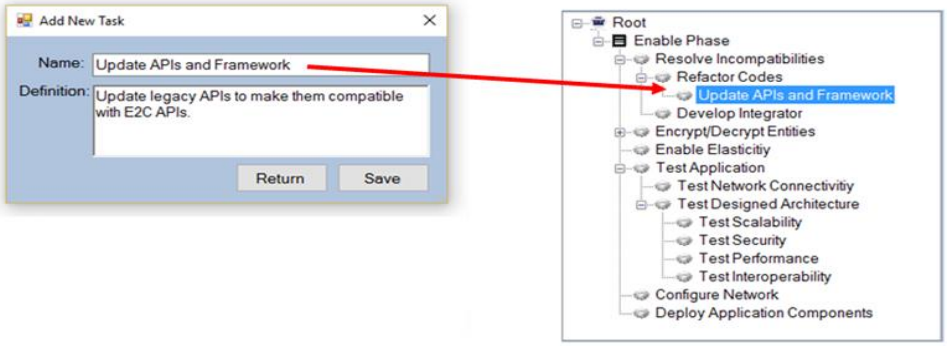

Figure 4. Adding a subclass to the method element *refactor codes*

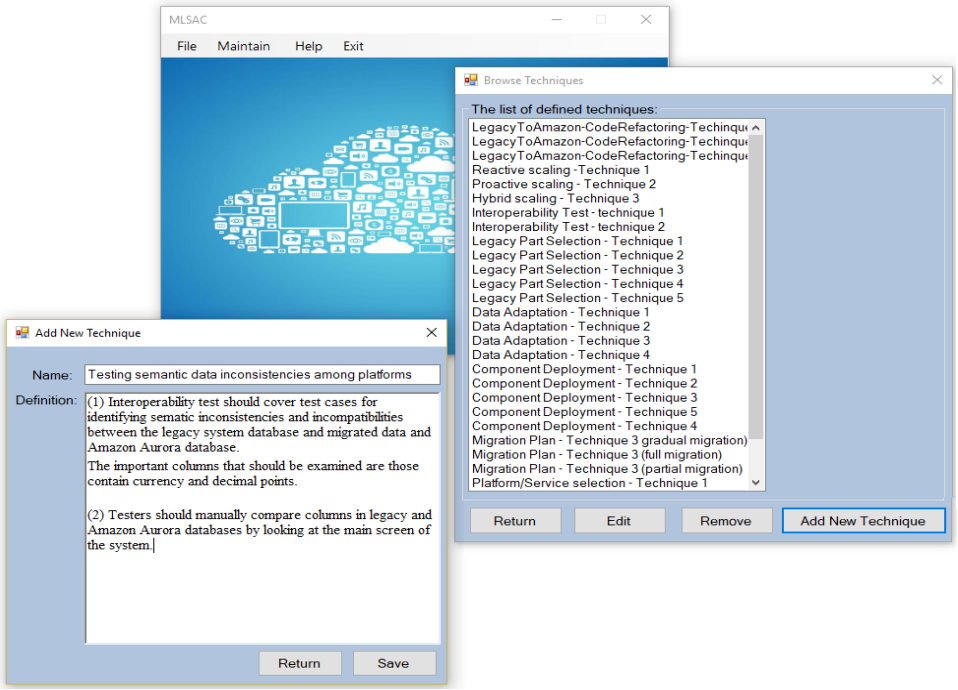

Figure 5 Defining an implementing technique for *test interoperability* method element avoiding semantic data inconsistency

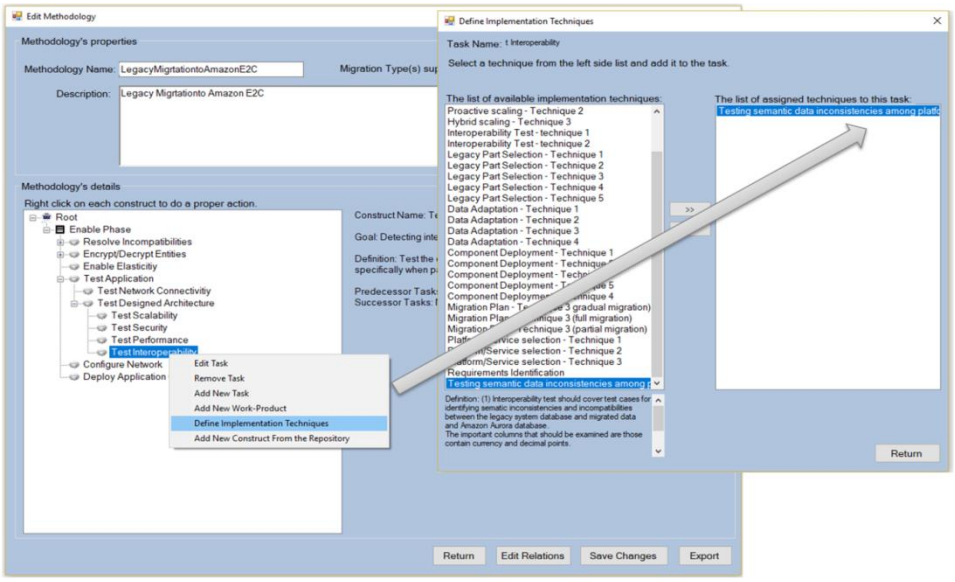

Figure 6 Assigning the defined technique to method element *test interoperability*

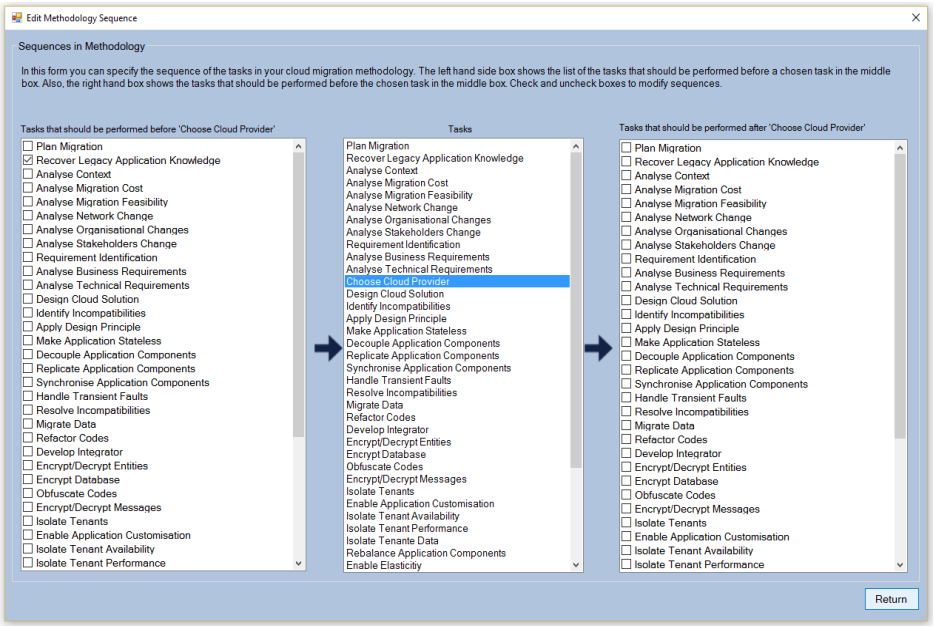

Figure 7 Specifying sequences among method elements, i.e. tasks