Specification of a Legacy Tool by Means of a Dependency Graph to Improve its Reusability

Paola Vallejo, Mickaël Kerboeuf, and Jean-Philippe Babau

University of Brest (France), Lab-STICC, MOCS Team
{vallejoco,kerboeuf,babau}@univ-brest.fr

Abstract. This position paper, investigates a way to improve the reusability of legacy tools in specific contexts (defined by specific metamodels). The approach is based on a dedicated language for co-evolution, called Modif. Its associated process involves two model migrations. The first one (Migration), allows to put data under the scope of a legacy tool. The second one (Reverse Migration), allows to put the legacy tool's output back into the original specific context. The approach is generalized by introducing the notion of dependency graph. It specifies the relations between the legacy tool's input and the legacy tool's output. The dependency graph is then used to address some complexities of the Reverse Migration. The improvement is illustrated by the reuse of a flattener tool defined on a specific metamodel of FSM (finite state machines).

Keywords: Legacy tool's reusability, DSML, metamodel transformation, model migration, code generation

1 Introduction

Reuse is the act of using an asset in different systems [2]. In DSML (Domain-Specific Modeling Languages), the reuse of legacy tools reduces the cost of producing the entire tool support of a DSML. The reuse is also employed in other contexts such as model transformations, [6] proposes reusing transformations instead of rewrite them.

The reuse brings up some difficulties with it; for example, when a designer is defining specific functions for the DSML, he frequently notices that the functions are already provided by a legacy tool. Nevertheless, they were developed for a variant of his metamodel.

In this regard, the aim of reusing, raises two questions: how the DSML model can be adapted to be conform to the legacy tool's metamodel? And how the output of the legacy tool can be adapted in return to the DSML context?

Modif [1] [5] addresses those two questions. Figure 1 shows the operations performed by *Modif* to handle the interactions between the elements of the two contexts (DSML and legacy tool). The DSML context's metamodel *MM*, its conforming model *M1*, the legacy tool context's metamodel *MM'* and the legacy tool *Tool* are those we aim at reusing. Then, from *M1*, the objective is to obtain *M2*, *M3* and *M4* automatically.

M1 has to be adapted to be *M2*, with the purpose of match the legacy tool context. The adaptation is achieved by the Modif's *Adaptation* step. In accordance with the principles of co-evolution between metamodels and models, *Adaptation* performs *Refactoring* operations at the metamodel level and *Migration* at the model level [3] [8]. Once *Tool* has processed *M3* from *M2*, it is necessary to adapt it to the DSML context by producing *M4*. The *Reverse Migration* is achieved by the Modif's *Contextualization* step, thanks to the relational notion of *key*.

The process compound of *Adaptation* (M1 to M2), *Tool* and *Contextualization* (M3 to M4) correspond to the *Tool reuse*.

It is important to notice that a common operation performed during *Migration* is the deletion of unnecessary information (slicing operation [7]). Then, the *keys* mechanism allows to recover at *Contextualization*, the instances that have been deleted during *Migration*. This approach based on *keys*, presents some limits when the legacy tool creates of aggregates different instances. Hence, the interest of find a mechanism able to contextualize deleted instances, but also the new ones.

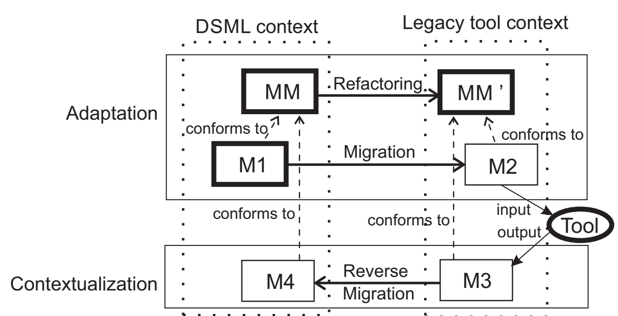

Fig. 1. Legacy tool reuse's process realized by Modif

In this paper, we aim at improving the Modif's *keys* mechanism [5] by using a *dedicated dependency graph*. Such graph determines the set of instances from the legacy tool's input that have been used to update or create an instance of the legacy tool's output.

This paper is organized as follows. The next section presents the background of this work and some motivations to improve the *keys* mechanism, in order to make a correct contextualization. It takes into account the links between *Migration* and *Reverse Migration*. Then, we present the proposition to assist the user in the process of putting back the legacy tool's output into his DSML context. We finally conclude the paper and give some perspectives.

2 Background and Motivation

In a simple example we show the *tool reuse* process and why *Reverse Migration* is a key problem.

Adaptation Modif's *Adaptation* is based on co-evolution operators (e.g. update, delete) like classically proposed by [4]. A legacy tool is defined for a specific usage, and its metamodel includes less concepts than a DSML metamodel proposes. Then, the most used operators for adaptation are rename and delete.

Tool The input and the output of the legacy tool *Tool* conform to the same metamodel. *Tool* executes creation, update and deletion.

Contextualization Modif [5] proposes a *keys* mechanism. A *key* is an attribute associated to each instance of *M1* that uniquely identifies it. The *keys* allow to keep a relationship between instances of *M1* and those still exist in *M2* and *M3* after *Migration* and *Tool* application. Then, *M4* is built by adding *M3* instances and instances that have been deleted during *Adaptation*.

This is possible by applying the concept of relational natural join of relational databases. The relationships between instances are also built reusing the *keys* information. Thus, instances of the legacy tool's output are reconnected to the instances that have been recovered. New instances cannot be reconnected to other instances, and an instance of *M3* can be reconnected to only one existing instance of *M1*.

To illustrate the approach and its limits, a case study of simple FSM is presented:

- *MM* defines the concepts of State, Transition, Action (associated to states) and Event (associated to transition). A state can contain other states inside it (hierarchical finite state machine);
- *MM'* is the metamodel of input data expected by a flattener legacy tool, it is similar to *MM*, except that it does not contain actions;
- *M1* (Figure 2) is a state-machine model conforms to *MM*. It is composed of a super-state with two actions, a substate with one action, a substate without actions, and two transitions;
- *M2* (Figure 2) is an adaptation of *M1*, in which actions are deleted;
- *Tool* is a flattener legacy tool that removes hierarchy by producing atomic states (aggregation of super-states and states). For each super-state, all substates are renamed and itself is removed. The renaming is done by concatenating the super-state's name and the substate's name;
- *M3* (Figure 2) depicts the legacy tool's output. Actions have to be reintegrated to it;

– *M4* (Figure 2) illustrates the result of the *Reverse Migration* by using the *keys* mechanism. The action *run1* is recovered and reconnected to *running nominal*. The actions *start* and *stop* are lost because they are associated to the super-state *running* that does not exist after tool application.

Figure 2 illustrates the way in which the states evolve and the keys are propagated. Only K*i* by characterizing a state concept are shown. This example underlines the limits of the approach by only using *keys* mechanism. The actions associated to a super-state cannot be recovered automatically. And, if *Tool* performs creation of new instances instead of updating the existing ones, it is not possible to recover any deleted action. In this case, either instances that have been deleted in *Migration* are lost, or specific user code has to be added to improve the *Reverse Migration*.

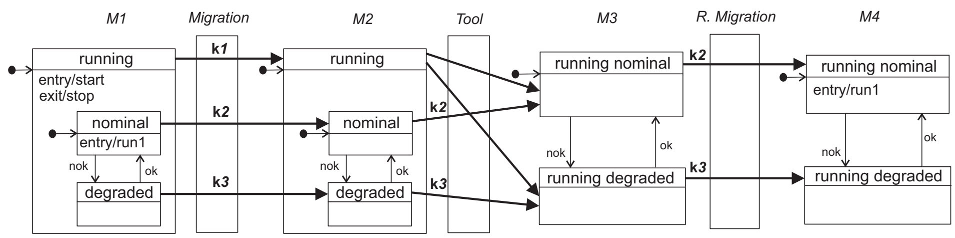

Fig. 2. States' evolution in a tool's reuse process

The major difficulties in the context of *tool reuse* are:

– *Reverse Migration* is not limited to be an inverse *Migration*, because of *Contextualization*, for example by using *keys*;
– *Contextualization* is not limited on adaptation, because it depends also on the legacy tool's behavior impact.

When *Tool* creates new instances, information about the relation between the new instances and the existing ones is missing. We propose to add information about the tool behavior impact on *Reverse Migration*.

3 Approach

3.1 Proposition

We present a proposition to enhance *Modif* and its *keys* mechanism, by introducing the notion of *dependency graph*. A dependency graph is considered a *specification* of the legacy tool. It specifies the dependencies between each instance of the legacy tool's output and a set of instances of the legacy tool's input. The set is compound of the instances that are involved in the creation or modification of the legacy tool's input instance. All types of instances can participate in the creation or update of other instances.

In this paper, the dependency graph is obtained by instrumenting the legacy tool. We log each concept of the legacy tool's output and the set of concepts of the legacy tool's input that participates in its creation or modification.

For the case study of FSM, *Reverse Migration* is applied to *M3* using the *keys* mechanism, in order to recover deleted actions. Moreover, we also use the information given by the dependency graph to reconnect more actions. For this example, the relations between input and output of the legacy tool are shown in Figure 3. Now, the challenge is how to use this information to keep the recovered instances and to reconnect them.

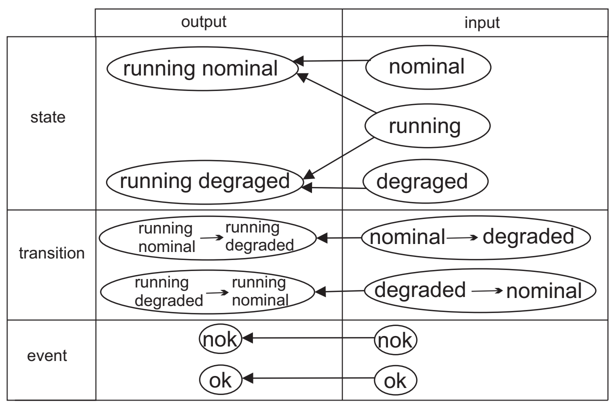

Fig. 3. Relation between the flattener's input and output

Reverse Migration is parameterized by *Adaptation* (initial model and *keys*), *dependency graph* (tool behavior) and legacy tool's output. From those parameters, the generated code can be executed to get a contextualized final model.

The process performed by the generated code to produce the final model is:

- to make an identical copy of each instance of the tool's output, taking into account its attributes and its references;
- to use the *keys* to identify the deleted instances;
- to recover the links to deleted instances and filter them by type, using the information provided by the *dependency graph*. The filter allows to recover instances of the appropriate type. We consider that an instance may be created from only instances of the same type (e.g. states are created from states);
- to offer an extension point in which the user can specialize the *by default* behavior by defining its *customized behavior*. If there is not *customized* behavior, only *by default* behavior is executed.

3.2 Experimentation

The approach is experimented with the case study of flattening finite state machines.

The following is the *by default* behavior proposed to reconnect each recovered action:

R1 If its related state in *M1* still exists in *M3*; the action is automatically connected to it;

R2 If the state no longer exists in *M3*, but another states of *M1* are related to it and they still exist; then, the action is connected to all of them;

R3 If the state no longer exists in *M3* neither the state related to it; then, the action is not connected to any state.

An excerpt of the code generated by *Modif* is shown in Listing 1.1. *function* is the main function, it takes as parameters the legacy tool output *M3model*, the dependency graph *dicoKeys* (it contains also the *keys*) and the initial model *M1model*).

Listing 1.1. Generated main class for the state machine example

```
public class ReverseMigration{
 migration (new DefaultBehavior()) ;
 migration (new CustomizedBehavior()) ;

 // Reverse Migration
 final void function(M3 M3model, Key dicoKeys, M1 M1model){

  for(M3.State state : M3StatesList){

   // related entry actions of an state
   relatedEntryActions=getRelatedEntryActions(state, dicoKeys, M1model);

   // related exit actions of an state
   relatedExitActions=getRelatedEntryActions(state, dicoKeys, M1model);

   for(M1.State related : relatedStates){

    // by default behavior
    byDefault.connectEntryAction(state, relatedEntryActions);
    byDefault.connectExitAction(state, relatedExitActions);

    // customized behavior
    customized.connectEntryAction(state, relatedEntryActions);
   }
  }
...
```

Listing 1.2 shows the functions *connectEntryAction* and *connectExitAction*. They are responsible for reconnect entry and exit actions to the states, taking into account the information gathered from the *dependency graph*. These functions execute the behavior defined in R1, R2 and R3.

If the designer does not agree the *by default* behavior, he can specialize the code by integrating his requirements. An example of the customized behavior defined by an user is (Listing 1.3):

D1 If the recovered action is an *entry* one, it is reconnected to only initial states (the attribute *initial* is set to *true*): it is an adaptation of R1;

D2 If the action is an *exit* one, it is reconnected to all states: it was already defined by R2.

Listing 1.2. By default behavior for the state machine example

```
public class DefaultBehavior implements Migration{

 // Function to reconnect entry actions
 public void connectEntryAction
                  (M3.State state, Elist<M1.Action> relatedEntryActions){
   ...
 }

  // Function to reconnect exit actions
 public void connectExitAction
                  (M3.State state, Elist<M1.Action> relatedEntryActions){
   ...
 }
```

Listing 1.3. Customized behavior for the state machine example

```
public class CustomizedBehavior extends DefaultBehavior{
 ...
 public void connectEntryAction
                  (State state, ArrayList<Action> relatedEntryActions){
   if(state.isIni()){
    for(Action action : relatedEntryActions){
     state.setEntry(action);
   }
...
```

Figure 4 presents the result of executing R1; *run1* is reconnected because *running nominal* still exists after flattening. Figure 5 presents the result of executing R2; *start* and *stop* are reconnected because *running* was involved in the update of the two substates. In this example, there are not changes while executing R3.

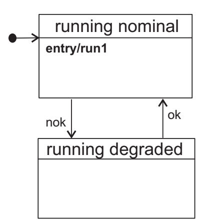

Fig. 4. R1 behavior

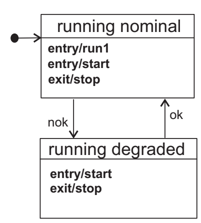

Fig. 5. R2 behavior

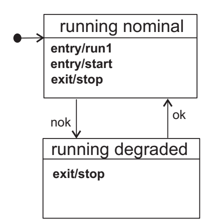

Fig. 6. D1 behavior

The final model obtained by following the *by default* behavior and then the *customized* behavior is presented in Figure 6. All actions are recovered and reconnected. *run1* still related to *running nominal*. *start* is deleted from *running degraded* because it is not an initial state. *stop* still is connected to all states.

This approach allows to keep at *Reverse Migration*, the DSML instances deleted during *Migration*. Contrary to the result obtained by using only the *keys* mechanism, the *dependency graph* allows to reconnect all actions without lost. Even if the legacy tool performs creation.

4 Conclusion and Future Works

In this paper, we present an approach to facilitate the legacy tool's reuse process. In particular, it improves the *Reverse Migration* for legacy tool's reuse by means of a dependency graph. The dependency graph provides a specification of the legacy tool to be reused. It enables to recover DSML instances deleted before using the legacy tool and to reintegrate them to its original DSML context.

Migration is metamodel dependent only; *Reverse Migration* is metamodel dependent, tool's behavior dependent, *Migration dependent* and original model dependent.

We are now working on the formalization of *Migration* and *Reverese Migration*. The approach will be experimented by reusing some legacy tools in the context of video transmission and coding in MPSoC.

References

1. J.-P. Babau and M. Kerboeuf. Domain Specific Language Modeling Facilities. In *proceedings of the 5th MoDELS workshop on Models and Evolution*, 2011.
2. J. L. Cybulski. Reuse introduction cybulski abstract introduction to software reuse, 1995.
3. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of ECMDA-FA*, 2009.
4. M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE*, 2010.
5. M. Kerboeuf and J.-P. Babau. A DSML for reversible transformations. In *proceedings of the 11th OOPSLA workshop on Domain-Specific Modeling*, 2011.
6. D. Mendez, A. Etien, A. Muller, and R. Casallas. Towards Transformation Migration After Metamodel Evolution. In *Model and Evolution Wokshop*, 2010.
7. S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model Pruning. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, Oct 2009.
8. G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proceedings of ECOOP*, 2007.