1 Introduction

This paper discusses the GReTL reference solution of the TTC 2011 Program Understanding case. The submitted solution covers both the core task as well as the two extension tasks.

GReTL (Graph Repository Transformation Language, [HEar, HE11]) is the operational transformation language of the TGraph technological space [ERW08]. Models are represented as typed, directed, ordered, and attributed graphs. GReTL uses the GReQL (Graph Repository Query Language, [EB10]) for its querying part. Transformations are either specified in plain Java using the GReTL API or using a simple DSL. In this paper, only the latter is used.

For interacting with the rest of the world, there are import/export tools for both EMF [SBPM09] as well as GXL.

The elementary GReTL operations follow the conception of incrementally constructing the target metamodel together with the target graph. When creating a new metamodel element, a set-based semantic expression is specified that describes the set of instances that have to be created in the target graph. This expression is described by a GReQL query on the source graph.

For this case, the given EMF JaMoPP models are converted to TGraphs, and these are used as the source graphs of the (out-place) transformation. The transformation itself creates the target statemachine metamodel, and thereby it creates one target graph per source graph conforming to that new metamodel. The target graphs are eventually exported back to EMF models.

[^1]: http://www.gupro.de/GXL/
In the next section, the solutions of the transformation tasks are discussed.

2 Task Solutions

As already mentioned, the submitted GReTL solution covers all tasks of the case. In this section, we will go through all tasks in sequence and explain the GReTL transformation operations and GReQL queries when they come along.

Running the solution. When logged into the SHARE image containing the GReTL solutions (Ubuntu_10.04_TTC11_gretl-cases), the reference solution for the program understanding case is located in the directory ~/Desktop/GReTL_TTC_2011_SOLUTIONS/PROGRAM_UNDERSTANDING/. The EMF import of the EMF models to TGraphs, followed by the execution of the GReTL transformation, and finally the EMF export can be performed by running the run.sh shell script in a terminal emulator.

The transformation ExtractStateMachines.gretl is located in the transforms/ directory. The target TGraphs, their visualizations, and the reconverted EMF models are generated into the output/ directory.

Hello World solution. As requested by the call for solutions, the mandatory Hello World solution is also provided on the SHARE image. The directory containing the solution including a paper explaining it is ~/Desktop/GReTL_TTC_2011_SOLUTIONS/HELLO_WORLD/.

2.1 The Core Task

The core task is responsible for creating States and Transitions without setting the attributes of the latter. As already mentioned in the introduction, the GReTL solution works without a preexisting statemachine target metamodel, but instead the transformation specifies it on its own.

In the following, the complete transformation is discussed. Every GReTL transformation starts by declaring its name.

```
1 transformation ExtractStateMachines;
```

Because the JaMoPP metamodel splits its types into various packages, we import the packages from which elements are used, so that we can refer to these types without having to qualify them.

```
2 import classifiers.*;
3 import types.*;
4 import members.*;
5 import references.*;
```
The first task we want to accomplish is the creation of State vertices. As explained in the task description, there is an abstract Java class named State, and all concrete subclasses can be considered states. At first, we bind that abstract State class to a variable abstractStateClass, so that we can easily refer to it in the transformation.

```gsql
abstractStateClass := theElement(from c: V(Class)
  with c.name = "State"
  reportSet c end);
```

The GReQL query following the assignment operator uses the theElement() function to extract the single element of a collection consisting of only one element. This function errors, if the given collection contains more or less than one element, so using it instead of simply choosing the first element makes our assumption that there is only one element more explicit.

A FWR expression (from-with-report) is used to calculate the collection. In the from part, a variable c is bound to every vertex of type Class one after the other. The with part restricts the classes to only those that are named “State”. The report part defines the structure of the result. Here, we return a set (reportSet) consisting of all class vertices that satisfy the constraint in the with part.

Now, we start with the transformation. The first operation invoked is CreateVertexClass. This operation creates a new vertex class (a node type) in the target metamodel. Its semantic expression is evaluated on the source graph and has to result in a set. For each member in this set (called archetype), a new vertex (called image) of the just created type is instantiated in the target graph. The mappings from archetypes to target graph images is saved in a function corresponding to the target metamodel vertex class.

```gsql
CreateVertexClass State
  <= from c: {Class} & (<|--{extends} <|--{classifierReferences} -->{target})+
    abstractStateClass
    with isEmpty(c <|--{annotationsAndModifiers} & {Abstract})
    reportSet c end;
```

In the target metamodel, the vertex class State is created. The semantic expression following the arrow symbol specifies a set of Class vertices. The variable c iterates over Class vertices, for which a path to the vertex abstractStateClass exists. The structure of this path is specified using a regular path expression [EB10]. First, an edge with containment semantics and role name extends at the far end has to be traversed, followed by another containment edge with role name classifierReferences, followed by a forward edge with role name target. This is exactly how subclasses relate to their superclass. The + specifies a one-or-many iteration. Thus, c is not only bound to direct subclasses of abstractStateClass, but also to indirect ones.
In the with part, the predicate ensures that \( c \) is not bound to an abstract class, i.e., a Class vertex that references an Abstract vertex using an edge with containment semantics and far end role name annotationsAndModifiers.

The report part defines the structure of the results. Here, we report the set of classes bound to \( c \) that fulfill the predicate in the with part.

Thus, for any non-abstract class that extends the abstract state class either directly or indirectly, a new target graph State vertex is created. The mappings from classes to states are stored in a function \( \text{img} \_\text{State} \), which can be used in following operation calls for navigating between archetypes and images. Furthermore, the inverse function \( \text{arch} \_\text{State} \) is provided. These function allow for bidirectional, constant-time navigation between archetypes and images and the other way round.

The next operation creates the name attribute of type String for the State vertex class, and it sets the attribute values for the vertices created by the last operation call.

```
16 CreateAttribute State.name : String
17 <= from c: keySet(img_State) reportMap c -> c.name end;
```

The semantic expression of the CreateAttribute operation has to result in a map. This map assigns to archetypes the values their target graph images should have set.

The variable \( c \) iterates over the key set of the \( \text{img} \_\text{State} \) function, i.e., source graph classes extending the abstract state class. To each of these archetypes, the value of its name attribute is assigned. Thus, each target graph State vertex is named exactly as its archetype Java class.

As last part of the core task, the transitions have to be created. Therefore, we use the CreateEdgeClass operation, which creates a new edge type in the target metamodel and new edges of that type in the target graph. For this operation, the semantic expression has to result in a set of triples. In each triple, the first component specifies the archetype of the new edge to be created. The second and third component specify the archetype of the start and end vertices. Thus, a new edge of the just created type will be created in the target graph using the given archetype, and this edge starts at the vertex that is the image of the second component and ends at the vertex that is image of the third component.

```
18 CreateEdgeClass Transition from State role src to State role dst
19 <= from c1, c2: keySet(img_State),
20   callingMethod: c1 <-> {members} & {Method},
21   call: callingMethod <-> & {MethodCall}
22   with call -->{target} instanceMethod
23   and not isEmpty(call <->{next} & {MethodCall} -->{target}
24     & {Method @ thisVertex.name = "activate"})
25   reportSet tup(c1, callingMethod, c2, instanceMethod), c1, c2 end
26 where instanceMethod := theElement(c2 <-> {members}
27   & {Method @ thisVertex.name = "Instance"});
```

The operation call specifies that in the target metamodel, the new edge class Transition has to be created. Edges of this type may start and end only at vertices of type State. The
start vertices have the role name src and the end vertices have the role dst. The multiplicities are omitted, and thus the default multiplicity of (0,*) is used at both ends.

In the semantic expression, c1 and c2 iterate over State archetypes, i.e., source graph Class vertices extending the abstract state class. callingMethod is bound to all methods of the class bound to c1 one after the other. In turn, call is bound to every MethodCall occurring somewhere in callingMethod’s body. The + following the aggregation symbol specifies, that one or many containment edges may be traversed from the whole to the part. Finally, instanceMethod is bound to the singleton Instance() Method of c2. The predicates in the with part ensure that the method call call indeed calls the instanceMethod and that on the object returned by the call, the activate() method is invoked.

For each variable combination fulfilling the predicates, a triple is reported. The first component (the new edge’s archetype) is a tuple containing the currently activated state class c1, its method that contains the activation call (callingMethod), the new state class to be activated c2, and its instanceMethod. The archetype of the new transition’s start state is c1 and the transition leads to the state that is the image of c2.

This is all that is required for the core task. The next sections describe the creation and setting of attributes for the Transition edge class and its instances.

2.2 Extension 1: Triggers

In the first extension task, the trigger attribute of transitions should be set. As described by the task description, we have to distinguish four cases.

1. If the transition occurs in some method except run(), then that method’s name is the trigger.
2. If it occurs in a switch statement inside the run() method, then the corresponding case’s enum constant name is the trigger.
3. If it occurs in a catch block, then the caught exception’s type name is the trigger.
4. If none of the former situations apply, then the trigger should be set to the string "-
   -".

We will cover the four situations with different rules.

Non-run() methods and default value. Again, we use the CreateAttribute operation to create the trigger attribute of type String for the Transition edge class. The string "-
   -" is chosen as default value, which handles the fourth case above.

```java
CreateAttribute Transition.trigger : String = "-
   -";

<== from t : keySet(img_Transition)
with t [1].name <> "run"
reportMap t -> t [1].name end;
```

The query returns a map that assigns to every Transition archetype whose second component’s name is not “run” the value of name. When looking at the CreateEdgeClass call
for Transition in the core task, the second component of Transition archetypes are exactly the methods in which the activation of the new state occurred. Thus, the trigger is that method's name for all non-run() methods, just as requested by the task.

**Switch statements.** Because the trigger attribute has been created by the previous CreateAttribute call, we use the SetAttributes operation. It is the little brother of CreateAttributes which only works on the instance level and requires an existing attribute given by its qualified name (Transition.trigger).

```plaintext
SetAttributes Transition.trigger
<== from t: keySet(img_Transition),
    case: t[1] <-> & {Switch}<--{cases},
    cond: case <->{condition} -->{target} & {EnumConstant}
    with t[1].name = "run"
    and case <-- & {MethodCall} -->{target} t[3]
    reportMap t -> cond.name end;
```

The query iterates over Transition archetypes (4-tuples) using the variable t. case is bound to all Case vertices reachable by diving into the body of the activating method referenced by t[1], reaching some Switch vertex, and selecting its Case vertices one after the other. In turn, cond is bound to the EnumConstant that is the condition of the case. The predicates in the with part ensure that the activation of the next state occurs in the run() method, and that the call of the Instance() method (the fourth component in the transition archetype tuples) indeed occurs in the body of case.

**Catch blocks.** Again, the SetAttributes operation is used.

```plaintext
SetAttributes Transition.trigger
<== from t: keySet(img_Transition),
    catch: t[1] <-> & {CatchBlock}
    with t[1].name = "run"
    and catch <-- & {MethodCall} -->{target} t[3]
    reportMap t -> theElement(catch -->{parameter}
    -->{typeReference}
    -->{classifierReferences}
    -->{target}).name end;
```

We iterate over Transition archetypes using the variable t. From the method containing the activation call (t[1]), we iterate all CatchBlock vertices contained somewhere in its body. The with part ensures the activation is inside the run() method and the activation call occurs inside the catch block. For all combinations of t and catch where the predicates hold, a mapping is added to the result map that assigns to the transition archetype t the name of the caught exception.
2.3 Extension 2: Actions

The second extension task deals with setting the action attribute of Transition edges. According to the task description, the value of this attribute is the name of the enumeration constant provided as argument to a send() method call appearing in the same block where the activation of the next state occurs. If there is no call to send() the attribute should be set to "--".

Again, we use the CreateAttribute operation to create the new action attribute of type String for the edge class Transition with the default value "--".

```gretl
CreateAttribute Transition . action : String = "--"

from t : keySet(img_Transition),
    container: t[1] <<= & {StatementListContainer},
    sendCall: container <<= {statements} <<= {expression} & {MethodCall}
with not isEmpty(sendCall ---> {target})
    & {Method @ thisVertex . name = "send"})
    and container <<= {statements} <<= {expression}
    ---> {next}* & {MethodCall}-->{target} t [3]
reportMap t --> theElement(sendCall <<= {arguments}
    ---> {next} -->{target}).name end;
```

The semantic expression iterates over the Transition archetype tuples. The variable container is bound to all blocks (StatementListContainer) contained in the method that contains the activation call of the next state one after the other, i.e., first it is bound to the method body, then to blocks of if, catch, or catch statements. The variable sendCall is in turn bound to all MethodCall vertices contained in the container.

The predicates in the with part of the query ensure that sendCall is in fact a call to a Method with name set to “send”. The second predicate ensures, that in the block container also the activation of the next state occurs, i.e., there is a MethodCall to the Instance() method of the state to be activated ($[3]$).

The query reports a map assigning to the selected Transition archetypes the name of the argument given to the send() method in sendCall.

This was the last operation of the ExtractStateMachines.gretl transformation. The complete task could be solved with only 57 lines of transformation source code.

3 Conclusion

In this paper, the GReTL reference solution of the program understanding case has been discussed in details. In this conclusion, we try to make some statements about the evaluation criteria.

The solution covers the core as well as both extension tasks, so it is complete.

With respect to consiseness, we think that 57 lines of transformation source code for this non-trivial task is pretty good.
We also think, that it is quite understandable once one has grasped GReTL’s traceability concept of archetypes and images.

The correctness of the solution has been validated with the three provided input models. Transforming them always results in the same target model, except that the (irrelevant) order of elements differs between models.

Moreover, the solution has a very good performance. On a usual three years old laptop, transforming the two smaller models takes between one and 1.3 seconds. Transforming the big model consisting of over 2 million vertices and edges still takes only 1.8 seconds. So although that model is about 150 times larger than the smaller models, it has nearly no impact on the runtime of the transformation. This is due to the fact that TGraphs are highly optimized for traversal speed. Likewise, the GReQL querying language used in the semantic expressions is extremely efficient due to a query optimizer component.

One last advantage of GReTL, although not part of the evaluation criteria, is that it doesn’t need a big surrounding framework and provides a nice command line interface. This allows for running transformations as batch jobs, just like demonstrated with the run.sh driver script.

References


