The Fulib Solution to the TTC 2021 Laboratory Workflow Case

Sebastian Copei, sco@uni-kassel.de Adrian Kunz, a.kunz@uni-kassel.de
Albert Zündorf, zuendorf@uni-kassel.de

Kassel University

1 Introduction

This paper outlines the Fulib [ZCD+19, ful] solution to the Laboratory Workflow Case of the Transformation Tool Contest 2021 [ttc]. Our analysis of the use case showed that it provides quite a number of different model elements that require individual treatment but the different cases are relatively simple. However, some parts of the predefined EMF metamodels do not work very well with the Fulib modeling approach. For example, the predefined metamodel uses index numbers to identify the tips of a liquid transfer job and these index numbers need to be mapped to the barcodes of the samples that are transported by a tip. Similarly, samples need to be mapped to cavities on micro-plates. Thus, we took the liberty to adapt the given metamodels by adding explicit associations between samples and some labware elements, cf. Fig. 1. Note, these adaptions connect elements from the source and from the target metamodel of our model to model transformation. For these adaptions, we loaded and combined the two given Ecore metamodels of the use case into the Fulib code generator and then did some manual modifications using Fulibs metamodeling API. Due to a misinterpretation, we also changed the cardinality of the previous-next association for *Job*s from many-to-many to one-to-one. We felt this meets the semantics of the use case, too, and it resulted in a somewhat simpler model that can be processed easier and faster.

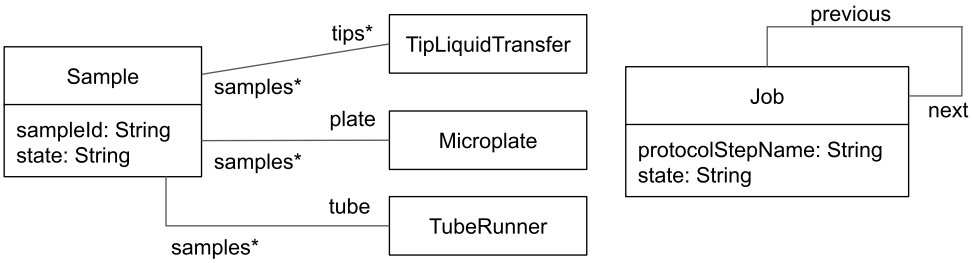

Figure 1: Design

The rest of the paper outlines the implementation of the different model processing steps and we conclude with some measurements.

2 Initialization and Loading

The initialization phase allows to load the metamodels and transformations. In our approach, Fulib generates a very light weight implementation of our model in Java code and Fulib generates a number of dedicated *Table* classes that enable efficient OCL [CG12] like queries. The actual model transformations are coded in Java against the generated model API. Thus, the Fulib solution has no initialization phase.

Copyright held by the author(s).

In: A. Garcia-Dominguez, G. Hinkel, A. Boronat, and F. Krikava (eds.): Proceedings of the 13th Transformation Tool Contest, on-line, 17-07-2020, published at http://arxiv.org

The various input models that describe a *JobRequest*, its *Assay*, and the target *Sample*s are given as EMF/XMI files (*/initial.xmi). We load the initial model with a generic XML parser and a DOM tree visitor, that builds the model based on our light weight model implementation.

3 Creating the initial JobCollection

Once the *JobRequest* and *Assay* are loaded, we use an *AssayToJobs* visitor [Gam95] to generate the initial *JobCollection*, cf. Listing 1. Using the visitor pattern allows for a nice seperation of model queries that look up elements and of tranformation rules that do the actual operations.

The *initial* method of our *AssayToJobs* visitor first creates the target *JobCollection* (cf. line 6 of Listing 1). Then it iterates through the samples, reagents, and asssay steps and calls appropriate assign rules (cf. lines 7 to 10).

The *assignToTube* rule checks, whether we have a *TubeRunner* that still has place for the new sample (cf. line 16). If not, a new *TubeRunner* is created (cf. line 17 to 20) and added to the *JobCollection* (cf. line 19). Then, the sample's barcode is added to the *TubeRunner* (cf. line 22) and in addition, we connect the sample to the *TubeRunner* for simple reference (cf. line23). The rules *assignToPlate* and *assignToTrough* work quite similarly.

```
1  public class AssayToJobs {
2     private JobCollection jobCollection;
3     private JobRequest jobRequest;
4     public JobCollection initial(JobRequest jobRequest) {
5        this.jobRequest = jobRequest;
6        jobCollection = new JobCollection();
7        jobRequest.getSamples().forEach(this::assignToTube);
8        jobRequest.getSamples().forEach(this::assignToPlate);
9        jobRequest.getAssay().getReagents().forEach((this::assignToTrough));
10       jobRequest.getAssay().getSteps().forEach(this::assignJob);
11       return jobCollection;
12    }
13    TubeRunner tube = null;
14    int tubeNumber = 1;
15    private void assignToTube(Sample sample) {
16       if (tube == null || tube.getBarcodes().size() == 16) {
17          tube = new TubeRunner();
18          tube.setName(String.format("Tube%02d", tubeNumber)
19                .setJobCollection(jobCollection);
20          tubeNumber++;
21       }
22       tube.withBarcodes(sample.getSampleID());
23       tube.withSamples(sample);
24    }
25    ...
```

Listing 1: Initial JobCollection via AssayToJobs Visitor

Listing 2 shows the handling of assay *ProtocolStep*s. As there are different types of *ProtocolStep*s we use a map of *stepAssignRules* that provides a special assign rule for each kind of step (cf. line 3, 6, 11 to 16). As an example, *ProtocolStep*s of type *DistributeSample* are handled by rule *assignLiquidTransferJob4Samples* (cf. line 19 to 22). Rule *assignLiquidTransferJob4Samples* just iterates through all samples and calls rule *assignTipLiquidTransfer*. Rule *assignTipLiquidTransfer* ensures that a *LiquidTransferJob* is available (cf. line 26 to 34). Then, lines 36 to 42 create the corresponding *TipLiquidTransfer* and initialize the corresponding attributes. Note, line 42 connects the *TipLiquidTransfer* to its sample for easy reference. The remaining *stepAssignRule*s work similar.

```
1  public class AssayToJobs {
2     ...
3     Map<Class, Consumer<ProtocolStep>> stepAssignRules = null;
4     private void assignJob(ProtocolStep protocolStep) {
```

```
5         initStepAssignRules();
6         Consumer<ProtocolStep> rule = stepAssignRules.get(protocolStep.getClass());
7         rule.accept(protocolStep);
8      }
9      private void initStepAssignRules() {
10        if (stepAssignRules == null) {
11           stepAssignRules = new LinkedHashMap<>();
12           stepAssignRules.put(DistributeSample.class,
13                               this::assignLiquidTransferJob4Samples);
14           stepAssignRules.put(Incubate.class, this::assignIncubateJob);
15           stepAssignRules.put(Wash.class, this::assignWashJob);
16           stepAssignRules.put(AddReagent.class, this::assignAddReagentJob);
17        }
18     }
19     private void assignLiquidTransferJob4Samples(ProtocolStep protocolStep) {
20        jobRequest.getSamples().forEach(
21           sample -> assignTipLiquidTransfer(protocolStep, sample));
22     }
23     LiquidTransferJob liquidTransferJob = null;
24     private void assignTipLiquidTransfer(ProtocolStep protocolStep, Sample sample) {
25        DistributeSample distributeSample = (DistributeSample) protocolStep;
26        if (liquidTransferJob == null || liquidTransferJob.getTips().size() == 8) {
27           liquidTransferJob = new LiquidTransferJob();
28           liquidTransferJob.setProtocolStepName(protocolStep.getId())
29                 .setState("Planned")
30                 .setJobCollection(jobCollection)
31                 .setPrevious(lastJob);
32           lastJob = liquidTransferJob;
33           liquidTransferJob.setSource(sample.getTube())
34                 .setTarget(sample.getPlate());
35        }
36        TipLiquidTransfer tip = new TipLiquidTransfer();
37        tip.setSourceCavityIndex(sample.getTube().getSamples().indexOf(sample))
38              .setVolume(distributeSample.getVolume())
39              .setTargetCavityIndex(sample.getPlate().getSamples().indexOf(sample))
40              .setStatus("Planned")
41              .setJob(liquidTransferJob)
42              .setSample(sample);
43     }
44     ...
```

Listing 2: Initial JobCollection via AssayToJobs Visitor

4 Reading Changes to Job Executions and Propagate

Updating is done via our *Update* class, cf. Listing 3. Updates are described by text lines in predefined files. Our *update* method calls method *updateOne* for each line (cf. line 7 and line 14 to 23). Basically, there are two kinds of updates, updates that effect a whole *Microplate* and updates that effect individual *Sample*s and *TipLiquidTransfer*s. Microplate related updates are handled by rule *updateJob* (cf. line 19 and 24 to 32). Rule *updateJob* uses FulibTable code generated for model specific queries. Line 26 creates a *JobCollectionTable* that has one row and one column containing the current *JobCollection*. Line 27 does a natural join with the *JobCollection* and its attached labware, i.e. we get a table with rows for each pair of *JobCollection* and *Labware*. Line 28 removes all rows that do not refer to a *Microplate*. Then, line 29 expands our table to *Job*s attached to the *Microplate*s, i.e. we get rows for all possible triples of *JobCollection*, *Microplate*, and attached *Job*s. Line 30 filters for *Job*s with the right *stepName*. For each resulting row, line 31 assigns the new state to the

corresponding *Job*.

Note, our *JobCollectionTable* query could also be expressed e.g. using the Java streams API. While using the Java stream API is quite comparable, the Java stream API requires some more steps and some extra operations like *flatMap* and probably some extra type casts. Thus, we prefer our FulibTables as we consider FulibTables queries to be more concise.

```
1  public class Update {
2     private JobCollection jobCollection;
3     public void update(JobCollection jobCollection, String updates) {
4        this.jobCollection = jobCollection;
5        String[] split = updates.split("\n");
6        for (String line : split) {
7           updateOne(line.trim());
8        }
9        new JobCollectionTable(jobCollection)
10             .expandJobs("job")
11             .filter(j -> j.getState().equals("Planned"))
12             .forEach(job -> removeObsoleteJob(job));
13    }
14    private void updateOne(String change) {
15       String[] split = change.split("_");
16       String stepName = split[0];
17       String states = split[2];
18       if (states.length() == 1) {
19          updateJob(states, stepName);
20       } else {
21          updateSamplesAndTips(stepName, states);
22       }
23    }
24    private void updateJob(String states, String stepName) {
25       String jobState = states.equals("S") ? "Succeeded" : "Failed";
26       new JobCollectionTable(jobCollection)
27             .expandLabware("plate")
28             .filterMicroplate()
29             .expandJobs("job")
30             .filter(j -> j.getProtocolStepName().equals(stepName))
31             .forEach(job -> job.setState(jobState));
32    }
33    private void updateSamplesAndTips(String stepName, String states) {
34       new JobCollectionTable(jobCollection)
35             .expandLabware("plate")
36             .filterMicroplate().expandSamples("sample")
37             .forEach(sample -> updateOneSampleAndTip(sample, states, stepName));
38    }
39    private void updateOneSampleAndTip(Sample sample, String states, String stepName) {
40       JobRequest jobRequest = sample.getJobRequest();
41       int index = jobRequest.getSamples().indexOf(sample);
42       char state = index >= states.length() ? 'F' : states.charAt(index);
43       if (state == 'F') {
44          sample.setState("Error");
45       }
46       TipLiquidTransfer tip = new SampleTable(sample)
47             .expandTips("tip")
48             .filter(t -> t.getJob().getProtocolStepName().equals(stepName))
49             .get(0);
```

```
50          if (state == 'S') {
51              tip.setStatus("Succeeded");
52              LiquidTransferJob job = tip.getJob();
53              tip.getJob().setState("Succeeded");
54          } else {
55              tip.setStatus("Failed");
56              LiquidTransferJob job = tip.getJob();
57              if (job.getState().equals("Planned")) {
58                  job.setState("Failed");
59              }
60          }
61      }
62      private void removeObsoleteJob(Job job)
63      {
64          if (isObsolete(job)) {
65              job.setJobCollection(null);
66              if (job.getPrevious() != null) {
67                  job.getPrevious().setNext(job.getNext());
68              }
69              else {
70                  job.setNext(null);
71              }
72          }
73      }
74      private boolean isObsolete(Job job) {
75          if (job instanceof LiquidTransferJob) {
76              LiquidTransferJob transferJob = (LiquidTransferJob) job;
77              for (TipLiquidTransfer tip : transferJob.getTips()) {
78                  if ( ! tip.getSample().getState().equals("Error")) {
79                      return false;
80                  }
81              }
82              return true;
83          } else {
84              for (Sample sample : job.getMicroplate().getSamples()) {
85                  if ( ! sample.getState().equals("Error")) {
86                      return false;
87                  }
88              }
89              return true;
90          }
91      }
92  }
```

Listing 3: Updating the Jobs

Updates with dedicated new states for each sample are handled by rule *updateSamplesAndTip* (cf. line 21 and 33 to 38). Rule *updateSamplesAndTip* uses a FulibTables query to look up all samples attached to some *Microplate* attached to our *JobCollection*. For each sample we call rule *updateOneSampleAndTip* (cf. line 37 and line 39 to 61). Rule *updateOneSampleAndTip* first retrieves the result state for the current sample (cf. lines 40 to 42) and updates the sample on failure (cf. line 44). Then the FulibTables query of lines 46 to 49 retrieves the tip that handles the current sample within the current *stepName*. Lines 50 to 59 then update the state of the tip and its job.

Once the updates are propagated, the FulibTables query of lines 9 to 12 of Listing 3 iterates through all jobs that are still *Planned* and applies rule *removeOsoleteJob* to them. LRule *removeObsoleteJob* calls *isObsolete* to check, whether the job can be removed (cf. line 64 and lines 74 to 89) and and in that case it does a classical

removal from a doubly linked list.

To be honest, our removal of obsolete jobs iterates through all jobs and thus it is not really incremental. This could be improved by collecting affected jobs during state changes and by investigating only affected jobs. However, due to the low number of jobs in the example cases, we do not believe that such a caching mechanism pulls it weight and thus we did go for conciseness.

5 Conclusions

Overall, the TTC 2021 Laboratory Workflow Case has reasonably simple queries and rules but it also has quite a number of different cases like different kinds of *Job*s and different kinds of *Labware* that all need special treatment. The Fulib solution addresses these different cases using maps of rules where appropriate rules are retrieved e.g. by the types of current objects. This allows to iterate through all tasks, very conveniently. For queries, our solution uses FulibTables, which are quite similar to Java Streams or to OCL expressions. For the actual transformations, we use plain Java code working directly on the Java implementation of our model(s).

Altogether, we consider our solution as easy to read and as quite concise, the whole update transformation needs roughly 90 lines of Java code.

In TTC 2020 the Fulib solution used transformation code working directly, with EMF based models [CZ20]. That solution was very slow. This, year we use the Fulib generated model implementation. As Figure 2 shows, the Fulib implementation uses an average of 16 megabytes of memory to handle a case while e.g. the reference solution requires an average of 46 megabytes. We believe that this reduction of memory consumption is a result of the more space efficient model implementation provided by Fulib.

Similarly, the Fulib solution seems to be quite fast: to run all phases of the test minimal case and of all scale_samples cases and of all scale_assay cases on a laptop with Intel Core i7 CPU 3.10GHz and 16 GB RAM we use a total time of about 2 seconds, the Reference solution uses about 5.2 seconds and the NMF solution coming from the central GitHub repository uses about 76 seconds. To us it seems that EMF is a performance bottleneck.

Tool	total time (milliseconds)	average memory (megabytes)
Reference	5227,95	46,45
NMF	76795,22	319,62
Fulib	1999,54	16,09

Figure 2: Measurements

Concerning correctness, we have difficulties to get the python script working that runs the checks and does the analysis of the measurements. We will try to solve these issues before the actual contest.

Concerning completeness, we did not yet implement updates that generate new samples on the fly. We just did not understand how these new samples shall be added to a running *JobCollection*: are you allowed to add new samples to an existing plate? Or does each new sample need a new plate? Or can you add samples to plates as long as those plates are not yet under processing? But when does the processing of a plate actually start? We are happy to clarify these issues and to complete our solution accordingly.

You find our solution on:

Github: https://github.com/sekassel/ttc2021fuliblabworkflow

References

[CG12] Jordi Cabot and Martin Gogolla. Object constraint language (ocl): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems*, pages 58–90. Springer, 2012.

[CZ20] Sebastian Copei and Albert Zündorf. The fulib solution to the ttc 2020 migration case. *arXiv preprint arXiv:2012.05231*, 2020.

[ful] Fulib web service. https://www.fulib.org/.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[ttc] Ttc2021 case: Incremental recompilation of laboratory workflows. https://www.transformation-tool-contest.eu/2021_labflows.pdf. Last viewed 25.05.2021.

[ZCD+19] Albert Zündorf, Sebastian Copei, Ira Diethelm, Claude Draude, Adrian Kunz, and Ulrich Norbisrath. Explaining business process software with fulib-scenarios. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 33–36. IEEE Computer Society, 2019.