The Fulib solution to the TTC 2021 laboratory workflow case

Sebastian Copei[1], Adrian Kunz[1] and Albert Zuendorf[1]

[1]Kassel University, Germany

Keywords
model transformation, tool presentation, java

1. Introduction

This paper outlines the Fulib [1, 2] solution to the Laboratory Workflow Case of the Transformation Tool Contest 2021 [3]. 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 `Jobs` 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.

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 [4] like queries. The actual model transformations are coded in Java against the generated model API. Thus, the Fulib solution has no initialization phase.

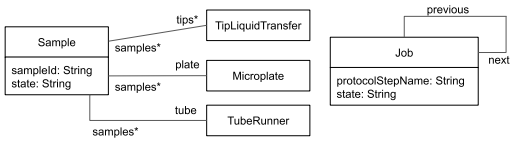

Figure 1: Design

The various input models that describe a `JobRequest`, its `Assay` and the target `Samples` 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 [5] to generate the initial `JobCollection`, cf. Listing 1.Using the visitor pattern allows for a nice separation of model queries that look up elements and of transformation 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 assay steps and calls appropriate assign rules (cf. lines 7 to 14).

The `assignToTube` rule checks, whether we have a `TubeRunner` that still has place for the new sample (cf. line 20). If not, a new `TubeRunner` is created (cf. line 22 to 26) and added to the `JobCollection` (cf. line 25). Then, the sample's barcode is added to the `TubeRunner`

TTC'21: Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 25 June 2021, ~~Bergen, Norway~~ (online).
sco@uni-kassel.de (S. Copei); a.kunz@uni-kassel.de (A. Kunz); zuendorf@uni-kassel.de (A. Zuendorf)

```
1  public class AssayToJobs {
2    private JobCollection jc;
3    private JobRequest jr;
4    public JobCollection initial(JobRequest jr) {
5      this.jr = jr;
6      jc = new JobCollection();
7      jr.getSamples()
8        .forEach(this::assignToTube);
9      jr.getSamples()
10       .forEach(this::assignToPlate);
11     jr.getAssay().getReagents()
12       .forEach(this::assignToTrough);
13     jr.getAssay().getSteps()
14       .forEach(this::assignJob);
15     return jc;
16   }
17   TubeRunner tube = null;
18   int tn = 1;
19   private void assignToTube(Sample sample) {
20     if (tube == null ||
21         tube.getBarcodes().size() == 16) {
22       tube = new TubeRunner();
23       tube
24         .setName(String.format("Tube%02d", tn))
25         .setJobCollection(jc);
26       tn++;
27     }
28     tube.withBarcodes(sample.getSampleID());
29     tube.withSamples(sample);
30   }
31    ...
```

Listing 1: Initial JobCollection via AssayToJobs Visitor

(cf. line 28) and in addition, we connect the sample to the `TubeRunner` for simple reference (cf. line 29). The rules `assignToPlate` and `assignToTrough` work quite similarly.

Listing 2 shows the handling of assay `ProtocolSteps`. As there are different types of `ProtocolSteps` we use a map of `stepAssignRules` that provides a special assign rule for each kind of step (cf. line 3, 7, 13 to 24). As an example, `ProtocolSteps` of type `DistributeSample` are handled by rule `assignLiquidTransferJob4Samples` (cf. line 30 to 32). Rule `assignLiquidTransferJob4Samples` just iterates through all samples and calls rule `assignTipLiquidTransfer`. Rule `assignTipLiquidTransfer` ensures that a `LiquidTransferJob` is available (cf. line 41 to 54). Then, lines 56 to 66 create the corresponding `TipLiquidTransfer` and initialize the corresponding attributes. Note, line 66 connects the `TipLiquidTransfer` to its sample for easy reference. The remaining `stepAssignRuleys` work similar.

```
1  public class AssayToJobs {
2    ...
3    Map<Class, Consumer<ProtocolStep>>
4      stepAssignRules = null;
5    private void assignJob(ProtocolStep ps) {
6      initStepAssignRules();
7      Consumer<ProtocolStep> rule =
8        stepAssignRules.get(ps.getClass());
9      rule.accept(ps);
10   }
11   private void initStepAssignRules() {
12     if (stepAssignRules == null) {
13       stepAssignRules = new LinkedHashMap<>();
14       stepAssignRules
15         .put(DistributeSample.class,
16           this::assignLiquidTransferJob4Samples);
17       stepAssignRules
18         .put(Incubate.class,
19           this::assignIncubateJob);
20       stepAssignRules
21         .put(Wash.class, this::assignWashJob);
22       stepAssignRules
23         .put(AddReagent.class,
24           this::assignAddReagentJob);
25     }
26   }
27   private void
28     assignLiquidTransferJob4Samples
29       (ProtocolStep protocolStep) {
30     jobRequest.getSamples().forEach(
31       sample -> assignTipLiquidTransfer
32         (protocolStep, sample));
33   }
34   LiquidTransferJob liquidTransferJob = null;
35   private void
36     assignTipLiquidTransfer
37       (ProtocolStep protocolStep, Sample sample)
38   {
39     DistributeSample distributeSample =
40       (DistributeSample) protocolStep;
41     if (liquidTransferJob == null ||
42       liquidTransferJob.getTips().size() == 8) {
43       liquidTransferJob =
44         new LiquidTransferJob();
45       liquidTransferJob
46         .setProtocolStepName(
47           protocolStep.getId())
48         .setState("Planned")
49         .setJobCollection(jobCollection)
50         .setPrevious(lastJob);
51       lastJob = liquidTransferJob;
52       liquidTransferJob
53         .setSource(sample.getTube())
54         .setTarget(sample.getPlate());
55     }
56     TipLiquidTransfer tip =
57       new TipLiquidTransfer();
58     tip.setSourceCavityIndex
59       (sample.getTube()
60         .getSamples().indexOf(sample))
61       .setVolume(distributeSample.getVolume())
62       .setTargetCavityIndex(sample.getPlate()
63         .getSamples().indexOf(sample))
64       .setStatus("Planned")
65       .setJob(liquidTransferJob)
66       .setSample(sample);
67   }
68   ...
```

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 `Samples` and `TipLiquidTransfers`. 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 27 creates a `JobCollectionTable` that has one row and one column containing the current `JobCollection`. Line 28 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 29 removes all rows that do not refer to a `Microplate`. Then, line 30 expands our table to `Jobs` attached to the `Microplates`, i.e. we get rows for all possible triples of `JobCollection`, `Microplate`, and attached `Jobs`. Line 31 filters for `Jobs` with the right `stepName`. For each resulting row, line 32 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.

Updates with dedicated new states for each sample are handled by rule `updateSamplesAndTip` (cf. line 21 and 34 to 42). 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 40 and line 43 to 65). Rule `updateOneSampleAndTip` first retrieves the result state for the current sample (cf. lines 45 to 47) and updates the sample on failure (cf. line 50). Then the FulibTables query of lines 52 to 56 retrieves the tip that handles the current sample within the current `stepName`. Lines 57 to 65 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. Rule `removeObsoleteJob` calls `isObsolete` to check, whether the job can be removed (cf. line 70 and lines 79 to 96) and and in that case it does a classical removal from a doubly linked list.

To be honest, our removal of obsolete jobs iterates

```
1  public class Update {
2    private JobCollection jc;
3    public void update(JobCollection jc, String updates) {
4      this.jc = jc;
5      String[] split = updates.split("\n");
6      for (String line : split) {
7        updateOne(line.trim());
8      }
9      new JobCollectionTable(jc)
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") ?
26       "Succeeded" : "Failed";
27     new JobCollectionTable(jc)
28       .expandLabware("plate")
29       .filterMicroplate()
30       .expandJobs("job")
31       .filter(j -> j.getProtocolStepName().equals(stepName))
32       .forEach(job -> job.setState(jobState));
33   }
34   private void updateSamplesAndTips
35     (String stepName, String states) {
36     new JobCollectionTable(jc)
37       .expandLabware("plate")
38       .filterMicroplate().expandSamples("sample")
39       .forEach(sample ->
40         updateOneSampleAndTip
41           (sample, states, stepName));
42   }
43   private void updateOneSampleAndTip
44     (Sample sample, String states, String stepName) {
45     JobRequest jobRequest = sample.getJobRequest();
46     int index = jobRequest.getSamples().indexOf(sample);
47     char state = index >= states.length() ?
48       'F' : states.charAt(index);
49     if (state == 'F') {
50       sample.setState("Error");
51     }
52     TipLiquidTransfer tip = new SampleTable(sample)
53       .expandTips("tip")
54       .filter(t ->
55         t.getJob().getProtocolStepName().equals(stepName))
56       .get(0);
57     if (state == 'S') {
58       tip.setStatus("Succeeded");
59       LiquidTransferJob job = tip.getJob();
60       tip.getJob().setState("Succeeded");
61     } else {
62       tip.setStatus("Failed");
63       LiquidTransferJob job = tip.getJob();
64       if (job.getState().equals("Planned")) {
65         job.setState("Failed");
66       }
67     }
68   }
69   private void removeObsoleteJob(Job job) {
70     if (isObsolete(job)) {
71       job.setJobCollection(null);
72       if (job.getPrevious() != null) {
73         job.getPrevious().setNext(job.getNext());
74       } else {
75         job.setNext(null);
76       }
77     }
78   }
79   private boolean isObsolete(Job job) {
80     if (job instanceof LiquidTransferJob) {
81       LiquidTransferJob transferJob =
82         (LiquidTransferJob) job;
83       for (TipLiquidTransfer tip : transferJob.getTips()) {
84         if (!tip.getSample().getState().equals("Error")) {
85           return false;
86         }
87       }
88       return true;
89     } else {
90       for
91         (Sample sample : job.getMicroplate().getSamples()) {
92         if (!sample.getState().equals("Error")) {
93           return false;
94         }
95       }
96       return true;
97     }
98   }
99 }
```

Listing 3: Updating the Jobs

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. Results

In TTC 2020 the Fulib solution used transformation code working directly, with EMF based models [6]. That solution was very slow. This, year we use the Fulib generated model implementation. As Table 1 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 (millisec)	avg. memory (mb)
Reference	5227,95	46,45
NMF	76795,22	319,62
Fulib	1999,54	16,09

Table 1
Measurements

Concerning correctness, we have had difficulties to get the Python script working that runs the checks and does the analysis of the measurements. Our solution has been developed on Windows and the Python installations we tried did not work. A Docker image with the correct Python version and the correct libraries would have been a great help. Concerning completeness, we did not 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? You find our solution on:

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

6. 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 `Jobs` 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.

The TTC 2021 Laboratory workflow Case is using a lot of EMF. EMF is the de-facto standard for model exchange these days. However, as we have discussed compared to our implementation, EMF has some serious performance problems. Using our own implementation (i.e. the Fulib code generator) provides us with a more efficient model implementation, however, we have to implement EMF readers and writers ourselves and we still fight some compatibility issues. These compatibility issues are another reason for our integration into the Python based test framework: we need to write EMF files that are not just EMF compatible but that are accepted by the test framework. We will improve our performance on EMF compatibility for next years TTC.

References

[1] A. Zündorf, S. Copei, I. Diethelm, C. Draude, A. Kunz, U. Norbisrath, Explaining business process software with fulib-scenarios, in: 2019 34th International Conference on Automated Software Engineering Workshop (ASEW), IEEE Computer, 2019, pp. 33–36.

[2] fulib, Fulib web service, https://www.fulib.org/, 2019.

[3] ttc2021labworkflow, Ttc2021 case: Incremental recompilation of laboratory workflows, https://www.transformation-tool-contest.eu/2021_labflows.pdf, 2021. Last viewed 25.05.2021.

[4] J. Cabot, M. Gogolla, Object constraint language (ocl): a definitive guide, in: International school on formal methods for the design of computer, communication and software systems, Springer, 2012, pp. 58–90.

[5] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[6] S. Copei, A. Zündorf, The fulib solution to the ttc 2020 migration case, arXiv preprint arXiv:2012.05231 (2020).