Incremental Recompilation of Laboratory Workflows

Georg Hinkel
georg.hinkel@tecan.com
Tecan Software Competence Center GmbH
Wiesbaden, Germany

ABSTRACT

When executing high-level process models, errors sometimes need to be handled by changing the planned process on the fly, leaving the jobs that have already been executed intact. Therefore, transformations that transform such a high-level process model to low-level jobs need to be incremental and respect low-level job elements that represent actions that have already happened. As an example, we consider the automation of laboratory workflows where failures of low-level jobs may need to get compensated by taking out samples from the automated process. Based on this scenario, we present a benchmark for transformation tools to deal with such kind of problems.

CCS CONCEPTS

• **Software and its engineering → Object oriented frameworks**; **Specialized application languages**; *API languages.*

KEYWORDS

incremental, model-driven, transformation

1 INTRODUCTION

When the pandemic situation due to Covid-19 started in spring 2020, the availability of test capacities was a huge problem and is still a critical problem when the paper was written. As one of the reasons, this is due to the fact that workflows executed in laboratories are usually automated in an inflexible manner or not automated at all. In the latter case, this was often not due to the fact that laboratories were lacking instruments capable of automating a workflow such as Covid-19 tests, but rather to the fact that it was hard to repurpose these instruments for Covid-19 tests, besides to verification and validation efforts also due to the lack of flexibility of the control software for such instruments.

These instruments are often robotic liquid handlers (RLHs) equipped with some additional devices such as thermocyclers, shakers or readers. These RLHs come either tied and optimized for a set of specific applications (especially clinical applications) or as flexible instruments that can solve various tasks. An example of the latter is the Fluent instrument from Tecan[1]. These instruments currently offer the end-user a domain-specific language to create and manage programs that can be executed by the robot, in the case of Fluent called scripts. This domain-specific language is modular, such that it can be extended by certain elements, e.g. to support the integration of third-party devices that can perform subtasks such as shaking or heating a plate.

The general problem with scripting is that the level of abstraction provided to the user, which usually is a chemist or biologist but not typically not a computer scientist, is often rather low. This makes it hardly accessible, unless users get a substantial amount of training. Furthermore, it is very hard for these low-level descriptions to cope with multiple scales, such as multiple supported amounts of samples processed by the robot at a time. To overcome this problem, there are several approaches to raise the level of abstraction in which the laboratory workflow is formulated and how it is transformed to low-level jobs executable by the robot.

When following such an approach, one quickly runs into problems that when errors occur, such as e.g. consumables or reagents run empty during the process. Errors usually occur at the lower levels of abstraction such as e.g., the hardware, and it is up to the automation system to translate this error into the high-level workflow description by tracing back into the high-level description of the process.

RLHs are expected to recover from errors whenever possible, e.g. by asking the user to refill the consumables and continue. Depending on the process, there may be implications on the process, for example because time constraints can no longer be met. In these situations, multiple options exist, depending on both the error and its context. If viable, a solution can be to just retry or to flag the sample that time constraints could not be met. Depending on the length of the pause and the process, this can already mean that the results for these samples are not usable, e.g. because reagents evaporated while the users was refilling consumables. In such cases, one would want to simply skip the processing for the affected samples in order to save consumables and reagents or one wants to abort processing for the samples not yet processed in order to save the sample. Another example are redundant devices that run into failures and one would like to use a different device instead.

Therefore, the automation system needs to do two things, one is to identify the required changes to the high-level process and the second one is to propagate these changes to low-level elements. However, since the error occured while the workflow was executing, some elements of the low-level model cannot be changed anymore since represent actions performed in the past. It is impossible to undo what the instrument has already done, but it is possible to change what the instrument shall be doing next. Because time constraints are in operation while these changes are propagated, it is also important that the replanning happens fast.

Given the recent advances in incremental model transformation and analysis, we believe a viable solution approach would to design the transformation from the high-level process model to low-level execution jobs as an incremental transformation. Ideally, the incrementalization is done implicitly, i.e. without the transformation developer having to adopt the transformation, to support a wide range of changes in the process model. When the transformation would create jobs, it would create them with an initial execution state and then restrict the change propagation to those elements that are still in initial state.

[1] https://lifesciences.tecan.com/fluent-laboratory-automation-workstation

Another way to look at the problem is to understand it as a consistency problem: For each sample and for each part of the input process, there must be a corresponding low-level action element. When such a low-level element fails, the corresponding sample would fail (if there is no alternative recovery). Setting the sample to a failed state would let the consistency mechanism remove all the low-level elements that have not been executed yet, ideally reporting those that have been executed as inconsistencies (because they would not have been necessary given that the processing of the corresponding sample failed).

For a practitioner, it is not clear to us which incremental transformation approaches are capable of specifying such constraints, i.e. that change propagation of some elements is restricted to some specific states. Therefore, this paper presents a transformation challenge and benchmark how such change propagation systems can be specified and executed in an incremental manner.

The remainder of this paper presents a tool challenge to solve a minimized version of this problem. At first, we describe the challenges that we see in this case in Section 2. Section 3 then describes the case itself. Section 4 introduces the benchmark framework and Section 5 briefly explains the reference solution and its deficits.

2 CHALLENGES

In particular, the following aspects of the problem are especially important:

- Because the transformation from high-level process models to low-level execution jobs happens at runtime, the performance of this transformation is important. However, it is sufficient when the jobs are clear when they have to be executed, i.e. it does not matter if the planning is not finished for the entire jobs as long as it is clear what the instrument shall be doing next. *How long does it take for the transformation to return the first low-level element?*
- If a job fails, one needs to identify how the process model needs to be adapted. *How to specify this in an understandable way?*
- Given a change of the process model, one needs to adjust the jobs that need to be performed as quick as possible. *How long does it take?*
- While propagating changes to the low-level jobs, it is tremendously important that the jobs that have been executed already stay in place. *How to make sure that the change propagation does not affect elements representing actions that have happened in the past?*
- Ideally, the transformation from the abstract process model to the low-level jobs would not have to altered. *What changes are necessary to enable an incremental change propagation?*

We ask all authors of solutions to comment on how their tool copes with the challenges described above.

3 CASE DESCRIPTION

The benchmark uses two metamodels. The first is a high-level description of laboratory workflows, the second one is a low-level description of liquid handling jobs. In the scope of the benchmark, we omit all layers below the specification of what an arm should do and we also removed the scheduling problem. Therefore, the transformation is limited to calculate *what* the RLH has to do, but not *when* or *where*.

In the remainder of this section, we present these two metamodels in Sections 3.1 and 3.2. Then, we discuss the transformation rules in Section 3.3 and the change propagation rules in Section 3.4.

3.1 The high-level laboratory process metamodel

The minimized metamodel of laboratory workflows that we use for the benchmark is shown in Figure 1. The central element is a JobRequest that represents to process a range of samples using a given Assay. This assay element represents the actual process. Meanwhile true laboratory workflows often consist of a vast variety of operations, the benchmark only considers four possible process steps, namely distributing sample, adding a reagent, washing or incubating. Each protocol step has an id for the sake of identifying it in the remainder. Further, protocol steps carry links to their previous and next steps in order to ease analysis. Meanwhile most workflows require the replication of samples and also need to include standards such as positive and negative controls, this is also omitted in this simplified model.

Samples are identified by a sample id, which is usually the barcode on the sample. Assays have a name assigned to them. In Figure 1, they are contained in a JobRequest to make it easier to work with them, though in practice they usually stand for themselves.

3.2 The low-level job metamodel

The low-level metamodel has a viewpoint that is more concerned with the actual execution of the workflow on a RLH. Here, the execution of the entire process is represented by a JobCollection that consist of a series of Job elements. These could be LiquidTransfer, Wash or Incubate elements.

Each LiquidTransfer element represents the execution of a liquid transfer command in the RLH. Because RLHs are usually equipped with multiple pipettes, the RLH can pipette multiple cavities of a microplate or tube rack at once. Therefore, a LiquidTransfer element has up to eight TipLiquidTransfer elements that specify which wells the individual tips of the transfer operation should target. It is important that the pipettes of a pipettor usually share the X axis, which means that all source cavity indices of TipLiquidTransfer elements inside a LiquidTransfer elements may only differ by their remainder in the division by 8. Similarly, all target cavity indices must be the same except for the modulo 8. That is, it is allowed to pipette cavity indices 0 and 1 inside the same element, but not 0 and 8. Liquid can be transferred from and to any kind of labware.

We assume here that a separate device is used for washing that only works with microplates, but washes cavities separately. Therefore, a Wash element carries a list of all cavity indices that should be washed and a reference to the plate that should be washed.

Similarly, an Incubate steps works only for microplates, but it always incubates an entire plate, because it is physically difficult to heat only parts of it.

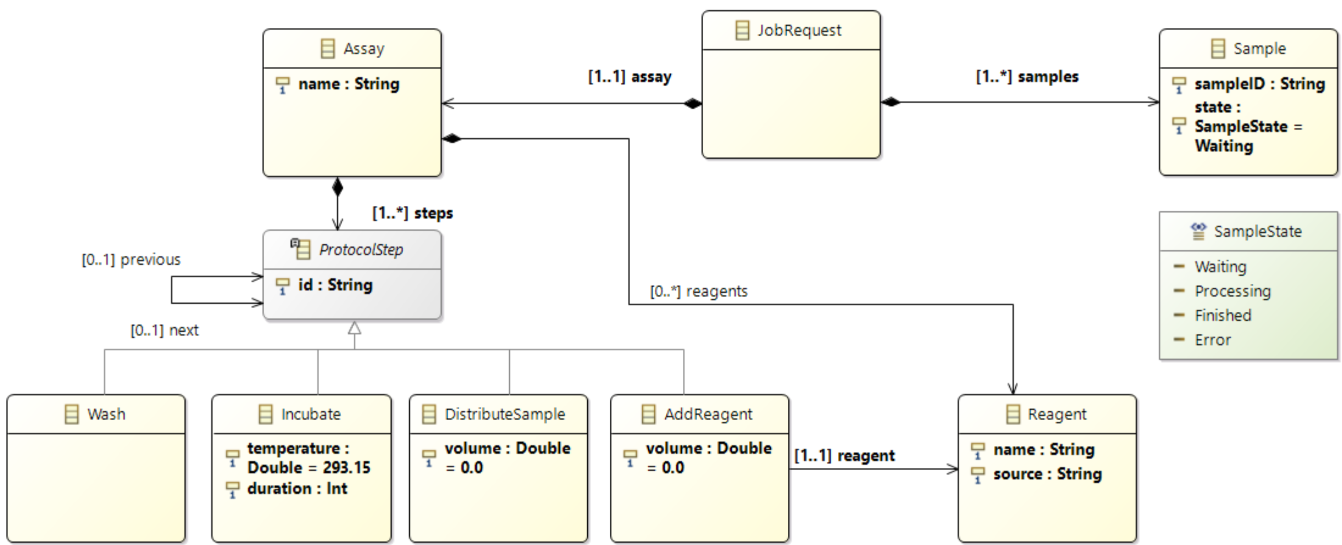

Figure 1: Minimized metamodel of laboratory workflows.

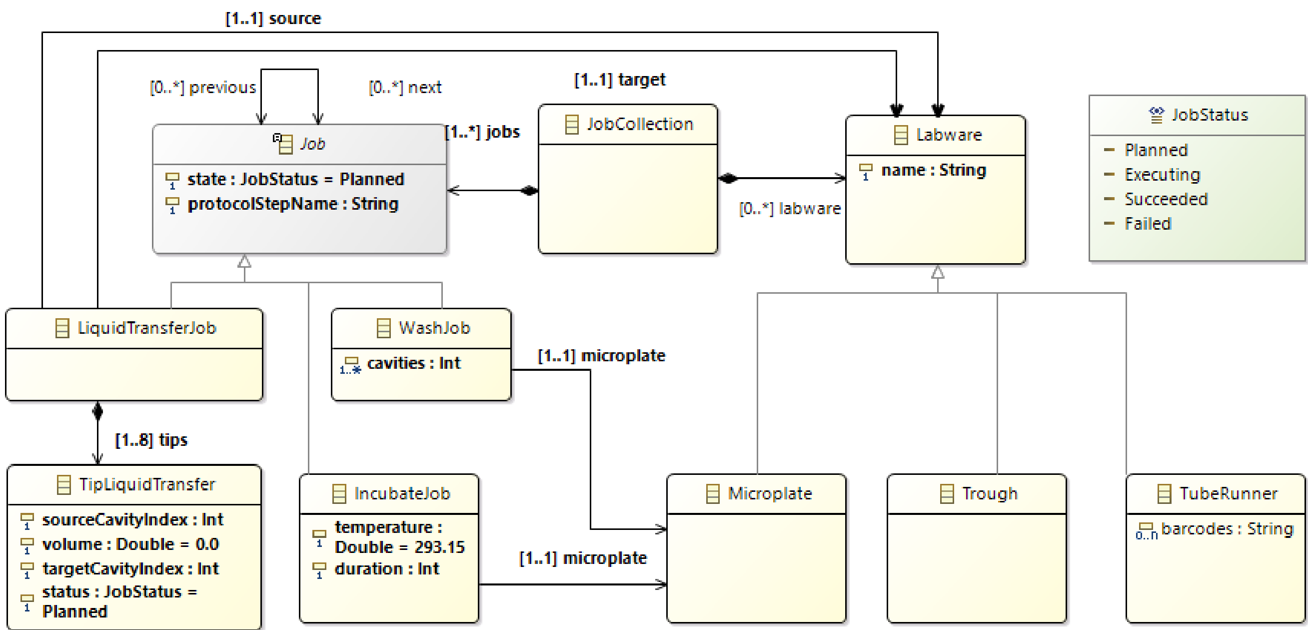

Figure 2: Minimized metamodel of liquid handling jobs

3.3 Transformation Rules

The transformation from the high-level process model to the low-level job model needs to follow these requirements:

- For each JobRequest, a JobCollection should be created.
- Samples are processed on microplates but come in tubes stored on racks with 16 tubes called tube runners. Therefore, for each Sample, a cavity of a TubeRunner should be assigned. Further, the transformation needs to store a mapping between a Sample and a combination of a Microplate and a cavity index. The combination of microplate and cavity needs to be unique. At most 96 cavities of a microplate can be used. In order for tracing, the microplates should be simply numbered with a common prefix. The first microplate should be named *Microplate1* and so on and so forth.
- We assume that reagents are stored in troughs. Therefore, for each Reagent element, there should be a Trough element with the name set to the name of the reagent.
- For each combination of ProtocolStep and Sample that is not in the Error state, a Job needs to be added to the

JobCollection corresponding to the parent JobRequest of the parent Assay as defined per the following rules:

- For each DistributeSample element, there must be a TipLiquidTransfer that represents the transfer of the sample from cavity 0 of the tube containing the sample to the cavity of the assigned microplate. Multiple TipLiquidTransfer elements may share the same parent LiquidTransfer element.
- For each AddReagent element, there must be a TipLiquidTransfer that represents the transfer of reagent to the cavity with the sample, i.e. from cavity 0 of the trough created for the reagent to the cavity and microplate assigned to the sample. Again, multiple TipLiquidTransfer elements may share a parent LiquidTransfer element if the conditions regarding the cavity indices are met.
- For each Wash, a WashJob should be created that washes the microplate assigned to the sample and at least the cavity index of the sample. However, the same WashJob may be reused for multiple samples provided that the transformation assigns the samples to the same microplate.
- For each Incubate, an IncubateJob with the same temperature and duration should be created that incubates the microplate assigned to the sample. The same IncubateJob must be reused for the incubation of the same microplate, provided that the transformation assigns them to the same microplate.
- Each created job must have the name of the protocol step from which it was created.
- Each created job must reference jobs created for the previous protocol step for the same samples such that the job is scheduled afterwards.

• The transformation should ideally produce the minimum amount of elements, i.e. the LiquidTransferJob, WashJob and IncubateJob elements should be shared where this is possible.

3.4 Change Propagation Rules

As denoted earlier, there are two change propagations in the benchmark. One is to change the high-level model as reaction to low-level changes and the other change propagation is in the direction vice versa.

The first change propagation will be very simple in the scope of this benchmark. Whenever a Job is failed, all corresponding samples have to change their state to failed as well. If a LiquidTransferJob fails, only those samples using the failed TipLiquidTransfer elements have to be set to failed.

The more interesting change propagation is the change propagation in the opposite direction. In the scope of the benchmark, we only consider changes to the state of samples. If a sample changes its state to failed (e.g. as a consequence of a change propagation from the lower level), all jobs created for this sample need to be removed, provided that they are still only planned and no other samples are affected. That is, a WashJob or IncubateJob may only be removed if all samples on the corresponding plate are failed.

4 BENCHMARK

We provide a benchmark framework that can automatically compile, run and check solutions and generate diagrams to analyze the results. The benchmark framework with the metamodels, input models, the reference solution are publicly available online at https://github.com/tecan/ttc21incrementalLabWorkflows.

In the remainder of this section, we first describe the phases of the benchmark in Section 4.1, then explain how to run it in Section 4.2. Next, we introduce the input models in Section 4.3. Section 4.4 explains the correctness checks and Section 4.5 introduces the evaluation criteria for solutions. Section 4.6 then explains the procedure to add a solution to the benchmark.

4.1 Phases

The benchmark is divided into the following phases:

(1) **Initialization**: Loading the transformation and metamodels
(2) **Load**: Loading the input models
(3) **Initial**: Creating the initial JobCollection
(4) **Update**: Reading changes to job executions and propagate

The last step is performed repeatedly 20 times. The benchmark seeks to compare execution times for all of the phases. For the last step, any efforts necessary for parsing and loading the changes can be excluded from the time measurements.

4.2 Running the benchmark

The benchmark framework only requires Python 2.7 or above and R to be installed. R is required to create diagrams for the benchmark results. Furthermore, the solutions may imply additional frameworks. We would ask solution authors to explicitly note dependencies to additional frameworks necessary to run their solutions.

If all prerequisits are fulfilled, the benchmark can be run using Python with the command `python scripts/run.py`. Additional command-line options can be queried using the option `-help`.

```
1  {
2    "Tools": ["Reference"],
3    "Scenarios": [
4    {
5      "Name": "test",
6      "Models": ["minimal"]
7    }],
8    "Sequences": 20,
9    "Runs": 5,
10   "Timeout": 6000
11 }
```

Listing 1: A minimal benchmark configuration

The benchmark framework can be configured using JSON configuration files. A minimal test configuration is depicted in Listing 1. When creating a new solution, we highly recommend to overwrite the contents of this configuration file locally. In the configuration from Listing 1, only the test scenario with just one minimal model is executed, using only the solution in `Reference` 5 times.

4.3 Input Models

As inputs, we use models of a very abstract Elisa (**E**nzyme-**l**inked **I**mmuno**s**orbent **A**ssay) workflows. These workflows are generally used to detect antibodies, for example to proof that a patient has experienced a Covid-19 infection or has been vaccinated. For this,

we assume that we have access to microplates that are already prepared with Covid antigens at the bottom of the wells. Then, we add the sample. After an incubation period, antibodies against Covid-19 in the sample, if any, will bind to the antigens at the bottom of the well. Then, the plate is washed (in order to get rid of other antibodies) and a conjugate is added. In a second incubation step, special marked antibodies bind to the Covid-19 antibodies, if any. Afterwards, the plate is washed again and a substrate is added that reacts with the marked antibodies and, after another incubation period, results in a color reaction.

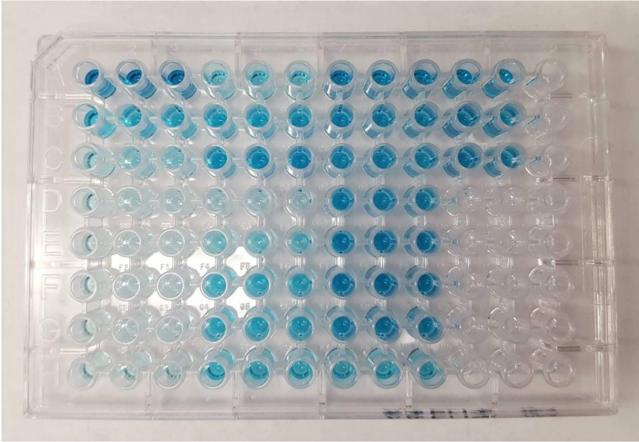

Figure 3: Example result of an Elisa process

The result is a plate where some cavities are colored while others are not, as illustrated in Figure 3. Such a plate can be read by an absorbance reader, which the limited process model does not support. In such an image, the color correlates with the amount of antibodies present in the sample.

In the scope of the benchmark, we always use the same assay, but vary the number of samples or work with duplicates of the assay. That is, the benchmark has a sequence of models with increasing number of samples but the same assay or an increasing number of steps in the assay (by replicating the steps) and a constant number of samples.

To specify changes in the low-level job model, we use text files that specify these jobs in terms of the high-level model. Each line of these text files corresponds to one state change of one job. The lines are in the format <ProtocolStepName>_<Plate>_<State> where state is either S for success or F for a failure (we skip the executing state). These status lines have been generated independently from an actual solution and thus, they include also status changes of jobs for samples that would have to be set to a failed state due to a previous failure. In case of a liquid transfer, there are 96 states, indicating the success state for each cavity, independently of whether this cavity is used or not.

However, changes may also affect the samples, i.e. new samples may be added dynamically. This is indicated by a line in the format NewSample_<SampleId>.

As stated above, the time needed to calculate the job elements that are actually affected by these changes should be excluded from the time measurements, but the actual change propagation should be included.

The high-level process models are available in EMF format, but can be made available in other formats as well, upon request.

4.4 Correctness Checks

The benchmark framework performs the following correctness checks after the initial transformation and after each update:

- It checks that only the step names of input model elements are used.
- It checks that no liquid transfer is made into the same cavity of the same labware and the same protocol name.
- It checks that no cavity index greater or equal to 96 is used.
- For the initial execution, the requirement that only the minimal amount of jobs are used is a must and the benchmark framework checks the number of elements.

4.5 Evaluation Criteria

The solutions are evaluated along the following criteria:

- Understandability
- Conciseness
- Number of elements in the low-level model
- Execution time

The understandability of the solutions will be evaluated by a poll during the TTC event. To evaluate the conciseness, we ask every solution to note on the lines of code of their solution. This shall include the model views and glue code to actually run the benchmark. Code to convert the change sequence can be excluded. For any graphical part of the specification, we ask to count the lines of code in a HUTN[2] notation of the underlying model.

The number of elements is collected from the check application and does not have to be calculated by the solutions.

4.6 Solution Requirements

The solutions are required to perform the steps of the benchmark in the order depicted above. Solutions must report the following metrics between these steps, in case of the update phase after every change sequence. The reporting is done by printing the following separated by ; to the standard output:

- **Tool**: The name of the tool.
- **Scenario**: The scenario of the models (i.e. scaling samples or assay steps)
- **Model**: The name of the input model set that is currently run
- **RunIndex**: The run index in case the benchmark is repeated
- **Iteration**: The iteration (only required for the Update phase)
- **PhaseName**: The phase of the benchmark
- **MetricName**: The name of the reported metric
- **MetricValue**: The value of the reported metric

Tool, *Scenario*, *Model* and *RunIndex* are provided to the solution using environment variables with the same name. Further, the benchmark framework passes the root directory of the models

[2] https://www.omg.org/spec/HUTN/

using the variable *ModelPath* and the number of update iterations using *Sequences*.

Solutions should report on the runtime of the respective phase in integer nanoseconds (**Time**) and the working set in bytes (**Memory**). The memory measurement is optional. If it is done, it should report on the used memory after the given phase (or iteration of the update phase) is completed. Solutions are allowed to perform a garbage collection before memory measurement that does not have to be taken into account into the times. In the update phase, we are not interested in the time to parse and identify the changes, but only the pure change propagation.

To enable automatic execution by the benchmark framework, solutions should add a subdirectory to the solutions folder of the benchmark with a *solution.ini* file stating how the solution should be built and how it should be run. Because the solution contains the already compiled reference solution, no action is required for build. However, other solutions may want to run build tools like maven in this case to ensure the benchmark runs with the latest version.

The repetition of executions as defined in the benchmark configuration is done by the benchmark. This means, for 5 runs, the specified command-line will be called 5 times, passing any required information such as the model that should be computed, the run index, etc. in separate environment variables. All runs should all have the same prerequisites. In particular, solutions must not save intermediate data between different runs. Meanwhile, all iterations of the Update phase are executed in the same process and solutions are allowed (and encouraged) to save any intermediate computation results they like, as long as the results are correct after each change sequence.

5 REFERENCE SOLUTION

The repository also contains a reference implementation that mimics how this kind of functionality would be implemented using standard object-oriented code. That is, it uses NMF [1] for the model representation but processes the models using the generated model API using only functionality offered by the .NET Framework. For this, we extended the model classes with a separate interface as needed for the transformation and created separated dedicated interfaces for tracing, which is done manually.

The following trace links are created:

- For each sample, we store the tube runner and the tube index where the sample comes into the system and the microplate and cavity where it is processed. These trace links are created in both directions, such that we can also trace a microplate cavity back to a sample.
- For each job, we trace which samples it processes, both forwards and backwards, such that we can identify the jobs for a sample and the samples processed by a job.
- For each reagent, we trace the trough in which the reagent is placed.

Essentially, the reference solution calculates the required jobs for a number of input samples separately through queries. As an example, the query for creating the liquid transfers for a AddReagent element is depicted in Listing 2.

```
1  from sample in samples
2  let location = locationRepository.LocateSampleProcessing( sample )
3  group (sample, location) by (location.Plate, location.Cavity / 8)
       into transferChunk
4  select TraceAll( AddTips( new LiquidTransferJob
5  {
6     ProtocolStepName = Id,
7     Source = locationRepository.LocateReagent( Reagent ),
8     Target = transferChunk.Key.Plate
9  }, transferChunk.Select(l => l.location.Cavity) ),
10  trace, transferChunk.Select(g => g.sample));
```

Listing 2: Generating the liquid transfers for an AddReagent in the reference solution.

```
1  Tips.Where( t => t.Status == JobStatus.Failed )
2      .Select( tip => locationRepository.IdentifySample( Target, tip
           .TargetCavityIndex ) )
3      .Where( s => s != null );
```

Listing 3: Calculating failed samples for a LiquidTransferJob

```
1  foreach(var sample in failedSamples.Distinct()) {
2    sample.State = SampleState.Error;
3    foreach(var job in _affectedJobsPerSample[sample]) {
4      if (job.State == JobStatus.Planned) {
5        job.GetProcessedSamples().Remove( sample );
6        if(job.GetProcessedSamples().Count == 0) {
7          job.Delete();
8        }
9        else if(job is LiquidTransferJob liquidTransfer) {
10         var processingLocation = _locationRepository.
               LocateSampleProcessing( sample );
11         var tip = liquidTransfer.Tips.FirstOrDefault( t => t.
               TargetCavityIndex == processingLocation.Cavity );
12         tip?.Delete();
13       }
14     }
15   }
16 }
```

Listing 4: Change propagation implementation of the reference solution

In this listing, we first obtain the processing location for each sample, group them by plate and row and create a liquid transfer job for each group, adding the tip indices and trace links in separate methods not shown in the listing. Similar queries exist also for the other high-level elements.

When changes of the high-level process model need to be propagated, the solution calculates the samples that are affected by the failures following the trace links. For LiquidTransferJobs, this is depicted in Listing 3.

Once the failed samples are identified, the reference solution removes the trace link to jobs. If a job or tip transfer does not have a referenced sample any more, it gets deleted. This is depicted in Listing 4.

We believe that the usage of the query syntax makes the reference solution actually not too bad from a readability perspective and given the fact that it is implemented in plain C#, we assume it also has a good performance. Still, the reference solution has multiple problems:

- The high-level process model elements have an explicit knowledge about their transformation to low-level job elements and in the other direction, the low-level job elements

407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464

have an explicit knowledge about which samples they are processing. This is good enough for a quick solution, but normally undesirable as the high-level model is also used in other contexts such as an editor.

- Because the tracing is done manually, it is only done on selected points. If the transformation becomes more complex, this leads to additional overhead as more trace links will become necessary.
- The change propagation is done manually, which means that only selected types of changes are actually supported. However, it is very difficult to exclude certain types of changes because there is usually still some scenario in which every part of the input changes. For example, in the scope of the benchmark, we considered the assay steps constant, but these may also change when some parts of the analysis could still be performed even if other parts are no longer possible (e.g. because reagents have run empty).
- The change propagation rules are repetitive and duplicated while ideally, they could all be summarized in the fact that Job elements should only be present either if they have already started or as long as any of the samples they process is not failed. In the reference solution, this abstract problem is encoded for each element separately and both for the high-level and the low-level model elements separately, leading to a threat of inconsistency problems.

Given the advances in model transformation, especially in incremental change propagation, we think that the problem could be solved in a better way.

REFERENCES

[1] Georg Hinkel. 2018. NMF: A Multi-platform Modeling Framework. In *Theory and Practice of Model Transformation*, Arend Rensink and Jesús Sánchez Cuadrado (Eds.). Springer International Publishing, Cham, 184–194.