The TTC 2015 Train Benchmark Case for Incremental Model Validation*

Gábor Szárnyas Oszkár Semeráth István Ráth Dániel Varró
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{szarnyas, semerath, rath, varro}@mit.bme.hu

In model-driven development of safety-critical systems (like automotive, avionics or railways), well-formedness of models is repeatedly validated in order to detect design flaws as early as possible. Validation rules are often implemented by a large amount of imperative model traversal code which makes those rule implementations complicated and hard to maintain. Additionally as models are rapidly increasing in size and complexity, efficient execution of these operations is challenging for the currently available toolchains. However, checking well-formedness constraints can be interpreted as evaluation of model queries, and the operations as model transformations, where the validation task can be specified in a concise way, and executed efficiently.

This paper presents a benchmark case and an evaluation framework to systematically assess the scalability of validating and revalidating well-formedness constraints over large models. The benchmark case defines a typical well-formedness validation scenario in the railway domain including the metamodel, an instance model generator, and a set of well-formedness constraints captured by queries and repair operations (imitating the work of systems engineers by model transformations). The benchmark case focuses on the execution time of the query evaluations with a special emphasis on reevaluations, as well as simple repair transformations.

1 Introduction

During the development of safety critical software like automotive, avionics or train control systems, different kind of models are frequently used. The goal of this approach is to develop models to assist the automated generation of various design artifacts (source code, configuration files, etc.) However, design errors of the system model invalidate the correctness of the generated artifacts, thus it is critical to check the well-formedness of such models. Additionally, it is considerably more expensive to fix design flaws in the later stage of the development, thus it is important to detect them as soon as possible by checking the well-formedness constraints repeatedly.

Model validation problems are often addressed by model transformation engines: error cases are defined by model queries, the results of which can be automatically repaired by transformation steps. In practice, this is challenging due to two factors: (i) *instance model sizes* are exhibiting a tremendous growth as the complexity of systems-under-design is increasing, (ii) the *sophistication of validation constraints* in toolchains is increasing. As a consequence, validation of industrial models is challenging or may become completely unfeasible.

To address this challenge, the Train Benchmark is a macro benchmark that aims to measure repetitive query evaluation performance. While there are a number of existing benchmarks for queries over relational databasesand triplestores, modeling tool workloads for well-formedness constraint validation

*This work was partially supported by the MONDO (EU ICT-611125) project and Red Hat Inc.

Submitted to:
TTC 2015

© G. Szárnyas et al.
This work is licensed under the
Creative Commons Attribution License.

are significantly different [2]. Specifically, modeling tools use much more complex queries than typical transactional systems, and the real world performance is more affected by response time (i.e. execution time for a specific operation such as validation or transformation) rather than throughput (i.e. the number of parallel transactions). Also, previous TTC cases did not focus on measuring the performance of query reevaluation.

The source code is available at `https://github.com/FTSRG/trainbenchmark-ttc`. This case is strongly based on the Train Benchmark [1], an ongoing benchmark project of our research group.

2 Case Description

A *benchmark case* configuration in the Train Benchmark consists of an *instance model* (Section 2.2), a *query* and a *repair transformation* (Section 3) describing constraint violating elements. As a result of a benchmark case run, the *execution times* of each phase, the *memory usage* and the *number of invalid elements* are measured and recorded. The number of invalid elements are used to check the correctness of the validation, however the collection of element identifiers must also be available for later processing.

2.1 Metamodel

The metamodel of the Train Benchmark is shown in Figure 3. A train route is defined by a sequence of sensors. Sensors are associated with track elements which are either segments (with a specific length) or switches. A route follows certain switch positions which describe the *required* state of a switch belonging to the route. Different route definitions can specify different states for a specific switch. Each route has a semaphore on its entry and exit. Figure 1 shows a typical railway network.

Every railway element is a subtype of the class RailwayElement which has a unique identifier (id). The root of the model is a RailwayContainer which contains the semaphores and the routes of the model. Additionally, the railway container has an invalids reference for storing elements. This is used for serializing EMF models (Section B.1.1).

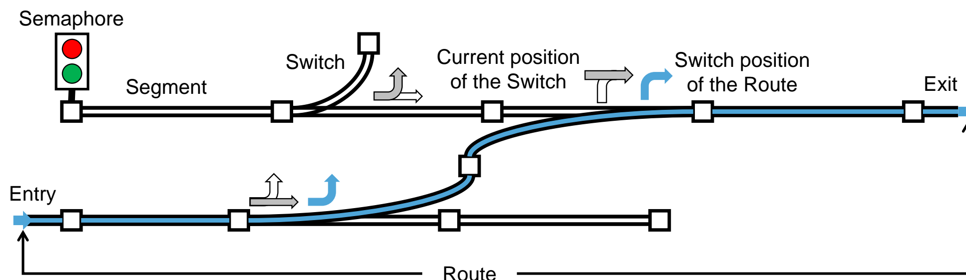


Figure 1: Illustration for the concepts in the Train Benchmark models.

2.2 Instance Models

The instance models are systematically generated for the metamodel: small model fragments are created and connected to each other. Based on the model queries, the generator injects errors to the model by removing edges and changing attribute values with a certain probability. The probability of injecting an error to violate a pattern (Section 3) is shown in Table 1.

This generation method controls the number of matches of all defined model queries. To avoid highly symmetric models, the exact number of elements and cardinalities are randomized. This brings

artificially generated models closer to real world instances and prevents query tools from abusing the artificial regularity of the model. To assess scalability, the benchmark uses instance models of growing sizes, each model containing twice as many model elements as the previous one. The instance models are designated by powers of two (1, 2, 4, 8, ...), the smallest model containing about 5000 model elements.

2.3 Benchmark Phases

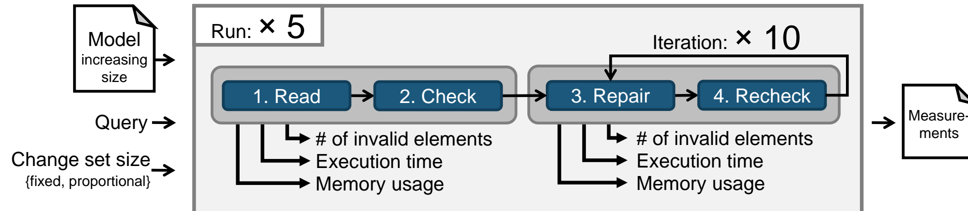


Figure 2: Phases of the benchmark.

To simulate a typical validation workload, four *phases* were defined (Figure 2).

1. During the read phase, the instance model is loaded from hard drive to memory. This includes the parsing of the input as well as initializing data structures (e.g. indexes) of the tool.
2. In the check phase, the instance model is queried to identify invalid elements. The result of this phase is a set of the invalid elements, which will be used in the next phase.
3. In the repair phase, the model is changed to simulate the effects (and measure the performance) of model modifying operations. The transformations are always performed on a subset of the model elements returned by the check phase.
4. The revalidation of the model is carried out in the recheck phase similarly to the check phase. In real-world scenarios, there are often multiple transformations in the system which may interfere with the results of the query. Because of this, we require the tools to reevaluate the query with regards to the current state of the model.

The repair operation intends to fix invalid models elements based on the invalid objects identified during the previous check or recheck phase. We defined two strategies to determine the size of the change set:

fixed 10 of invalid model elements is modified. This tests the efficiency of handling small change sets.
proportional 10% of the dresult set is modified. This tests the efficiency of handling large change sets.

2.4 Queries

The *queries* used in the validation scenario are introduced both informally and as graph patterns. In complexity, the queries range from simple attribute value checks to complex path constraints consisting of several join operations: two simple queries use at most 2 objects (PosLength and SwitchSensor) and three complex queries use 4–8 objects and multiple join operations (RouteSensor, SemaphoreNeighbor, SwitchSet).

2.4.1 Graph Patterns and Transformations

The purpose of the queries is to check well-formedness constraints by matching graph patterns looking for errors in the model. The *graph patterns* are defined by a *name*, a list of symbolic object parameters and the constraints to be satisfied by the parameters. A *pattern match* maps each symbolic parameter

to a model object, where the mapping satisfies the conditions defined by the constraints. The result of the query is the set of all possible matches. The absence of pattern matches means that the model is well-formed, and the matches of the error pattern marks the invalid elements. The *match set* contains all matches for a given pattern.

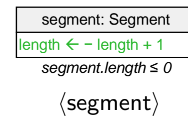
In the *repair* phase, some model elements are *deterministically selected* and repaired. In order to ensure *repeatable results*, (1) the elements for transformation are chosen using a pseudorandom generator, (2) the elements are always selected from the deterministically sorted list (Section 2.4).

3 Tasks

For each task, we present the well-formedness constraints. The *queries* are looking for violations of these constraints. We describe the meaning and the goal of each query and show a graphical notation of the associated graph pattern. We also define the matches as tuples to ensure that the ordering of the matches is consistent between the implementations (Section B.2). The *repair transformations* are represented as graph transformations. For defining the patterns and transformations, we used a graphical syntax similar to GROOVE [3] with a couple of additions:

- Filter conditions are shown in *italic* font.
- Negative application conditions are shown with in a red rectangle with the NEG caption.
- The insertions are with a «new» caption. Attribute updates are also show in green.

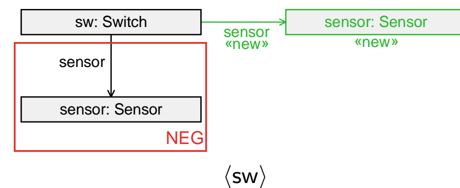
PosLength. Every segment must have a positive length.
Query. The query checks for segments with a length less than or equal to zero.



⟨segment⟩

Repair transformation. The length attribute of the segment in the match is updated to $-\text{length}+1$.
Goal. This query defines an attribute check. This is a common use case in validation scenarios.

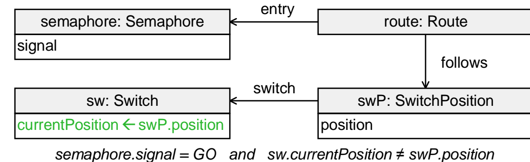
SwitchSensor. Every switch must have at least one sensor connected to it.
Query. The query checks for switches that have no sensors associated with them.
Repair transformation. A sensor is created and connected to the switch.



⟨sw⟩

Goal. This query checks whether an object is connected to a relation. This pattern is common in more complex queries, e.g. it is used in the RouteSensor and the SemaphoreNeighbor queries.

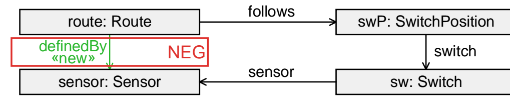
SwitchSet. The entry semaphore of a route may only show GO if all switches along the route are in the position prescribed by the route.
Query. The query checks for routes which have a semaphore that show the GO signal. Additionally, the route follows a switch position (swP) that is connected to a switch (sw), but the switch position (swP.position) defines a different position from the current position of the switch (sw.currentPosition).



⟨semaphore, route, swP, sw⟩

Repair transformation. The currentPosition attribute of the switch is set to the position of swP.
Goal. This pattern tests the efficiency of the join and filtering operations.

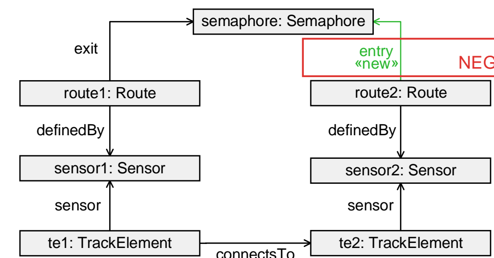
RouteSensor. All sensors that are associated with a switch that belongs to a route must also be associated directly with the same route.
Query. The query looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route.



⟨route, sensor, swP, sw⟩

Repair transformation. The missing definedBy edge is inserted by connecting the route in the match to the sensor.
Goal. This pattern checks for the absence of circles, so the efficiency of the join and the antijoin operations is tested.

SemaphoreNeighbor. Routes that are connected through sensors and track elements must belong to the same semaphore.
Query. The query checks for routes (route1) which have an exit semaphore (semaphore) and a sensor (sensor1) connected to a track element (te1). This track element is connected to another track element (te2) which is connected to another sensor (sensor2) which (partially) defines another, different route (route2), while the semaphore is not on the entry of this route (route2).
Repair transformation. The route2 node is connected to the semaphore node with an entry edge.



⟨semaphore, route1, route2, sensor1, sensor2, te1, te2⟩

Goal. This pattern checks for the absence of circles, so the efficiency of the join operation is tested. One-way navigable references are also present in the constraint, so the efficiency of their evaluation is also measured. Subsumption inference is required, as the two track elements can be switches or segments.

Acknowledgements. The authors would like to thank Benedek Izsó for originally designing and implementing the Train Benchmark, Tassilo Horn for providing valuable comments regarding both the specification of the case and the implementation of the benchmark framework, and Zsolt Kővári for his contributions in the benchmark and visualization scripts.

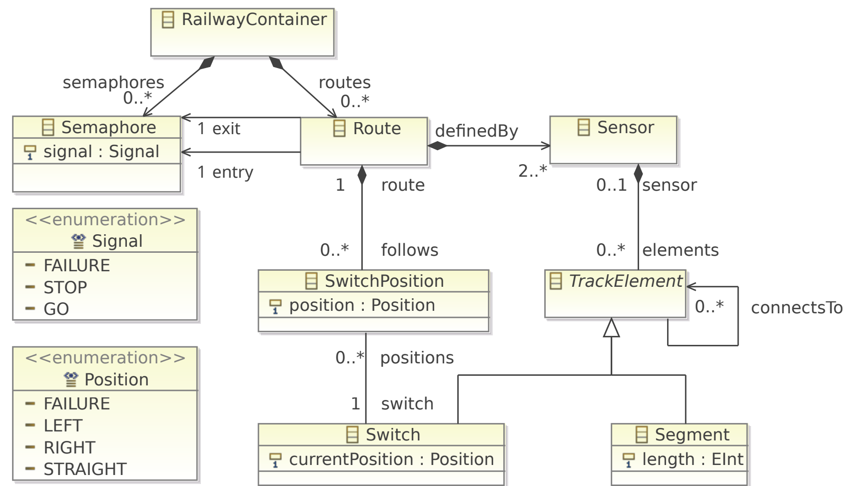
References

[1] Benedek Izsó, Gábor Szárnyas & István Ráth (2014): *Train Benchmark*. Technical Report, Budapest University of Technology and Economics.

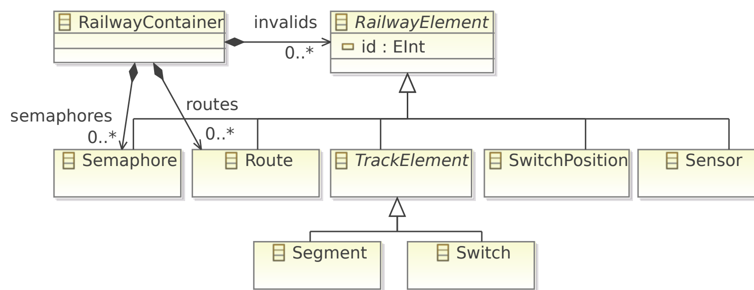
[2] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth & István Ráth (2013): *Towards Precise Metrics for Predicting Graph Query Performance*. In: *ASE 2013*, IEEE, pp. 412–431, doi:10.1109/ASE.2013.6693100.

[3] Arend Rensink (2004): *The GROOVE simulator: A tool for state space generation*. In: *Applications of Graph Transformations with Industrial Relevance*, Springer, pp. 479–485.

A Metamodel



(a) Containment hierarchy and references



(b) Supertype relations

Figure 3: The metamodel of the Train Benchmark.

B Implementation

To aid the development of case solutions, we provide a framework using predefined input and output formats, along with two reference implementations.

B.1 Instance Model Formats

B.1.1 EMF Models

The EMF models are serialized to standard XMI format using the generated EMF code. The injection of errors during the instance model generation (Section 2.2) causes some containment errors. Invalid elements violating the containment hierarchy could not be serialized. As the benchmark requires invalid models, the invalid elements are connected to the root element of the instance model by the invalids reference Figure 3b.

attribute / edge	error percentage
Segment.length	6%
Route.definedBy	10%
Route.exit	15%
Switch.sensor	35%
SwitchPosition.position	30%

Table 1: Error percentages in the generated instance model.

B.1.2 Non-EMF Models

The generator defines a graph-like interface for creating the models. The EMF model generator is an implementation of this interface. To generate non-EMF models, the following approaches are recommended: (1) either create a custom class which implements the Generator interface or (2) generate the EMF models and convert them to another representation.

B.2 Ordering of the Match Set

The matches in the match set may be returned in any collection (e.g. a list or a set) in any order, given that the collection is unique. In order to ensure that the benchmark is *repeatable*, this collection is copied to a sorted list. The sorting is carried out using by defining the ordering between matches.

To *compare* matches $M_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $M_2 = \langle b_1, b_2, \ldots b_n \rangle$, we take the first elements in each match (a_1 and b_1) and compare their identifiers. If the first elements are equal, we compare the second elements (a_2 and b_2) and so on until we find two different model elements. This is guaranteed by the fact that the collection is unique, so it cannot contain two identical matches.

For example, for the RouteSensor query, a match set may be returned by tool A as list

$$(\langle \mathsf{route}: 8, \mathsf{sensor}: 12, \mathsf{switchPosition}: 4, \mathsf{sw}: 10 \rangle; \langle \mathsf{route}: 5, \mathsf{sensor}: 1, \mathsf{switchPosition}: 13, \mathsf{sw}: 7 \rangle)$$

and by tool B as set

$$\{\langle \mathsf{route}: 5, \mathsf{sensor}: 1, \mathsf{switchPosition}: 13, \mathsf{sw}: 7 \rangle; \langle \mathsf{route}: 8, \mathsf{sensor}: 12, \mathsf{switchPosition}: 4, \mathsf{sw}: 10 \rangle\}$$

For both implementations, the framework creates a *sorted copy*, resulting in the list

$$(\langle \mathsf{route}: 5, \mathsf{sensor}: 1, \mathsf{switchPosition}: 13, \mathsf{sw}: 7 \rangle; \langle \mathsf{route}: 8, \mathsf{sensor}: 12, \mathsf{switchPosition}: 4, \mathsf{sw}: 10 \rangle)$$

The ordered list is also used to ensure that the transformations are performed on the same model elements, regardless of the return order of the match set.

B.3 Building the Projects

The Train Benchmark case defines a framework and application programming interface that enables the integration of additional tools. The reference implementation contains a benchmark suite for queries implemented in Java and EMF-INCQUERY. Both the framework and the reference implementations are written in Java 7.

For building the projects, we used Apache Maven[1], one of the most widely used Java build systems. The build is configured so that the binaries are able to run without an Eclipse application. A significant

[1] `https://maven.apache.org/`

proportion of modeling tools are integrated to the Eclipse plug-in environment. In order to support such systems, our projects also have a plug-in nature. This way, they can be integrated with Eclipse (and OSGi) plug-ins as well and can be built without Maven.

B.4 Running the Projects

The scripts can parametrized by a simple JSON configuration file which defines:

- the range of the instance models from `minSize` to `maxSize`,
- the list of queries specified (Section 2.4),
- the list of tools,
- the number of runs,
- the number of repair–recheck iterations,
- the change set strategies,
- the JVM arguments (e.g. maximum heap memory).

The default configuration is stored in the `config/config.json` file. Please use this as a basis for your configuration.

```
{
  "MinSize": 1,
  "MaxSize": 2,
  "Queries": ["PosLength", "RouteSensor", "SwitchSensor", "SwitchSet", "SemaphoreNeighbor"],
  "Tools": [<your tool>],
  "ChangeSets": ["fixed", "proportional"],
  "Runs": 1,
  "IterationCount": 5,
  "JVM": {"vmargs": "-Xmx4G"}
}
```

B.5 Interpreting the Output

Measurements are automatically recorded by our benchmark framework and stored in TSV (Tab-Separated Values) format. This can be used to automatically create diagrams with the provided R[2] script and provide comparable plots. For publishing performance results, please stick to the format generated by the framework.

Table 2 shows an example output. The ChangeSet defines the change set size (*fixed* or *proportional*, see Figure 2). The Train Benchmark is executed 5 times, the index of the current run is stored in the RunIndex attribute. The Query is executed by the Tool on the model with the given Size. The validation errors are repaired in multiple iterations, the index of the current iteration is shown in the Iteration attribute. Multiple values (MetricValue) of different metrics (MetricName) are measured during the benchmark. The execution time (*time*) and memory consumption (*memory*) for the *read*, *check*, *repair* and *recheck* phases are collected. The name the current phase is defined by the PhaseName. Additionally, the result set size (*rss*) is stored for the *check* phase and the iterations in the *recheck* phase.

C Evaluation Criteria

The solutions are checked and evaluated for functional, usability and performance aspects.

[2] `https://www.r-project.org/`

C.1 Correctness and Completeness of Model Queries and Transformations

The goal of the correctness check is to determine if the different model query and transformation tasks are correctly and fully implemented in the submitted solutions. We provide the number of invalid model elements in several models detected by our reference implementation for each query and iteration step. If the result sizes are consistently equal, the solution is considered to be correct.

The expected results are available at `https://github.com/FTSRG/trainbenchmark-ttc/tree/master/expected-results`.

Each task is scored independently $0-3$ points by the following rules:

- **0 points:** The task is not solved.
- **$1-2$ points:** The task is partially solved, the solution provides the subset or the superset of the expected results.
- **3 points:** The task is completely and correctly solved.
- **-1 point:** Only the query is implemented, but the transformation is not.

Correctness and completeness: 5 tasks $\times$ 3 points = 15 points

C.2 Conciseness

The validation rules are frequently changed and extended, therefore it is important to be able to define queries and transformations in a concise manner. These properties are scored based on the following rules:

- **0 points:** The task is not solved.
- **1 point:** The task is solved, but the solution is not significantly more concise than it would be in a general-purpose imperative language (e.g. Java), or the task is partially solved and the result set needs additional processing.
- **2 points:** The task is solved, the query and the transformation is defined in a declarative, visual or other query language, but the specification is hard to formulate.
- **3 points:** The solution is compact, the query and the transformation are defined in a concise manner.
- **-1 point:** Either the query or the transformation is implemented.

Conciseness: 5 tasks $\times$ 3 points = 15 points

C.3 Readability

The readability and descriptive power of each query and transformation is scored with respect to a model validation use case. The score represents how well model queries are used as model constraints, and how well repair operations can be expressed by model transformations. The score is given based on the following rules:

- **0 points:** The task is not solved.
- **1 point:** The task is solved, but the solution is not significantly more readable than it would be in a general-purpose imperative language (e.g. Java), or the task is just partially solved. For example, a typical EMF validator should get 1 point.

- **2 points:** The task is solved, the query and the transformation follows the description of the constraint and repair rule, but it is difficult to comprehend the meaning of the solution. For example, a foreign key constraint checked by a query formulated in SQL should get 2 points.
- **3 points:** The solution could be presented in the documentation of the modeling domain, and it is easier to comprehend than a textual description in natural language. For example, a solution similar to the graphical notation used in this paper should get 3 points.
- **−1 point:** If the language is only able to express either the constraint (e.g. OCL) or the repair operation.

Readability: 5 tasks $\times$ 3 points = 15 points

C.4 Performance on Large Models

The goal of the performance measurements is to check the applicability of the submitted solutions on large industrial models. During the performance tests the execution times will be measured for different scenarios and increasing model sizes.

Please restrict your benchmarks to those input models that can be processed within 5 minutes or less. Runs that take longer than 5 minutes will not be considered in the evaluation. Please provide a solution that can run on an x64-based Linux system with 4+ GB of memory, and that can be started on the command-line. This will be important to reproduce your results on a remote testing system. Please document the setup of your solution and the requirements to the system environment.

We defined two validation scenarios, based on the phases defined in Section 2.3:

batch The model is loaded (read) and validated (check).

repeated The model is loaded (read) and validated (check), then the model is edited (repair) and revalidated (recheck) 10 times.

The performance of the solutions are compared in 20 *tournaments*:

- The tournaments are calculated for the 5 tasks. If a solution skips a task, it is not considered in the tournament.
- Each solution is measured for both *batch* and *repeated* validation.
- Each solution is measured for both *fixed* and *proportional* change sets.

A solution gets from 0 to 1 points for a tournament which is launched for increasing model sizes. The score is based on the maximum size that the solution is able to handle, and its execution time relative to the fastest solution. Each measurement is executed 5 times and the median value is taken.

- The model size is increased as long as there is a solution that is able to solve it in the given time limit. This results in *rounds* $k = 1,2,3,\ldots,n$ for sizes 2^{k-1} $(1,2,4,\ldots,2^{n-1})$.
- For each tournament, a solution earns a score between 0 and 1, determined by

$$\frac{\sum_{k=1}^{n} score(k)}{\sum_{k=1}^{n} k},$$

where

$$score(k) = \begin{cases} score_{size}(k) \times score_{time}(k), & \text{if the solution runs correctly and within the given time limit,} \\ 0, & \text{if the solution fails to run correctly or exceeds the given time limit,} \end{cases}$$

and $\sum_{k=1}^{n} k = n \cdot (n+1)/2$ is used for normalizing the result.

- For each round k from 1 to n, if a solution is able to complete the validation, it is rewarded k points:
  - round 1 (size 1): the winner earns 1 point,
  - round 2 (size 2): the winner earns 2 points,
  - round 3 (size 4): the winner earns 3 points,
  - ...
  - round n (size 2^{n-1}): the winner earns n points.

  The formula is specified as:

  $$score_{size}(k) = k$$

- The fastest solution in each round earns 1 point, the other solutions earn partial points, based on the proportion of the current solution's execution time to the fastest execution time. The logarithm of this ratio for base 2 defines the score. For example:
  - if a solution takes 2× as long, it earns ½ points,
  - if a solution takes 4× as long, it earns ⅓ points,
  - if a solution takes 8× as long, it earns ¼ points,
  - and so on.

  The formula is specified as:

  $$score_{time}(k) = \frac{1}{1 + \log_2\left(\frac{\text{the solution's execution time in round } k}{\text{the fastest execution time in the round } k}\right)}$$

In conclusion, a solution earns up to 20 points for **performance**:

5 tasks × 2 validation scenarios × 2 change set sizes × up to 1 points = 20 points

C.5 Overall Evaluation

The scores of each aspect of the submitted solution are summarized to derive the final score (max. 65 points) used for ranking the submitted solutions.

ChangeSet	RunIndex	Tool	Size	Query	PhaseName	Iteration	MetricName	MetricValue
fixed	1	EMFIncQuery	1	PosLength	check	0	rss	43
fixed	1	EMFIncQuery	1	PosLength	recheck	1	rss	33
fixed	1	EMFIncQuery	1	PosLength	recheck	2	rss	23
fixed	1	EMFIncQuery	1	PosLength	recheck	3	rss	13
fixed	1	EMFIncQuery	1	PosLength	recheck	4	rss	3
fixed	1	EMFIncQuery	1	PosLength	recheck	5	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	6	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	7	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	8	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	9	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	10	rss	0
fixed	1	EMFIncQuery	1	PosLength	read	0	time	754739233
fixed	1	EMFIncQuery	1	PosLength	read	0	memory	6711048
fixed	1	EMFIncQuery	1	PosLength	check	0	time	51752
fixed	1	EMFIncQuery	1	PosLength	check	0	memory	6582280
fixed	1	EMFIncQuery	1	PosLength	recheck	1	time	5116
fixed	1	EMFIncQuery	1	PosLength	recheck	1	memory	2848944
fixed	1	EMFIncQuery	1	PosLength	recheck	2	time	4304
fixed	1	EMFIncQuery	1	PosLength	recheck	2	memory	2823352
fixed	1	EMFIncQuery	1	PosLength	recheck	3	time	8533
fixed	1	EMFIncQuery	1	PosLength	recheck	3	memory	2798328
fixed	1	EMFIncQuery	1	PosLength	recheck	4	time	4362
fixed	1	EMFIncQuery	1	PosLength	recheck	4	memory	2781144
fixed	1	EMFIncQuery	1	PosLength	recheck	5	time	4086
fixed	1	EMFIncQuery	1	PosLength	recheck	5	memory	2780248
fixed	1	EMFIncQuery	1	PosLength	recheck	6	time	4723
fixed	1	EMFIncQuery	1	PosLength	recheck	6	memory	2780344
fixed	1	EMFIncQuery	1	PosLength	recheck	7	time	8350
fixed	1	EMFIncQuery	1	PosLength	recheck	7	memory	2780440
fixed	1	EMFIncQuery	1	PosLength	recheck	8	time	12007
fixed	1	EMFIncQuery	1	PosLength	recheck	8	memory	2780536
fixed	1	EMFIncQuery	1	PosLength	recheck	9	time	4107
fixed	1	EMFIncQuery	1	PosLength	recheck	9	memory	2780632
fixed	1	EMFIncQuery	1	PosLength	recheck	10	time	21459
fixed	1	EMFIncQuery	1	PosLength	recheck	10	memory	2780776
fixed	1	EMFIncQuery	1	PosLength	repair	1	time	2861134
fixed	1	EMFIncQuery	1	PosLength	repair	1	memory	2855192
fixed	1	EMFIncQuery	1	PosLength	repair	2	time	3558045
fixed	1	EMFIncQuery	1	PosLength	repair	2	memory	2824640
fixed	1	EMFIncQuery	1	PosLength	repair	3	time	1090021
fixed	1	EMFIncQuery	1	PosLength	repair	3	memory	2800656
fixed	1	EMFIncQuery	1	PosLength	repair	4	time	1062007
fixed	1	EMFIncQuery	1	PosLength	repair	4	memory	2781272
fixed	1	EMFIncQuery	1	PosLength	repair	5	time	1235721
fixed	1	EMFIncQuery	1	PosLength	repair	5	memory	2780336
fixed	1	EMFIncQuery	1	PosLength	repair	6	time	8123
fixed	1	EMFIncQuery	1	PosLength	repair	6	memory	2780360
fixed	1	EMFIncQuery	1	PosLength	repair	7	time	3636
fixed	1	EMFIncQuery	1	PosLength	repair	7	memory	2780456
fixed	1	EMFIncQuery	1	PosLength	repair	8	time	14451
fixed	1	EMFIncQuery	1	PosLength	repair	8	memory	2780552
fixed	1	EMFIncQuery	1	PosLength	repair	9	time	2880
fixed	1	EMFIncQuery	1	PosLength	repair	9	memory	2780648
fixed	1	EMFIncQuery	1	PosLength	repair	10	time	3767
fixed	1	EMFIncQuery	1	PosLength	repair	10	memory	2780744

Table 2: Example output of the benchmark measurements.

Figure 4: Well-formed railway instance model.