Solving the TTC'15 Train Benchmark Case Study with SIGMA

Filip Křikava
Faculty of Information Technology
Czech Technical University, Czech Republic
filip.krikava@fit.cvut.cz

This paper describes a solution for the *Transformation Tool Contest 2015* (TTC'15) Train Benchmark case study using SIGMA, a family of Scala internal *Domain-Specific Languages* (DSLs) that provides an expressive and efficient API for model consistency checking and model transformations.

1 Introduction

The purpose of the TTC'15 Train Benchmark case study [3] is to systematically assess the scalability of consistency checking and repair of large scale models. It presents a scenario from the railway domain for which the solution requires to implement 5 constraints and repair operations of increasing complexity. An associated framework is then used to evaluate the correctness and performance of the solutions over large model instances.

In this paper we present our solution using SIGMA [1], a family of Scala[1] internal DSLs for model manipulation tasks such as model validation, model to model (M2M), and model to text (M2T) transformations. Scala is a statically typed production-ready *General-Purpose Language* (GPL) that supports both object-oriented and functional styles of programming. It uses type inference to combine static type safety with a *"look and feel"* close to dynamically typed languages. Furthermore, it is supported by the major integrated development environments bringing EMF modeling to other IDEs than traditionally Eclipse (*e.g.* IntelliJ IDEA was used for this solution).

SIGMA DSLs are embedded in Scala as a library allowing one to manipulate models using high-level constructs similar to the ones found in the external model manipulation DSLs. The intent is to provide an approach that developers can use to implement many of the practical model manipulations within a familiar environment, with a reduced learning overhead as well as improved usability and performance.

The solution is based on the *Eclipse Modeling Framework* (EMF) [2], which is a popular metamodeling framework widely used in both academia and industry, and which is directly supported by SIGMA. The complete source code is available on Github[2] in the fork of the original case study repository.

[1] http://scala-lang.org

[2] https://github.com/fikovnik/trainbenchmark-ttc in the `hu.bme.mit.trainbenchmark.ttc.benchmark.sigma` module

Submitted to:
TTC 2015

2 Solution Description

The solution for this transformation case study consist of a set of queries that check for violations of a number of model constrains and repair transformations that in turn fixes them. SIGMA provides a dedicated model consistency checking DSL with the ability to provide quick fixes repairing invariant validations. However, given the benchmark framework used in the case study, we decided to provide a more dedicated support for the given query/repair tasks in a form of an internal DSL. The reason is that (1) it allows for an easy comparison between the reference implementations in Java and EMF-IncQuery and (2) it shows the expressiveness of the language allowing one in few lines of code to bridge the gap between the problem-level abstractions (query, repair transformation) and the implementation-level concepts (*e.g.*, classes, higher-order functions). We therefore only rely on the SIGMA operations for model navigation (*i.e.* projecting information from models) and modification (*i.e.* changing model properties or elements). Essentially, these operations bridge the model classes (Ecore classes in this case) to be compatible with Scala allowing for example one to use the powerful Scala collection library.

On the top of SIGMA, we have created a small internal DSL that allows us to solve the given benchmark cases in an expressive and compact way. Following the case study description, the top-level domain concept is a *constraint*. A constraint is composed of a model *query* that finds all model instances violating a certain model restriction and a *repair* transformation correcting the failed instances. Concretely, a query is a function that given a model element—*i.e.* a context of the constraint in the classical model consistency checking—returns a set of *matches*. A match can either be a single instance or a tuple of instances of model elements that are related to the violations.

The following description of the solution is split in two parts: (1) the core part that describes the queries and repair transformations, (2) the integration part gives an overview how it has been integrated in the case study source code.

2.1 Queries and Repair Transformations DSL

A typical way of creating an internal DSL in Scala is by designing a library that allows one to write fragments of code with domain-specific syntax. These fragments are woven within Scala's own syntax so that it appears different.

One way to represent the above concepts is using a Scala case class:

```
1 case class Constraint[A <: EObject, B <: AnyRef](
2   query: (A) => Iterable[B],
3   repair: (B) => Unit
4 )
```

This defines a case class with a field for both query and repair. A case class in Scala is like a regular class with some additional properties out which, in our case, the important one is that it can be instantiated without the `new` keyword and thus limiting the language noise. The two type parameters `A, B` specify the model context for the query and the types of matches the query produces. The input type is further constrained to be a subtype of an `EObject`. The query and repair are defined as functions `A` → `Iterable[B]` and `B` → `Unit` where `Unit` is like `void` in Java.

In some cases the match returned by the query is of the same type as the query context. The query can be therefore simplified to a boolean expression selecting instances on which it evaluates to true. For these types of queries we provide a dedicated construct called `BooleanConstraint`:

```
case class BooleanConstraint[A <: EObject : ClassTag](
  query: (A) => Boolean,
  repair: (A) => Unit
)
```

For example, the first query, *PosLength*, which finds all the segments with negative length can be written as:

```
BooleanConstraint[Segment](
  query = segment => segment.length < 0,
  repair = segment => segment.length += -segment.length + 1
)
```

We do not have to specify the types of the parameter nor the result as they will be inferred by the Scala compiler.

Another example using more complex expression is the *SwitchSet* constraint:

```
Constraint[SwitchPosition, (Semaphore, Route, SwitchPosition, Switch)](
  query = swP => {
    for {
      semaphore <- Option(swP.route.entry) if semaphore.signal == Signal.GO
      sw = swP.switch if sw.currentPosition != swP.position
    } yield (semaphore, swP.route, swP, sw)
  },

  repair = {
    case (_, _, swP, sw) => sw.currentPosition = swP.position
  }
)
```

This is a more complex constraint that matches a tuple of model elements. It is using a *for comprehension*, a lightweight notation for expressing sequence comprehensions[3]. Scala for comprehensions have the form `for (enumerators) yield e`, where enumerators refers to a list of enumerators. An *enumerator* is either a generator which introduces new variables, or it is a filter. A comprehension evaluates the body *e* for each binding generated by the enumerators and returns a sequence of these values.

In this concrete example, the generator is the optional value of the `Route.entry` reference. It either generates a single value in the case the actual instance contains one or it does not produce anything. There is a small inconsistency in the model, the `Route.entry` should have the cardinality set to `0..1` instead of `1`, and that is why we need to explicitly convert the reference to an `Option`.

Finally, the repair function is defined using a pattern matching construct allowing us to concisely assign variables from the matching tuple.

[3] http://docs.scala-lang.org/tutorials/tour/sequence-comprehensions.html

2.2 Operationalization and Integration

The integration consists in making our solution work within the provided benchmark framework. First, next to constraint syntax, we also need to define its semantics. For that we define a validator which operationalizes the DSL executing the checks and consequent repairs of the incorrect model instances. It is defined as an abstract class with two methods that correspond to the two operations:

```
abstract class Validator[A <: EObject, B <: AnyRef] {
  def check(container: EObject): Iterator[B]
  def repair(matches: Iterator[B]): Unit
}
```

The implementation is straight forward. For all elements contained in a `container`, we first collect all instances of the required context type and then query them using the query function provided by the given constraint. The repair simply executes the constraint repair function on the matching element.

```
case class ConstraintValidator[A <: EObject, B <: AnyRef](constraint: Constraint[A, B])
  extends Validator[A, B] {

  override def check(container: EObject) =
    container.eAllContents collect { case x: A => x } flatMap constraint.query

  override def repair(matches: Iterator[B]) =
    matches foreach constraint.repair
}
```

Finally, we instantiate all the constraints, plugs them into the validator and connects the result to the provided framework. The integration schema is shown in Appendix B. We also create a `SigmaBenchmarkComparator` that is used to compare the matches as required by the case study. It is a general comparator that either compares single instances (results from boolean constraints violations) or tuples (regular constraints violations).

3 Evaluation

In this section we provide an evaluation of our solution following the categories specified in the case study.

Correctness and Completeness of Model Queries and Transformations. We developed a solution for all of the tasks required by the case study and the solution passes the provided tests.

Conciseness. The solution itself consists of 52 lines of Scala code the internal DSL developed for this case study. The DSL itself has been implemented using 20 lines of Scala code using SIGMA. The integration part consists of three files with the total of 65 lines. All measures are source lines only excluding comments and new lines. Given these measures, we believe that the code is rather concise.

Readability. Next to being concise, the solution is also quite expressive. This means that the given problem (queries and repair transformations) naturally maps into the implementation. The higher-level abstraction provided by both SIGMA and the internal DSLs helps to facilitate it making

a significant improvement over the Java reference implementation. The code is also type-safe as Scala is statically typed language. A notable consequence is that it is very easy to use the DSL with an IDE like Eclipse or IntelliJ that provides a robust code completing functionalities, outline views and other features increasing one's productivity.

In summary, while readability is a subjective matter and largely depends on the background and experience of users, we believe that SIGMA scores well. Thanks to the syntax of Scala which is close to one of Java/C++ and hence shall be familiar to many developers. The expressiveness of the first-order logic collection operation should be familiar to anyone knowing OCL or any other function language.

Performance on Large Models. The tests have been performed on an 2.3 GHz Intel Core i7 machine with 16 GB of RAM being dedicated to the JVM process. We ran our solution together with the reference implementation in Java. We used the model instances from size 1 to 8192 and set 8GB memory to be dedicated to the JVM. The performance is similar to the Java reference implementation which has been expected due to the fact that Scala compiles directly to Java bytecode and we use the same underlying libraries for accessing EMF models. This shows that we can leverage from concise and expressive queries without sacrificing performance. The overhead of using SIGMA is mostly on the compile time where the implicit conversions are inlined by the Scala compiler.

It is important to note that we do not developed any extra functionality for these benchmarks—*i.e.* no caching or incremental validations. On the other hand, functional approach we have selected makes it perfect for further parallelization.

4 Conclusion

This paper presents a solution for the Train Benchmark case study of the 2015 Transformation Tool Contest. It demonstrates some of the features of the SIGMA internal DSLs for model manipulation as well as the extensibility, expressiveness and scalability of the Scala host language. The solution is realized as a tiny internal DSL in Scala that mixes in SIGMA common infrastructure for EMF model querying and manipulation. There is a significant improvement in the readability and conciseness of the solution, yet the performance is similar to the reference Java version.

Acknowledgments. This work is partially supported by the Datalyse project[4].

References

[1] Filip Krikava, Philippe Collet & Robert France (2014): *SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations*. In: *Proceedings of 17th International Conference on Model-Driven Engineering Languages and Systems*, MODELS, Valencia.

[2] Dave Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2008): *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional.

[3] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation**. In: *Transformation Tool Contest 2015*.

[4] http://www.datalyse.fr

A Constraints

In the following we describe the individual constraints that were part of the case study (case study tasks) except the *PosLength* and *SwitchSet* which have already been shown above (*cf.* Section 2).

— *SwitchSensor*

```
BooleanConstraint[Switch](
  query = switch => switch.sensor.isEmpty,
  repair = switch => switch.sensor = Sensor()
)
```

The `isEmpty` is a method that is defined on an `Option` type (coming from the standard Scala library) representing a type which may or may not have a value. Since in the model, the *sensor* reference of the `Switch` class is defined as optional (with cardinality `0..1`), in SIGMA we represent the reference using the `Option` class. Not only makes this the cardinality expressed in the type definition, but it also prevents some of the `NullPointerException`s caused by traversing unset references. Technically, this is realized by implicit conversions (*cf.* Krikava *et al.* [1]).

— *RouteSensor*

```
Constraint[Route, (Route, Sensor, SwitchPosition, Switch)](
  query = route => {
    for {
      swP <- route.follows
      sw = swP.switch
      sensor <- sw.sensor if !(route.definedBy contains sensor)
    } yield (route, sensor, swP, sw)
  },

  repair = {
    case (route, sensor, _, _) => route.definedBy += sensor
  }
)
```

The implementation is similar to the the previous case. It is also based on a for comprehension and closely follows the description of the query.

— *SemaphoreNeighbor*

```
Constraint[Route, (Semaphore, Route, Route, Sensor, Sensor, TrackElement, TrackElement)](
  query = route1 => {
    for {
      sensor1 <- route1.definedBy if route1.exit != null
      te1 <- sensor1.elements
      te2 <- te1.connectsTo
      sensor2 <- te2.sensor
      route2 <- sensor2.sContainer[Route] if route1 != route2
      semaphore = route1.exit if semaphore != route2.entry
    } yield (semaphore, route1, route2, sensor1, sensor2, te1, te2)
  },

  repair = {
    case (semaphore, _, route2, _, _, _, _) => route2.entry = semaphore
  }
)
```

Again based on the for comprehension. Additionally, we provide a shortcut using the `route1.exit != null` so immediately skip the route instances that do not have an exit semaphore set.

B Integration with the Train Benchmark Framework

Figure 1 shows the various layers of integration of the solution into the train benchmark framework.

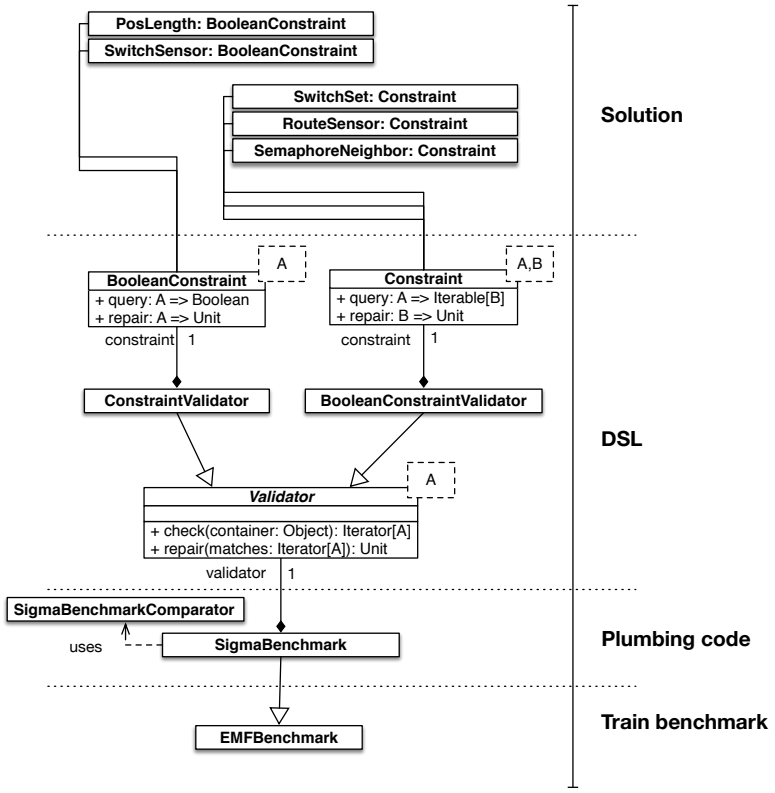

Figure 1: Integration schema

C Performance Comparison

The performance comparison charts (*cf.* Figures 2, 3, 4 and 5) have been generated by the case study benchmark. They present a performance comparison between SIGMA and the reference implementation in Java on the model instances from size 1 to 8192 using 8GB memory dedicated to the JVM. The corresponding results are shown in the figures 2 and 3. We compare them to the Java solution which is shown in the figures 4 and 5.

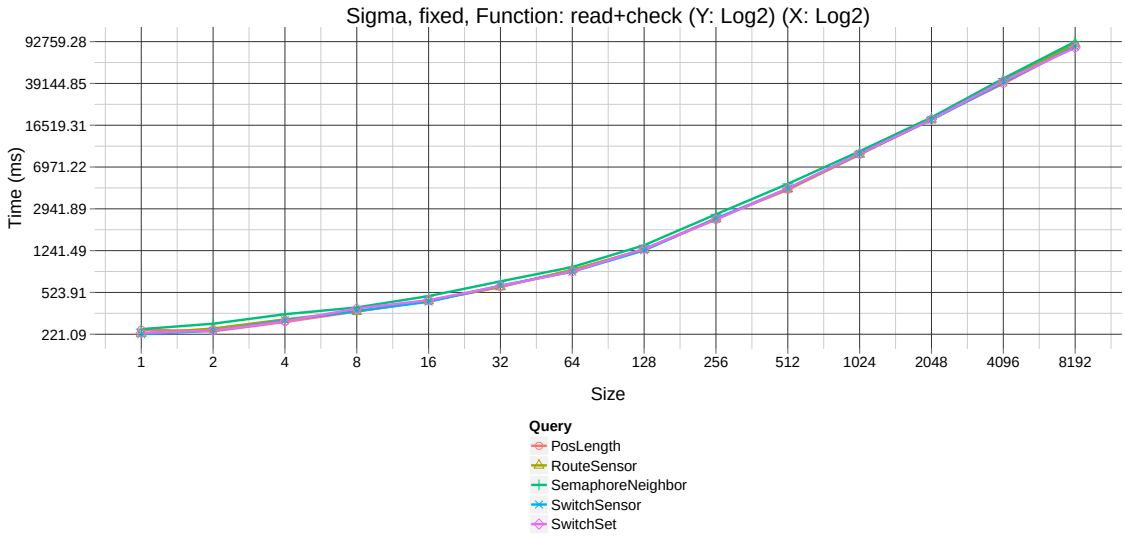

Figure 2: SIGMA fixed validation batch

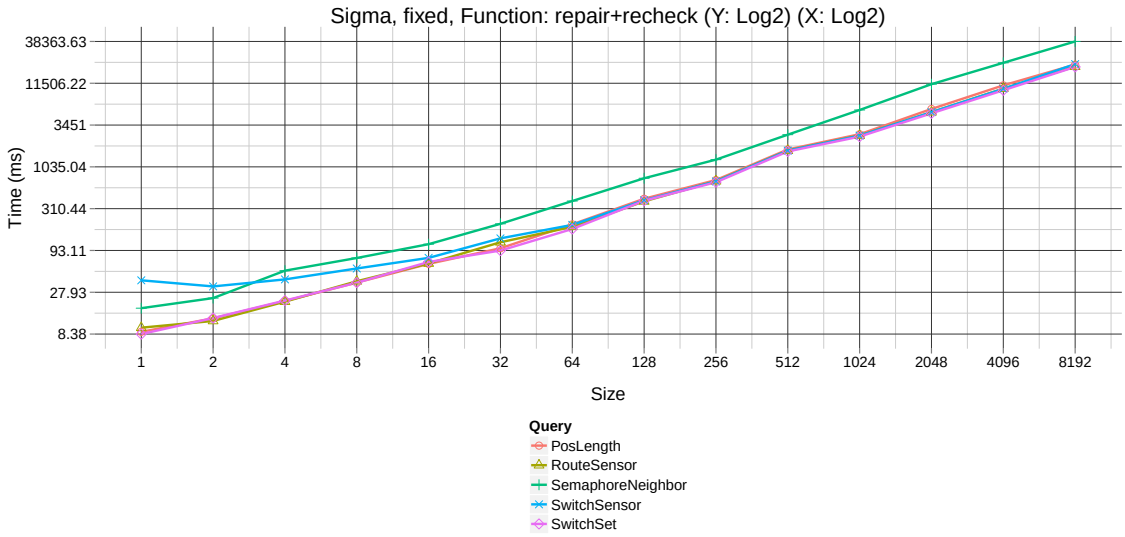

Figure 3: SIGMA fixed revalidation

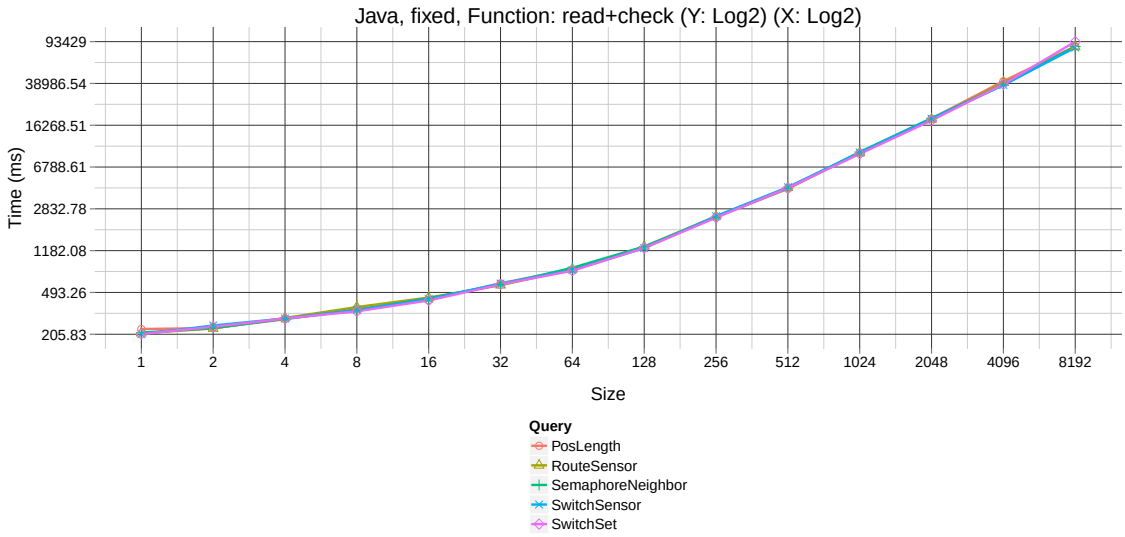

Figure 4: Java fixed validation batch

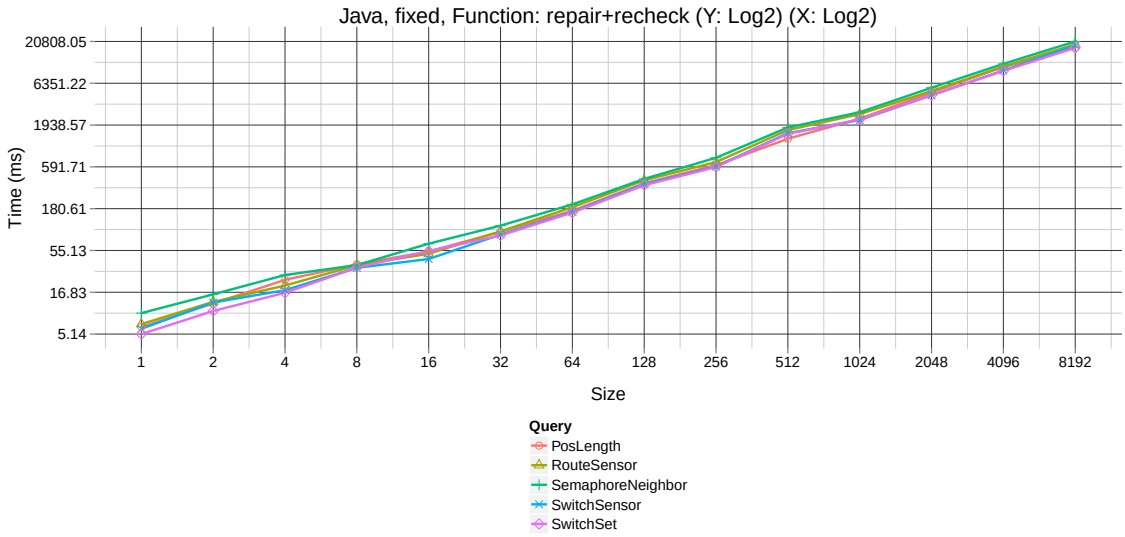

Figure 5: Java fixed revalidation