An NMF solution to the Train Benchmark Case at the TTC 2015

Georg Hinkel
Forschungszentrum Informatik (FZI)
Haid-und-Neu-Straße 10-14, Karlsruhe, Germany
hinkel@fzi.de

Lucia Happe
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, Karlsruhe, Germany
lucia.kapova@kit.edu

Model validation in model-driven development gains in importance as the systems grow in size and complexity. In this situation an efficiency of validation execution and an immediate feedback whether a recent manual edit operation broke a validation rule is desirable. To increase efficiency, incremental model validation tries to minimize the proportions of the model that have to be rechecked by reusing previous validation results. As a benchmark for efficiency of validation tools, the Train Benchmark Case at the Transformation Tool Contest 2015 was created. In this paper, we present a solution using NMF Expressions, a tool for incremental evaluation of arbitrary expressions on the .NET platform.

1 Introduction

This paper proposes a solution for the Train Benchmark Case[1] at the Transformation Tool Contest (TTC) 2015. Our solution is publicly available on CodePlex[1] and SHARE[2] and built upon the .NET Modeling Framework[3] (NMF) and especially on *NMF Expressions*[4]. NMF is a tool suite on the .NET platform to support model-driven engineering. Its metamodel NMeta is largely compatible with Ecore so that Ecore metamodels can be transformed to NMeta with a compliant XMI format, i.e. models according to an Ecore metamodel can be deserialized using the transformed NMeta metamodel.

NMF Expressions is designed for implicitly incremental evaluation of arbitrary (lambda calculus) expressions. This is done based on a theoretical foundation of representing incremental computation systems as a monad. The implicit approach means that developers specify the expressions in a batch mode whereas the incrementality is added through the monad. As a consequence, the syntax is very understandable as also remarked by the peer reviewers.

So far, few companies have adopted MDE as their main development paradigm with one of the major reasons being the lack of tool support [2], [3]. Developers are used to an excellent tool support for languages like Java or C# which many MDE tools cannot bear to meet. Furthermore, studies as e.g. by Meyerovich [4] suggest that developers only change their primary programming language when a project requires them to or they can reuse a large proportion of code. We see no reason why this should not extend to model validation tools and thus we are seeking for the ways to let developers specify these expressions in their primary languages.

Our goal is to hide the incrementality concerns from the developer, who only has to specify the validation expression, and automate the incrementalization of the validation expression, aiming for a declaritive usage of the C# language.

[1] http://ttc2015trainbenchmarknmf.codeplex.com
[2] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64-TTC15_NMF.vdi
[3] http://nmf.codeplex.com
[4] http://nmfexpressions.codeplex.com

In this paper, we evaluate the efficiency of incremental validation with *NMF Expressions*. The rest of this paper is structured as follows: Section 2 gives a very short introduction to *NMF Expressions* and Section 3 explains our solution. Finally Section 5 summarizes the paper.

2 NMF Expressions

The goal of *NMF Expressions* is to give developers an automated tool at hand providing them with advantages of incremental evaluation for arbitrary expressions. Unlike many other approaches, our approach works implicitly, so developers only have to specify their expressions and *NMF Expressions* takes care of how to turn this into an algorithm that will evaluate the expression in an incremental fashion. On the other hand, the traditional batch mode specification is still available so that *NMF Expressions* yields a choice whether to run a given expression incrementally or in batch mode.

In the incremental mode, the approach creates a dynamic dependency graph from a given expression and observes changes. These changes originate from elementary update notifications and are propagated through the dependency graph. Operating on the .NET platform, *NMF Expressions* uses the industry standard `INotifyPropertyChanged` and `INotifyCollectionChanged` interfaces to record elementary changes. These are also required by a lot of other tools including the modern UI libraries on the .NET platform. As a consequence of the theoretical foundation using monads, the dependency graph contains specialized nodes for optimized incrementalization of queries.

While *NMF Expressions* works with arbitrary model representations implementing the interfaces for elementary change propagation, we use the model representation of *NMF*. That is, we transformed the given Ecore metamodel of the railway domain into an NMeta metamodel and generated model representation code. *NMF* thus offers us a deserialization mechanism to load the resulting models as objects into memory.

3 Solution with NMF Expressions

The intended usage of *NMF Expressions* in incremental mode is that users would modify the model in some editor through a sequence of change operations, each of which providing elementary change notifications. Then, *NMF Expressions* would use the elementary change notifications and combine them to provide immediate feedback whether the most recent model manipulation has caused some validation rule to fail for some model elements. Currently, *NMF Expressions* always minimizes the model elements that it has to look at, even at the cost of high memory usage. However, in the Train Benchmark, the only model manipulations we can see are the repair operations, so for us the benchmark does not really reflect the situation for which we have designed *NMF Expressions*.

In incremental mode, *NMF Expressions* creates a cache for the selected expressions and maintains this cache. This maintenance happens automatically as *NMF Expressions* adds computational effort to the (in-memory) online model manipulation. In this case solution, we created expressions for the validation patterns so *NMF Expressions* caches the invalid elements continuously. However, this means that the phases drawn from the case description get blurred. In particular, the check phases get meaningless as the updated results are always available and could be used for immediate feedback, while more computational effort is put to the model manipulation such as the modify operations.

Because *NMF Expressions* allows to use the same specification both in a classic batch manner as also incrementally, our solution can also be configured to run in batch mode without any changes to the

PosLength

```
Fix(pattern: rc.Descendants().OfType<Segment>()
               .Where(seg => seg.Length <= 0),
   action: segment => segment.Length = -segment.Length + 1);
```

SwitchSensor

```
Fix(pattern: rc.Descendants().OfType<Switch>()
               .Where(sw => sw.Sensor == null),
   action: sw => sw.Sensor = new Sensor());
```

SwitchSet

```
var routes = rc.Routes.Concat(rc.Invalids.OfType<Route>());
Fix(pattern: from route in routes
             where route.Entry != null
               && route.Entry.Signal == Signal.GO
             from swP in route.Follows.OfType<SwitchPosition>()
             where swP.Switch.CurrentPosition != swP.Position
             select swP,
   action: swP => swP.Switch.CurrentPosition = swP.Position);
```

RouteSensor

```
Fix(pattern: from route in routes
             from swP in route.Follows.OfType<SwitchPosition>()
             where swP.Switch.Sensor != null &&
               !route.DefinedBy.Contains(swP.Switch.Sensor)
             select new { Route = route, Sensor = swP.Switch.Sensor },
    action: match => match.Route.DefinedBy.Add(match.Sensor),
```

SemaphoreNeighbor

```
Fix(pattern: from route1 in routes
             from route2 in routes
             where route2.Entry != route1.Exit
             from sensor1 in route1.DefinedBy
             from te1 in sensor1.Elements
             from te2 in te1.ConnectsTo
             where te2.Sensor == null
               || route2.DefinedBy.Contains(te2.Sensor)
             select new { Route = route2, Semaphore = route1.Exit },
    action: match => match.Route.Entry = match.Semaphore);
```

patterns. When executed in batch mode, *NMF Expressions* simply forwards the call to the LINQ to objects implementation. Besides a negligible runtime compilation effort, this utilizes the highly optimized platform LINQ implementation.

The patterns are **enumerable expressions** where developers can choose at runtime whether the pattern should be executed in batch mode or whether *NMF Expressions* should register for elementary change notifications to keep a cache of the result up to date. To specify patterns, we created a small method `Fix` that captures them.

```
public void Fix<T>(IEnumerableExpression<T> pattern, Action<T> action) {
  var patternInc = pattern.AsNotifiable();
  foreach (T element in patternInc) action(element);
  patternInc.CollectionChanged += (o,e) => {
    if (e.NewItems != null)
      foreach (T element in e.NewItems)
        action(element);
  }}
```

Listing 1: A simplified implementation of the `Fix` function

The easiest implementation for the `Fix` function repairing any validation error as soon as they occur

would be the one presented in Listing 1. In Line 2, we tell *NMF Expressions* that we want to obtain incremental updates for the given pattern. Line 3 repairs all occurences existing so far and Lines 4-8 handle new pattern matches. For the benchmark, we adopted the `Fix` function to account for the benchmark phases. In particular, the implemented version takes a third parameter to allow us to sort matches. Since these sort keys offer little insight, we omit them in the pattern presentation.

In the following we will present the solution to the tasks, following the structure of the case description, though with omitted sort keys.

Please note that the parameter names such as `pattern` or `action` are optional, we only included them for better understandability.

The solutions to *SwitchSet*, *RouteSensor* and *SemaphoreNeighbor* use the query syntax of C#. This syntax is translated to the method chaining syntax by mapping the query keywords like `from` or `where` to method calls of *NMF Expressions*. Such query expressions are commonality on the .NET platform and thus easy to write and understand by most developers.

Note that the order in which the statements occur does make a difference. In particular, e.g. lines 2 and 3 of the *SwitchSet* solution could logically be interchanged but cause a slightly different implementation. *NMF Expressions* currently does not optimize the query for performance.

In the solution for *SemaphoreNeighbor* we can observe that *NMF Expressions* is not able to inverse directed references. We argue that such inversion is always limited to a particular scope, which is unclear from the context. If the context was clear, the reference should have been navigable in both directions in the metamodel. As this is not the case, we have to cross join the two respective routes and filter them on the semaphores.

4 Evaluation

To evaluate our solution, we ran it in comparison to the reference implementations in Java and EMF-IncQuery [5]. The measurements were taken on a system with an Intel i5-4300U processor in a system equipped with 12GB RAM running on Windows 8.1 Pro, .NET 4.5 and Java 1.8 update 45. The results for recheck and repair are shown in Figure 1. The *SemaphoreNeighbor* ran out of memory for larger models. A discussion is omitted for space limitations.

In all four presented queries, both versions are up to multiple magnitudes faster than the plain Java solution. In medium-sized models, the incremental version also beats EMF-IncQuery, in the *SwitchSet* pattern it is even faster on the larger models.

5 Summary

In this paper, we presented an NMF solution to the Train Benchmark case at the TTC 2015.

The queries and repair transformations demonstrate why we have sticked to the C# language. We think that it is very hard to get a more concise textual solution for this case. At the same time, developers get the full tool support from e.g. Visual Studio and the query syntax that we use is used by thousands of developers already and widely understood.

The performance figures shows that the incremental version of our solution outperforms the batch mode execution of the same solution in all cases. At the same time, our solution yields a batch mode execution in cases where only a single analysis run is needed. This version outperforms the classic Java solution in several orders of magnitude. On the other hand, for large models our incremental solution

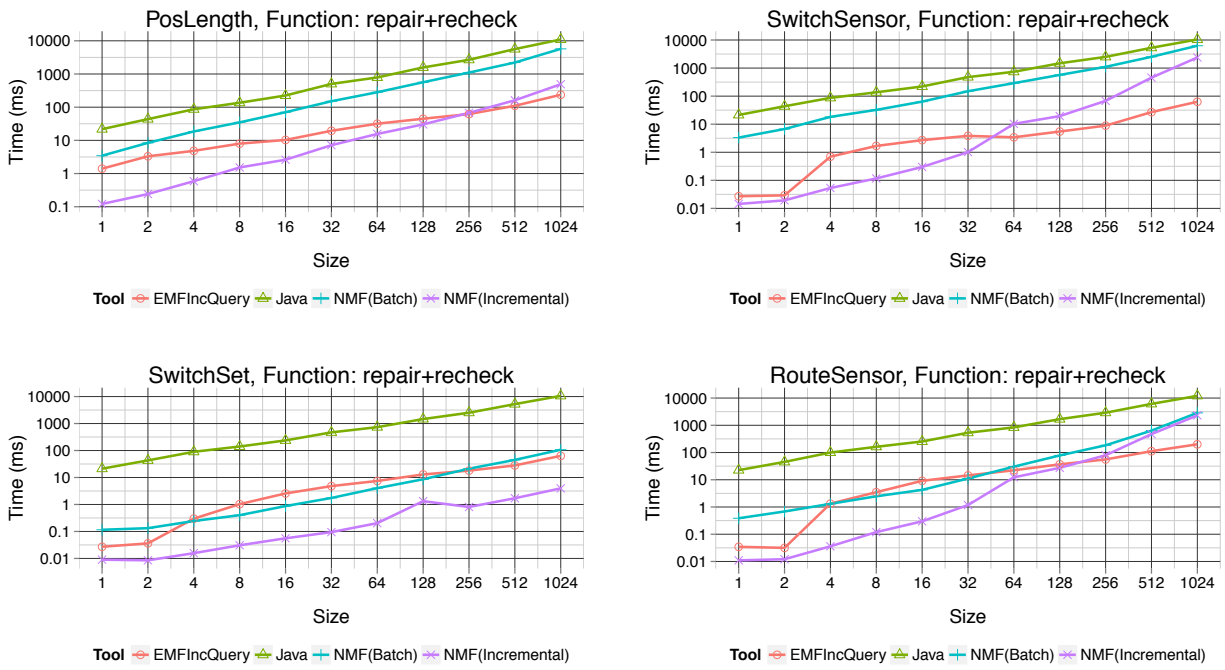

Figure 1: Performance Results for the NMF solution versions compared to the reference solutions in Java and EMF-IncQuery

cannot keep up with the EMF-IncQuery solution, except for one pattern where the NMF solution is slightly faster.

The biggest advantage of our solution is that it gives both a batch mode solution and an incremental solution our of the same pattern specifications. Thus, the same analysis code can be used in the case setting where incrementality is a clear advantage, or in a batch mode, e.g. when memory is a sparse resource or the analysis results are only required once.

References

[1] G. Szárnyas, O. Semeráth, I. Ráth, and D. Varró, "The TTC 2015 Train Benchmark Case for Incremental Model Validation," in *8th Transformation Tool Contest (TTC 2015)*, 2015.

[2] M. Staron, "Adopting model driven software development in industry–a case study at two companies," in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 57–72.

[3] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering*, vol. 18, no. 1, pp. 89–116, 2013.

[4] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ACM, 2013, pp. 1–18.

[5] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös, "Incremental evaluation of model queries over emf models," in *Model Driven Engineering Languages and Systems*, Springer, 2010, pp. 76–90.