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

Georg Hinkel[1] and Lucia Happe[2]

[1] Forschungszentrum Informatik (FZI)
Haid-und-Neu-Straße 10-14, Karlsruhe, Germany
hinkel@fzi.de
[2] Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, Karlsruhe, Germany
lucia.kapova@kit.edu

Abstract 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[3] at the Transformation Tool Contest (TTC) 2015. Our solution is publicly available on CodePlex[4] and SHARE[5] and built upon the .NET Modeling Framework[6] (NMF) and especially on *NMF Expressions*[7]. 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 incremental evaluation of arbitrary (lambda calculus) expressions. This is done based on a theoretical foundation of representing incremental expressions as monads. These monads represent the reguired

[3] https://github.com/FTSRG/trainbenchmark-ttc/raw/master/paper/trainbenchmark-ttc.pdf
[4] http://ttc2015trainbenchmarknmf.codeplex.com
[5] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64-TTC15_NMF.vdi
[6] http://nmf.codeplex.com
[7] http://nmfexpressions.codeplex.com

validation expressions and can be specified in C# conveniently through the query syntax. Our goal is to hide the incrementality concerns from the developer, who has to specify the validation expression only, and automate the incrementalisation of the validation expression. Therefore, we use the C# language in as declarative style as possible. 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.

2 NMF Expressions

The goal of *NMF Expressions* is to give developers a automated tool at hand providing them with advantages of incremental evaluation for arbitrary expressions. Unlike many other approaches, our approach works implicit, 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. The presented approach creates a dynamic dependency graph from a given expression and observes changes. These changes are recorded by elementary update notifications. As *NMF Expressions* operate on the .NET platform where these update notifications are sent via events of the `INotifyPropertyChanged` and `INotifyCollectionChanged` interfaces. These interfaces are an industry standard on the .NET platform and are also required by a lot of other tools including the modern UI libraries on the .NET platform that describe user interfaces through XAML (an XML dialect) and use these change notifications to keep the user interface updated. These elementary notifications are then assembled by *NMF Expressions* through these dependency graphs which hold the current value of a given expression and propagate updates through the graph.

3 Solution with NMF Expressions

The intended usage of *NMF Expressions* is that users would modify the model in some editor as expressions. Then, *NMF Expressions* would use the elementary update notifications and use them intelligently 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 if that causes a high memory usage. However, in the Train benchmark, the only model manipulations we can see are the repair operations, so the benchmark does not really reflect for us the situation for which we have designed *NMF Expressions*.

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, the program that drives our solution can be also be configured to run in batch mode without any changes to the patterns.

In the following we will present the solution to the tasks, following the structure of the case description, using *NMF Expressions*.

PosLength

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

SwitchSensor

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

SwitchSet

```
1 Fix(pattern: from route in rc.Routes
2             where route.Entry != null && route.Entry.Signal == Signal.GO
3             from swP in route.Follows.OfType<SwitchPosition>() where
4             swP.Switch.CurrentPosition != swP.Position select swP,
5    action: swP => swP.Switch.CurrentPosition = swP.Position);
```

RouteSensor

```
1 Fix(pattern: from route in rc.Routes
2             from swP in route.Follows.OfType<SwitchPosition>() where
3             swP.Switch.Sensor != null &&
4             !route.DefinedBy.Contains(swP.Switch.Sensor) select new { Route
5             = route, Sensor = swP.Switch.Sensor },
6    action: match => match.Route.DefinedBy.Add(match.Sensor),
```

SemaphoreNeighbor

```
1 Fix(pattern: from route1 in rc.Routes
2             from route2 in rc.Routes where route2.Entry != route1.Exit from
3             sensor1 in route1.DefinedBy from te1 in sensor1.Elements from
4             te2 in te1.ConnectsTo where te2.Sensor == null ||
5             route2.DefinedBy.Contains(te2.Sensor) select new { Route =
6             route2, Semaphore = route1.Exit },
7    action: match => match.Route.Entry = match.Semaphore);
```

The patterns are **enumerable expressions** where developers can choose at runtime whether the pattern should be executed in a batch manner or whether *NMF Expressions* should register for atomic element changes to keep the pattern computation up to date. Note as well that the parameter names `pattern` and `action` are optional, we only included them for better understandability.

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

```
6       foreach (T element in e.NewItems)
7        action(element);
8     }
9   }
```

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

The easiest implementation for the `Fix` function using the latter and 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. However, the implementation of `Fix` that we use in our solution is much more complicated taking into account the frame conditions of the benchmark. In particular, we need a third parameter which selects us a pattern sort key, so that we have the chance to sort the patterns.

The solution to *SwitchSet* is a bit more interesting since the pattern involved is a bit more complex. Here, the method chaining syntax would no longer be concise and understandable. Thus, we 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, lines 2 and 3 could logically be interchanged but cause a slightly different implementation. In particular, *NMF Expressions* currently does not optimize the query for performance.

4 Summary

In the following we discuss our observations about the evaluation criteria suggested by the case description, especially: conciseness, readability and performance.

Conciseness and readability The queries and repair transformations demonstrates 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 syntax that we use is used by hundred thousands of developers already.

Furthermore, the Line 4 of the solution to *RouteSensor* shows that our solution is also able to select multiple elements from a given pattern through the usage of anonymous types. 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.

The *NMF Expressions* can be specified in C# conveniently through the query syntax. Apart from the expressiveness of our approach, we regard this as a big advantage. 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 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 expressions and thus we are seeking for the ways to let developers specify these expressions in their primary languages.

Performance With our implementation of *NMF Expressions*, we have manifested speedups up to 4.8 for the incremental evaluation of some expressions compared to their batch execution running on .NET. These advantages come usually at the price of a higher memory consumption which we argue is affordable given the current memory prices. Meanwhile, the performance that can be gained from incremental execution is not constant but individual for all queries. For the given benchmark, the platform restriction that the solution has to run on Linux means we have to use Mono as the runtime environment, which we have not properly tested yet. We already noticed that the memory metric by taking the working set size does not work. Furthermore, the restriction that the matches should be sorted means that the model manipulation steps get inflated by the sorting, which is not necessary for the determinism of the results but introduces bias.

With NMF Expressions, we have come to a point where we can let developers choose at runtime whether they want their expressions to be evaluated incrementally or in classical batch mode, depending on which solution seems more appropriate. However, given the page limit, we have not included any performance figures.

References

[1] 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.

[2] 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.

[3] 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.