Karlsruhe Reports in Informatics 2016,9
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

NMF: A Modeling Framework for the .NET Platform

Georg Hinkel

2016

NMF: A Modeling Framework for the .NET Platform

Georg Hinkel

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

Abstract For its promises in terms of increased productivity, Model-driven engineering (MDE) is getting applied increasingly often in both industry and academia. However, most tools currently available are based on the Eclipse Modeling Framework (EMF) and hence based on the Java platform whereas tool support for other platforms is limited. This leads to a language and tool adoption problem for developers of other platforms such as .NET. As a result, few projects on the .NET platform adopt MDE. Furthermore, the limited tool availability introduces a technical barrier in the interoperability between EMF and .NET applications. In this paper, we present the .NET Modeling Framework (NMF), a tool set for model repositories, model-based incrementalization, model transformation, model synchronization and code generation. The framework makes intensive use of the C# language as host language for model transformation and synchronization languages, whereas the model repository serialization is compatible with EMF. This solves the language adoption problem for C# programmers and creates a bridge to the EMF platform.

1 Introduction

Model-driven engineering (MDE) is getting applied increasingly often both in industry and academia. Dedicated support to use models for analysis or transformation purposes reduces manual development efforts as repetitive infrastructure code can be reused. Most of the existing tools that support MDE are currently based on the Java platform. As a consequence, legacy software built on other platforms can hardly be reused.

According to many indices on the popularity of programming languages such as the TIOBE index[1] or the IEEE Spectrum[2], Java is the most popular programming language. Most students in technical majors learn programming through Java in their first term. Furthermore, it is platform independent. This makes Java a straight-forward implementation choice for Model-driven tools, which are still mostly developed within academia. However, Java is not used everywhere. In

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[2] http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

some domains, other languages are more popular. Furthermore, as Meyerovich suggests, most programmers do not easily change their primary programming languages [1]. When the adoption of MDE implies the adoption of the Java framework, this can block the adoption of MDE in domains where Java is not the primary language to work with. In addition, although MDE has existed for more than a decade now, tool support is still one of the major factors that hampers a wide adoption of MDE in industry [2], [3]. The often home-brew tools with a lack of professional support can hardly keep up with industry standard integrated development environments such as IntelliJ or Visual Studio.

In this paper, we present the .NET Modeling Framework (NMF), a framework of libraries, tools and languages to support model-driven engineering on the .NET platform. The framework is dedicated to process existing models through analysis, transformation and synchronization. NMF contains tools to generate model representations compliant with EMF, supports a model management repository system and allows developers to specify model analyses, model transformations and model synchronizations. To minimize both the language adoption problem and the tool support problem, NMF is entirely based on internal languages that use C# as a host language.

An introductory tutorial for NMF can be found on YouTube[3].

The remainder of this paper is structured as follows: Section 2 presents the meta-metamodel used in NMF and discusses serialization. Section 3 explains the support for model repositories and how they are used. Section 4 describes the support for implicit incremental model analyses that is built into NMF. Section 5 introduces the model transformation language NTL. Section 6 shows how the concepts are combined in a language for the synchronization of heterogeneous metamodels. Finally, Section 7 concludes the paper.

2 Meta-Metamodel

NMF contains its own meta-metamodel called NMeta. NMeta is similar to Ecore but contains dedicated support for type system features widely used on the .NET platform such as structures or events. Furthermore, it also supports an extension mechanism closely related to stereotypes as well as refinements. The semantics of NMeta is clearly defined through a mapping to category theory. Though there is a high semantic overlap with the Essential Meta Object Facility (EMOF) standard, there are also some features that do not have a counterpart in NMeta, in particular factories and generic types.

An excerpt of NMeta is depicted in Figure 1. Similar to Ecore, it describes classes by inheritance, attributes and references (and operations, omitted in Figure 1). References are allowed to be containments and to define opposite references. However, both attributes and references may also refine other attributes and references and an additional `InstanceOf` relation supports deep modelling, though we omit a detailed description in this paper due to space limitations.

[3] `https://youtu.be/NIMYuwTltVs`

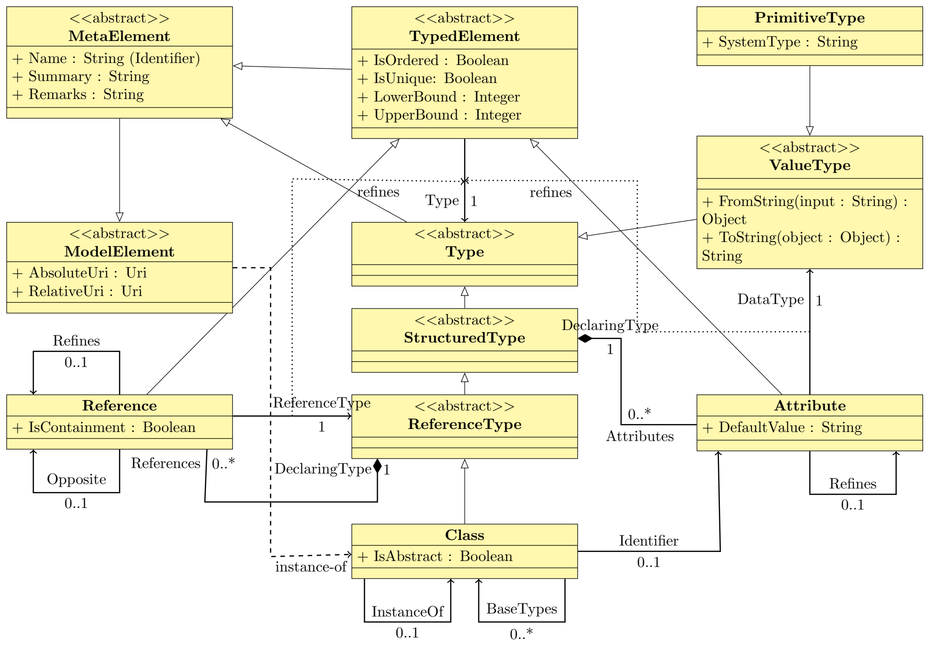


Figure 1. The meta-metamodel NMeta (simplified)

However, since Ecore is the meta-metamodel most often used in MDE, NMF contains a model transformation from Ecore to NMeta. This transformation is based on the extensible Model Transformation Language NTL (cf. Sec. 5 or [4]) and thus support for other types can be easily extended.

The resulting NMeta metamodel is compliant with the original Ecore metamodel if the latter only contains basic structures (packages, classes, attributes and references). Here, compliant means that serialized instances of the original Ecore metamodel can be deserialized with the NMeta metamodel (if no custom XMI handlers are used) and vive versa. In particular, the XMI serialization of the metamodels is equivalent and the NMF serializer uses the same addressing scheme for cross references as the EMF serializer uses for Ecore. This allows developers to easily use Ecore-based models in .NET applications and already allowed us to participate in several TTC cases [5]–[8].

Similar to Ecore, NMeta is bootstrapped and the classes `ModelElement` and `Model` are the only ones with a custom implementation, the implementation of all other classes originate directly from the code generator.

3 Model Repositories

In NMeta, all model elements have both an absolute and a relative URI that allow developers to easily reference model elements in a defined way. The addressing scheme is based on the containment hierarchy where the elements are identified by their identifier or by the collection index. The syntax is the same as used in the EMF serializer to push interoperability to EMF.

NMF is able to resolve URIs from different sources, including files, embedded resources and network streams. To resolve a model, NMF uses a singleton meta repository (which itself is a model repository) where all metamodels are loaded and linked to the implementation, if available. The registration of model representation code is done simply through an assembly annotation that links a namespace to an assembly embedded resource where the metamodel is formally described. Here, assemblies are the components of the .NET component model. When the meta-repository is loaded for the first time, it iterates through the loaded assemblies and finds all metamodels registered, so that a repository is able to load a model just in case the assembly containing the model representation classes is referenced.

4 Model-based Incrementalization

Again similar to EMF, NMF provides elementary change notifications, offered through the industry standard interfaces `INotifyPropertyChanged` and `INotifyCollectionChanged`. These interfaces are required by many modern user interface libraries, hence the model representation code can directly be used for these techniques.

However, NMF is also able to combine these elementary change notifications to determine when the result of analyses based on a model has changed. Furthermore, an incremental algorithm is inferred to recalculate the analysis upon a model change more efficiently by the implicit introduction and management of buffers to save intermediate results. This incrementalization works online, i.e. the model needs to be kept in memory and changes must be made on the model elements in memory.

The incrementalization has a sound theoretical foundation based on category theory and is implemented in NMF Expressions. NMF Expressions operates on lambda expressions, supported by many .NET languages such as C# and VB.NET in their regular syntax. Lambda expressions are based on the Turing-complete λ-calculus. Thus, all model analyses can be specified in this format. To realize the incrementalization, the abstract syntax tree is converted into a dynamic dependency graph on a high abstraction level. Changes of the model under analysis are then propagated through the dependency graph, ultimately updating the analysis result.

The high abstraction level in the dynamic dependency graph is achieved by a manual incrementalization of analysis operators yielding valid results as a consequence of the underlying formalization as a categorial functor. NMF Expressions includes a library of such manually incrementalized operators, including most of the Standard Query Operators (SQO)[4]. As a consequence, developers can specify query analyses conveniently through the query syntax offered e.g. by C# and

[4] `http://msdn.microsoft.com/en-us/library/bb394939.aspx`

VB.NET, although also implementations for other forms of analysis exist such as for example connectivity analysis.

5 Model Transformation

To support model transformation, NMF contains the NMF Transformations Language (NTL) [9], an internal model transformation language integrated in C#, reusing the tool support for C# [10]. This transformation language allows to specify extensible rule-based model transformations with explicit dependencies between the transformation rules. The underlying transformation engine is not restricted to NMeta models as input or output models such that also arbitrary CLR objects can be transformed where the CLR denotes the .NET virtual machine, similar to the JVM in Java.

Model transformations in NTL are essentially classes whose transformation rules are inferred by the public nested classes. These are encoded also as separate classes that inherit from a set of generic base classes and the generic type parameters specify the source and target model elements. These transformation rule classes may override a method to define their dependencies. Inside this method, transformation rules may define dependencies to other transformation rules, their instantiation or patterns that declaratively specify when the transformation rule should be called. Other than that, the transformation rules may override a method that is called to initialize the transformation rule result. Similar to ATL, NTL also allows transformations to be based on other transformation rules overriding some of their transformation rules. This technique is called superimposition in ATL [11], in NTL it is called transformation rule inheritance as it is realized in inheriting the transformation rule classes.

The fact that NTL allows to target models not formally described by an NMeta metamodel yields the potential to reuse existing code and especially analyses. As an example transformation, the NMF code generator, probably the largest model transformation ever written in NTL, transforms NMeta metamodels into code models of the .NET framework `System.CodeDOM` object model. From this object model, code can be generated in a multitude of languages so that our code generator does not only support to generate C# model representation code but also code for other languages like VB.NET, C++ and F#. Here, the .NET framework code to generate code based on a `System.CodeDOM` model can be reused to generate the model representation code. This not only has the advantage that we do not have to write code generators for each of the supported languages separately, but also means that we can inherit support for more exotic .NET languages such as Boo, as soon as they support a `System.CodeDOM` code generator.

This code generator is also a showcase for the extensibility of NTL as it is a refinement of a more general code generator component that resolves multiple inheritance into interfaces and implementation classes, replicating the code if necessary, also part of the NMF framework. This code generator can be reused if model representation code should be generated from other models than NMeta metamodels.

6 Model Synchronization

Based on NTL and NMF Expressions, NMF also contains a language to synchronize models of heterogeneous metamodels, named NMF Synchronizations [12]. Like NTL, it is also implemented as an internal DSL so that developers can familiarize quickly. This synchronization language is able to support 18 different operation modes out of a single specification: One may choose between three different change propagation modes (none, one-way and two-way) and six different directions (left-to-right and right-to-left in three different variants each). Also much like NTL, it supports the synchronization of arbitrary CLR types, though we require them to correctly implement `INotifyPropertyChanged` and `INotifyCollectionChanged` correctly, e.g. by using the model representation generated by NMF from an NMeta metamodel.

Similar to NTL, a synchronization rule in NMF Synchronizations is represented by a class, inferring the synchronization rules by the public nested classes. The synchronization rules each define an isomorphism between the classes they are to synchronize, referred to as left-hand-side (LHS) and right-hand-side (RHS) class. These classes are passed as generic type parameters.

Unlike NTL, NMF Synchronizations does not allow the synchronization body to contain arbitrary code but only expressions without side-effects. Instead, developers can specify a list of synchronization jobs through the specification of lambda expressions from LHS and RHS. If the result types of these lamda expressions differs, an isomorphism between them must be specified in the form of a synchronization rule between them. Depending on the operation mode selected, these lambda expressions are then compiled, inverted, or a dynamic dependency graph is created from them, potentially supporting inversion as well. For example, if the operation mode is two-way change propagation, then an invertible dynamic dependency graph is created for the expressions on both sides.

7 Conclusion

In this paper, we have given an overview on NMF, a framework to support model-driven engineering on the .NET platform. In particular, the framework consists of tools to transform Ecore metamodels to NMeta, generate appropriate model representation code and analyze, transform and synchronize the models. This enables practicioners of MDE to reuse existing code available in .NET to process models while existing EMF models can be reused.

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] M. Staron, “Adopting model driven software development in industry - a case study at two companies,” in *MoDELS*, 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] G. Hinkel and L. Happe, “Using component frameworks for model transformations by an internal DSL,” in *Proceedings of the 1st International Workshop on Model-Driven Engineering for Component-Based Software Systems co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*, ser. CEUR Workshop Proceedings, vol. 1281, CEUR-WS.org, 2014, pp. 6–15.

[5] G. Hinkel, T. Goldschmidt, and L. Happe, “An NMF solution for the Petri Nets to State Charts case study at the TTC 2013,” *EPTCS*, vol. 135, pp. 95–100, 2013.

[6] ——, “An NMF solution for the Flowgraphs case at the TTC 2013,” *EPTCS*, vol. 135, pp. 37–42, 2013.

[7] G. Hinkel and L. Happe, “An NMF Solution to the TTC Train Benchmark Case,” in *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L’Aquila, Italy, July 24, 2015*, ser. CEUR Workshop Proceedings, vol. 1524, CEUR-WS.org, 2015, pp. 142–146.

[8] G. Hinkel, “An NMF Solution to the Java Refactoring Case,” in *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L’Aquila, Italy, July 24, 2015*, ser. CEUR Workshop Proceedings, vol. 1524, CEUR-WS.org, 2015, pp. 95–99.

[9] ——, “An approach to maintainable model transformations using an internal DSL,” Master’s thesis, Karlsruhe Institute of Technology, 2013.

[10] G. Hinkel, T. Goldschmidt, and L. Happe, “Tool Support for Model Transformations: On Solutions using Internal Languages,” in *Modellierung 2016, Karlsruhe, Germany, 2–4 March 2016*, to appear, 2016.

[11] D. Wagelaar, R. Van Der Straeten, and D. Deridder, “Module superimposition: a composition technique for rule-based model transformation languages,” *Software & Systems Modeling*, vol. 9, no. 3, pp. 285–309, 2010.

[12] G. Hinkel, “Change Propagation in an Internal Model Transformation Language,” in *Theory and Practice of Model Transformations*, Springer, 2015, pp. 3–17.

A Tutorial of NMF

The following instructions will describe how to set up a project with NMF and create a model transformation from state machines to Petri nets. Although the tutorial is specifically written for a usage in Visual Studio, the tutorial can be adapted to any IDE on the .NET platform. The tutorial also assumes that you have already started to create a metamodel and some instances of it in EMF, i.e. Eclipse.

If you get stuck at any point, there is a ready-made solution available on GitHub[5] that can be just downloaded and tried. Furthermore, there is a YouTube video available[6] demonstrating creating a new project, loading, altering and saving a model.

A.1 Create a Project

NMF is a framework that can be easily installed through the NuGet Package-manager[7]. Therefore, first create a new project. In Visual Studio, click on *File →New → Project*, select a C# console application as project type and name it as you wish, though in the remainder we will assume the name *NMFDemo*. Note that NMF is generally not restricted to console applications nor to C#, you can use it in any .NET project.

To import NMF, go to *Tools→NuGet Package-manager→Manage NuGet packages for this project* and search for *NMF*. You should find the package **NMF-Basics**. Install it by hitting the *Install* button while your project is selected.

Alternatively, there is also a NuGet console at the bottom, where you can install NMF as follows:

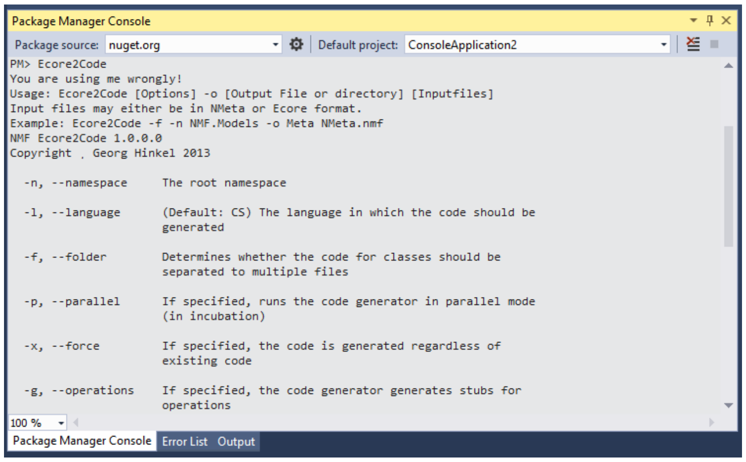
```
PM> Install-Package NMF-Basics
```

NuGet will download the package for you together will all of its dependencies and add all the contained libraries as references into the current project. There is no strict 1:1-mapping from NuGet packages to libraries so there are multiple libraries being installed that may be not needed.

A.2 Import Metamodels from Ecore

Metamodels are at the core of any model-driven development process. Thus, as a first step, we will generate code in order to be able to load any models for a given metamodel in our .NET application. For this, the NuGet package NMF-Basics contains the console application Ecore2Code. After a restart of Visual Studio, NuGet will automatically add Ecore2Code to the Path variable used inside Visual Studio, so you can just use the NuGet Package-manager console. If run without any arguments, this application prints a help information showing its correct usage (cf. Fig. 2).

[5] `https://github.com/NMFCode/NMFDemo`
[6] `https://youtu.be/NIMYuwTltVs`
[7] `http://www.nuget.org`



```
PM> Ecore2Code
You are using me wrongly!
Usage: Ecore2Code [Options] -o [Output File or directory] [Inputfiles]
Input files may either be in NMeta or Ecore format.
Example: Ecore2Code -f -n NMF.Models -o Meta NMeta.nmf
NMF Ecore2Code 1.0.0.0
Copyright , Georg Hinkel 2013

  -n, --namespace     The root namespace

  -l, --language      (Default: CS) The language in which the code should be
                      generated

  -f, --folder        Determines whether the code for classes should be
                      separated to multiple files

  -p, --parallel      If specified, runs the code generator in parallel mode
                      (in incubation)

  -x, --force         If specified, the code is generated regardless of
                      existing code

  -g, --operations    If specified, the code generator generates stubs for
                      operations
```

Figure 2. The console application Ecore2Code to generate model representation code.

Now, use this tool to generate the code for the state machine metamodel and the Petri net metamodel. You can download these metamodels from our examples project[8]. First copy the metamodels into your project folder, then generate the code for them. The complete commandline for the latter is as follows:

```
PM> Ecore2Code -f -n NMFDemo.Metamodels -m fsm.nmf -o Metamodels\FiniteStateMachines fsm.ecore
PM> Ecore2Code -f -n NMFDemo.Metamodels -m pn.nmf -o Metamodels\PetriNets pn.ecore
```

The generated code now has to be added to your project. Thus, first display all files in the projects folder by clicking on *Show All Files* in the project explorer, then include the generated folder *Metamodels* and the generated NMeta metamodels into your project (right-click and *Include In Project*).

As soon as the generated code is added to the project, it is already possible to programatically create and save models. However, the metamodel is not yet registered and thus no models can be loaded. To register the metamodel, we first need to include the NMeta metamodel in the assembly as an embedded resource and then registers the metamodel. To make the metamodel an embedded resource, simply change its *Build Option* to *Embedded Resource* in the properties view while the metamodel is selected.

The metamodel registration is done through an assembly-wide attribute, which can be specified anywhere in the project. The typical place for this registration, however, would be the *AssemblyInfo.cs* file in the properties folder. At the top of this file, add the following two lines:

```
[assembly: NMF.Models.ModelMetadata("http://github.com/NMFCode/Examples/FiniteStateMachines", "NMFDemo.
      fsm.nmf")]
[assembly: NMF.Models.ModelMetadata("http://github.com/NMFCode/Examples/PetriNets", "NMFDemo.pn.nmf")]
```

This is all there is, even if you compile your project not as an executable but as a reusable library. As a reason, when loading the serializer, NMF looks for

[8] https://github.com/NMFCode/NMFDemo

these attributes in all assemblies referenced by the executing assembly and loads any metamodel registrations it can get.

A.3 Loading a Model

In NMF, models are loaded by resolving their URI in a model repository. If the repository does not contain a model with the given URI, then the model is automatically loaded into the repository, provided NMF is able to locate it. Repositories are closed under cross-reference, meaning that all references to other model elements are always resolved within the repository or its parent repository.

To create a repository, we simply need to create an object of type `ModelRepository`. With the default configuration, this repository is able to deserialize any models conforming to metamodels registered in referenced assemblies, as all repositories implicitly use the meta repository where the metamodels are loaded into.

```
var repository = new ModelRepository();
var model = repository.Resolve("Example.fsm");
var fsm = model.RootElements[0] as FiniteStateMachine;
```

Listing 1. Loading Models in NMF

For example, the code needed to load a model from the file *Example.fsm*[9] representing a small order process is depicted in Listing 1. Add these lines to the main method. You can now launch the application and validate that the model can be loaded successfully.

A.4 Incrementalization

The generated model representation classes for the metamodel support change notifications through the .NET de-facto standard interfaces `INotifyPropertyChanged` and `INotifyCollectionChanged`. Thus, the generated classes raise events whenever some properties have been changed or elements have been added to or removed from collections. NMF is able to combine these elementary change notifications to deduct when the value for a combined expression has changed.

For example, let us analyze hubs in the finite state machines, i.e. states that have the maximum incoming transitions. A set of such states can be deducted through the analysis depicted in Listing 2.

```
var stateHubs = from s in fsm.States
   where s.Incoming.Count == fsm.States.Max(s2 => s2.Incoming.Count)
   select s.Name;
```

Listing 2. Analyzing which states are hubs

Verify that the variable `stateHubs` is of type `IEnumerable<string>`, i.e. a standard collection of strings. Now, we need to add a using statement at the top of the program file to the Linq implementation of NMF Expressions. Add the code from Listing 3 to the top of the program file.

[9] `https://github.com/NMFCode/NMFDemo/blob/master/Example.fsm`

```
using NMF.Expressions.Linq;
```

Listing 3. Registering the Linq implementation of NMF Expressions

As a consequence, the Linq implementation of NMF Expressions is used and thus, the variable `stateHubs` has the type `IEnumerableExpression<string>`. This adds a method to obtain an incrementalized version of the query through the `AsNotifiable` method.

```
stateHubs.AsNotifiable().CollectionChanged += (o,e) => {
  if (e.NewItems != null)
    for (string name in e.NewItems) { Console.WriteLine("{0}␣is␣a␣new␣hub", name); }
  if (e.OldItems != null)
    for (string name in e.OldItems) { Console.WriteLine("{0}␣is␣no␣longer␣a␣hub", name); }
};
```

Listing 4. Adding handlers when the analysis results have changed

To verify the change propagation, visualize changes made to the state hub analysis through the code shown in Listing 4. Normally, the method `AsNotifiable` is a very expensive operation, thus one would save the return value.

```
1  var checkStock = fsm.States[1];
2  checkStock.Outgoing.Add(new Transition() {
3    Input = "items␣are␣for␣free",
4    Target = fsm.States[2]
5  });
```

Listing 5. Adding a new transition to imply a new hub

Add some change operations after registering the handler, step through the console application and see how new hubs are immediately shown in the console. For example, you can use the code listed in Listing 5 to create a new transition to skip payment when items of the order process are for free. As a consequence of this change, a message will pop up in the console that a new hub has been detected directly afterwards Line 2-5 of Listing 5 have been executed.

A.5 Creating a Model Transformation

Now, we are going to transform the state machine model into a different model, for instance in a Petri net.

At first, we need to add the libraries to run model transformations in NTL. The easiest way to get them is to download them as another NuGet package. Install NMF Transformations through the NuGet command shown in Listing 6 or again through the GUI.

```
PM> Install-Package NMF-Transformations
```

Listing 6. Installing NMF Transformations

A model transformation in NMF Transformations is a special class, inheriting from `ReflectiveTransformation`. Thus, create a new class by adding a new class to the project. Download the model transformation `FSM2PN` from finite state machines to Petri nets from the examples page[10] and copy its contents into the new file.

To run this model transformation, we need to instantiate the model transformation, initialize it and run it. The initialization can be reused for multiple passes of a model transformation, in case the model transformation initialization is costly. To apply the model transformation, we need to pass the source and target model type as generic parameters. The transformation then selects an appropriate rule to start with and traverses the transformation through the rule dependencies. Thus, we can ask the transformation to transform states or entire state machines.

```
var transformation = new FSM2PN();
var context = new TransformationContext();
var petriNet = TransformationEngine.Transform<StateMachine, Net>(fsm, context);
```

Listing 7. Running NMF Model Transformations

After the transformation, the `context` object can be used for tracing purposes.

A.6 Saving result model to a file

In NMF, the serialization information of model elments is attached directly to the model representation classes. The NMF serializer uses this information and interprets how the model should be serialized to XMI. To serialize the Petri net, we simply save it into our model repository (or create a new one). Any referenced model element already contained in another existing file is referenced through a fully qualified reference.

```
repository.Save(petriNet, "Example.pn");
```

Listing 8. Serializing models in NMF

To save a model element to a file, it is sufficient to call the Save method on the repository such as shown in Listing 8. Verify that you can open this file in Eclipse.

[10] `https://github.com/NMFCode/NMFDemo/blob/master/FSM2PN.cs`