Hochschule RheinMain

TWO SOLUTIONS TO INCREMENTAL CLASS TO RELATIONS CASE

Plain C#, NMF Synchronizations

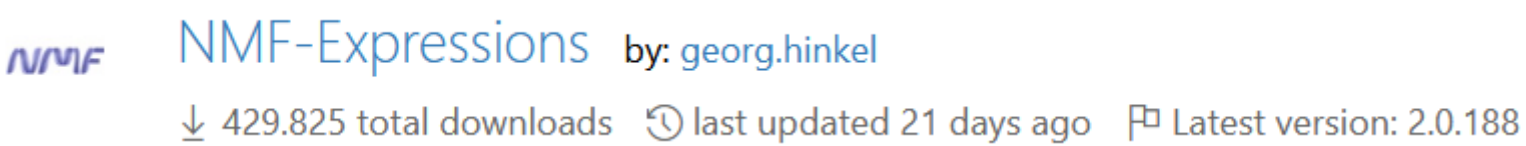
Prof. Dr. Georg Hinkel
20.07.2023

NMF
.NET Modeling Framework

Hochschule **RheinMain**

- Modelling platform for the .NET platform
  - Generate code from NMeta or Ecore metamodels
  - Load and save XMI models
  - Integrated incrementalization capabilities

- Open Source
  - Apache 2.0
  - https://github.com/NMFCode/NMF



Hochschule RheinMain

PLAIN C#

Including Dynamic Language Runtime (DLR) features

TRACING

Hochschule RheinMain

```
private Dictionary<object, IModelElement> _trace = new();
private object? TraceOrTransform(object item)
{
    if (item == null) return null;
    if (!_trace.TryGetValue(item, out var transformed))
    {
        transformed = Transform((dynamic)item);
        _trace.Add(item, transformed);
    }
    return transformed;
}
```

Break out of static type system,
Method overload is selected at runtime based on dynamic type

MODEL NAVIGATION

Hochschule **RheinMain**

```
foreach (var tableValuedAttribute in
    from cl in classModel.RootElements.OfType<IClass>()
    from att in cl.Attr
    where att.MultiValued
    select att)
{
    result.RootElements.Add(CreateAttributeTable(tableValuedAttribute));
}
```

CHANGE PROPAGATION

A simple case

Hochschule **RheinMain**

```
var type = new Type
{
    Name = dataType.Name
};
dataType.NameChanged += (o, e) => type.Name = dataType.Name;
```

Problematic:

- Easy to forget
- Duplicated logic
- Decreases understandability

CHANGE PROPAGATION

A more sophisticated example

Hochschule **RheinMain**

```
void OnNameChanged(object? sender, ValueChangedEventArgs? e)
{
   table.Name = attribute.Owner.Name + "_" + attribute.Name;
   key.Name = attribute.Owner.Name.ToCamelCase() + "Id";
}
OnNameChanged(null, null);
attribute.Owner.NameChanged += OnNameChanged;
attribute.OwnerChanged += (o, e) =>
{
    if (e.OldValue != null) ((IClass)e.OldValue).NameChanged -= OnNameChanged;
    OnNameChanged(o, e);
    if (e.NewValue != null) ((IClass)e.NewValue).NameChanged += OnNameChanged;
};
```

Change of owner relevant?
Also not in the future?

CHANGE PROPAGATION

Hochschule RheinMain

```
foreach (var tableValuedAttribute in
    from cl in classModel.RootElements.OfType<IClass>()
    from att in cl.Attr
    where att.MultiValued
    select att)
{
    result.RootElements.Add(CreateAttributeTable(tableValuedAttribute));
}
```

Implementing change propagation is hard,
loses query syntax

REVIEW PLAIN C# SOLUTION

Hochschule **RheinMain**

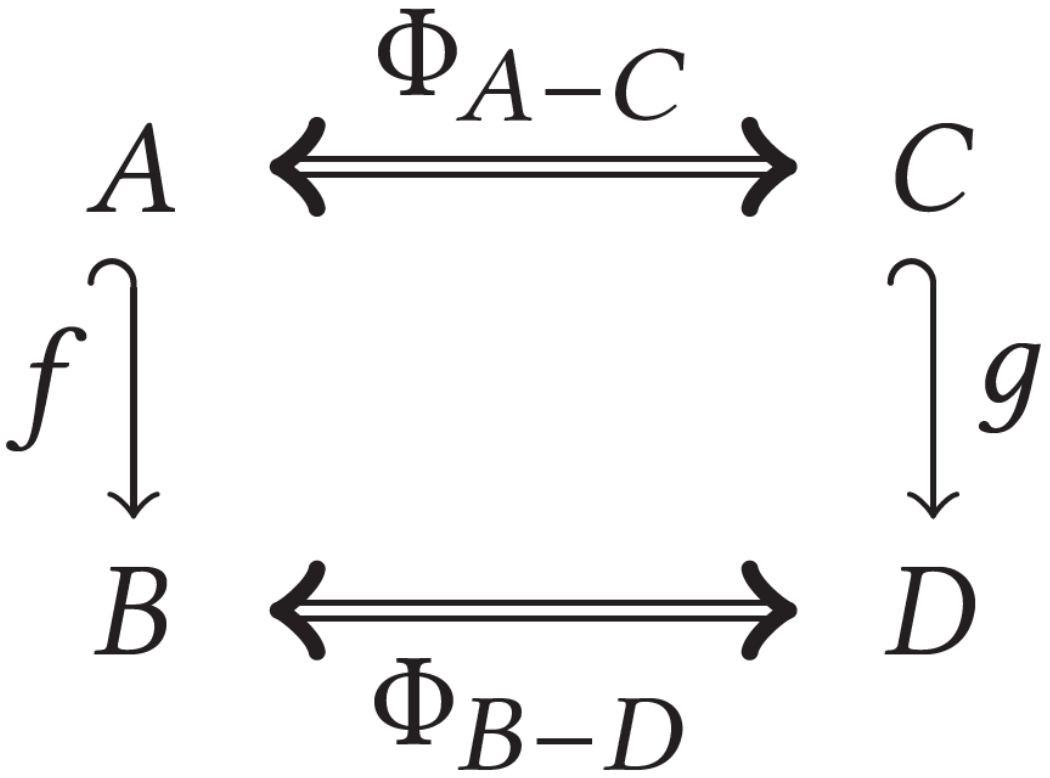
- **Tracing in C# is not a problem**
- **Model navigation in C# is not a problem**
- **Implementing change propagation is hard**
  - Solution is incomplete w.r.t. change propagation

Hochschule RheinMain

NMF SYNCHRONIZATIONS

NMF SYNCHRONIZATIONS

Hochschule **RheinMain**



- Model synchronization language and framework
- Algebraic model of synchronization blocks
  - Proved correctness
  - Proved hippocraticness
- Declarative
  - Uni- or bidirectional execution
  - Optional change propagation
  - Check-only mode
- Internal DSL in C#

ISOMORPHISMS

Hochschule RheinMain

- First step: Identify isomorphisms (aka correspondences)
  - Entire class model corresponds to entire relational model
  - A class corresponds to a table
  - A data type corresponds to a type
  - An attribute corresponds to a column
  - An attribute corresponds to a table, if and only if it is multi-valued
- Isomorphisms are reflected in rules

```
public class ClassToRelational :
  ReflectiveSynchronization
{
  public class MainRule :
    SynchronizationRule<Model, Model> …
  public class ClassToTable :
    SynchronizationRule<IClass, ITable>
  public class DataTypeToType :
    SynchronizationRule<IDataType, IType>
  public class AttributeToColumn :
    SynchronizationRule<IAttribute, IColumn>
  public class AttributeToTable :
    SynchronizationRule<IAttribute, ITable>
}
```

SYNCHRONIZATION BLOCKS

Hochschule RheinMain

```
SynchronizeManyLeftToRightOnly(SyncRule<AttributeToTable>(),
    m => from c in m.RootElements.OfType<IClass>()
         from a in c.Attr
         where a.MultiValued
         select a,
    rels => rels.RootElements.OfType<IModelElement, ITable>());
```

Same query as before, this time with change propagation

SYNCHRONIZATION BLOCKS (CONT.)

Hochschule RheinMain

```
SynchronizeLeftToRightOnly(a => a.Owner.Name + "_" + a.Name, t => t.Name);
SynchronizeLeftToRightOnly(a =>
    a.Owner.Name.ToCamelCase() + "Id", t => t.Col[0].Name);
```

LeftToRightOnly to allow to use expressions that NMF is unable to invert

STARTING THE SYNCHRONIZATION

Hochschule **RheinMain**

```
ClassToRelational synchronization = new ClassToRelational();

Model target = null;
synchronization.Synchronize(ref inputModel, ref target,
     SynchronizationDirection.LeftToRight,
     ChangePropagationMode.OneWay);
```

CONCLUSION

Hochschule **RheinMain**

- The plain C# solution…
  - …shows that supporting a trace in plain C# is easy
  - …shows that model navigation in plain C# is easy
  - …shows that change propagation in plain C# is **not** easy

- The NMF Synchronizations solution…
  - …requires developers to think in terms of abstractions (isomorphisms)
  - …is based on a proven algebraic framework to ensure correctness
  - …makes change propagation invisible to the developer

Hochschule RheinMain

THANKS FOR YOUR ATTENTION

Prof. Dr. Georg Hinkel, georg.hinkel@hs-rm.de