Change Propagation in an Internal Model Transformation Language

Georg Hinkel
Sparse adoption of MDE in industry

- Tool support perceived insufficient [Sta06, Mo+13]
  - Much less manpower in tool development than IDEs such as Visual Studio, IntelliJ, ...

- Developers hardly change their primary language [MR13]
  - Project requirements or code reuse
Promising approach: Internal DSLs

- Inherit tool support from host language
- Let developers stay with their primary language

- Downside: No change propagation, few bidirectional approaches
  - Change Propagation $\rightarrow$ Performance, Efficiency
  - Bidirectionality $\rightarrow$ Applicability, Reuse
Related Work

- **Self-adjusting Computation**
  - Explicit [Acar05, Bur+11]
  - Implicit [Che+14]
  - No abstractions for model transformation

- **Internal Model Transformation Languages**
  - RubyTL [CMT06]
  - FunnyQT [Hor13]
  - NTL [Hin13]
  - No Change Propagation, few bidirectionality

- **Model Transformation with Change Propagation**
  - TGG [GW06]
  - ... 
  - No internal Languages, other transformation paradigms
How to obtain Change Propagation?

- Combine Self-adjusting Computation with Model Transformation
  - Model Transformation languages provide domain-specific abstractions
  - Self-Adjusting Computation provides techniques for change propagation

- Implementations for both as internal DSLs in .NET Modeling Framework (NMF)
  - Model Transformation: NMF Transformations
  - Self-adjusting Computation: NMF Expressions
  - Reversibility extensions for NMF Expressions
NMF Transformations

Transformation

+ Initialize()

transformation 1

rules 0..*

TransformationRule

+ InputType : Type[]
+ OutputType : Type
+ CreateOutput : Function
+ Transform : Function

SingleDependency

MultipleDependency

Dependency

+ ExecuteBefore : Boolean
+ Filter : Function
+ Selector : Function
+ Persistor : Function

# HandleDependency()
NMF Expressions

- Self-adjusting computation at instruction level
- Based on C# Expression Trees

- \( p.\text{Age} > 18 \)
Putting it all together

- **NMF Synchronizations**
  - Transformation Engine of NMF Transformations
  - Elementary Change Propagation through NMF Expressions
  - Reversability through Lense-extensions of NMF Expressions

- **Internal Model Transformation & Synchronization DSL in C#**
- **18 Modes out of a single specification**
Transformation Directions

- **LeftToRight**
  - Transformation enforces that model elements exist in RHS if a corresponding element is in LHS

- **LeftToRightForced**
  - LeftToRight + Elements on RHS with no corresponding element in LHS are deleted

- **LeftWins**
  - LeftToRight + Elements on RHS with no corresponding element in LHS are transformed to LHS

- **RightToLeft, RightToLeftForced, RightWins**
Change Propagation Modes

- **None**
  - Change Propagation is disabled

- **OneWay**
  - Change Propagation from Source to Target

- **TwoWay**
  - Change Propagation in both direction
Decisions for Expressions in NMF Expressions

1. Compile expression and run it normally
2. Reverse expression, compile and run normally
3. Create Dynamic Dependency Graph
   - Receive Incremental Updates
4. Create Reversible Dynamic Dependency Graph
   - Receive Incremental Updates
   - Push Updates backwards through the graph
Finite State Machines and Petri Nets

Motivation
Foundations
Approach
Example
Validation
public class PSM2PN : ReflectiveSynchronization
{
    public class AutomataToNet : SynchronizationRule
        <FiniteStateMachine, PetriNet>
    
    {...}

    ...

}
public override void DeclareSynchronization()
{
    SynchronizeMany(SyncRule<StateToPlace>(),
        fsm => fsm.States, pn => pn.Places);
    SynchronizeMany(SyncRule<TransitionToTransition>(),
        fsm => fsm.Transitions,
        pn => pn.Transitions.Where(t => t.To.Count > 0));
    ...
    Synchronize(fsm => fsm.Id, pn => pn.Id);
}
Validation goals

- Validate Correctness
  - Generate Change sequences
  - Propagate changes or retransform input
  - Comparison against NMF Transformations solution

- Validate Performance
  - Response time to apply full change sequence & propagate changes
  - 100 change sequences á 100 elementary changes
Composition of the change sequence

- Add a state to the finite state machine (30%)
- Add a transition to the finite state machine with random start and end state (30%)
- Remove a random state and all of its incoming and outgoing transitions (10%)
- Remove a random transition from the finite state machine (10%)
- Toggle end state of a random state (5%)
- Change the target state of a randomly selected transition to a random other state (5%)
- Rename a state (9%)
- Rename the finite state machine (1%)
Results

![Graph showing performance comparison between NMF Transformations, NMF Synchronizations (Batch), and NMF Synchronizations (Incremental). The graph plots Total Response Time [ms] against Size (n).]
Limitations

- Correspondences fixed once created
  - Not a general restriction
Conclusion

- Proof-of-concept: Change Propagation and Bidirectional Transformations possible with Internal Model Transformation Languages
- 18 operation modes from single specification
- Speedups of up to 48 measured
## References I

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title and Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sta06</td>
<td>M. Staron</td>
<td>“Adopting model driven software development in industry—a case study at two companies,” in <em>Model Driven Engineering Languages and Systems</em>, Springer, 2006, pp. 57–72</td>
</tr>
<tr>
<td>Reference</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td></td>
</tr>
</tbody>
</table>
Reversible Expressions through Lenses

- Algebraic construct consisting of two functions
  - Get $l \downarrow: A \rightarrow B$
  - Put $l \uparrow: A \times B \rightarrow A$

\[ A, B \in ob \mathcal{C} \]

- Specification of Lenses through attributes
  - Allow developers to specify how to revert function calls

```csharp
[ObservableProxy(typeof(ObservableFirstOrDefault<>), "CreateExpression")]
[SetExpressionRewriter(typeof(ObservableFirstOrDefault<>), "CreateSetExpression")]
public static TSource FirstOrDefault<TSource>(this IEnumerableExpression<TSource> source)
```

[Fo+05]
public override bool ShouldCorrespond
    (FSM.Transition left, PN.Transition right,
     ISynchronizationContext context)
{
    var stateToPlace = SyncRule<StateToPlace>().LeftToRight;
    return left.Input == right.Input
        && right.From.Contains
            (context.Trace.ResolveIn(stateToPlace, left.StartState))
        && right.To.Contains
            (context.Trace.ResolveIn(stateToPlace, left.EndState));
}
public override void DeclareSynchronization()
{
    SynchronizeLeftToRightOnly(SyncRule<StateToPlace>(),
        state => state.IsEndState ? state : null,
        transition => transition.From.FirstOrDefault());
}