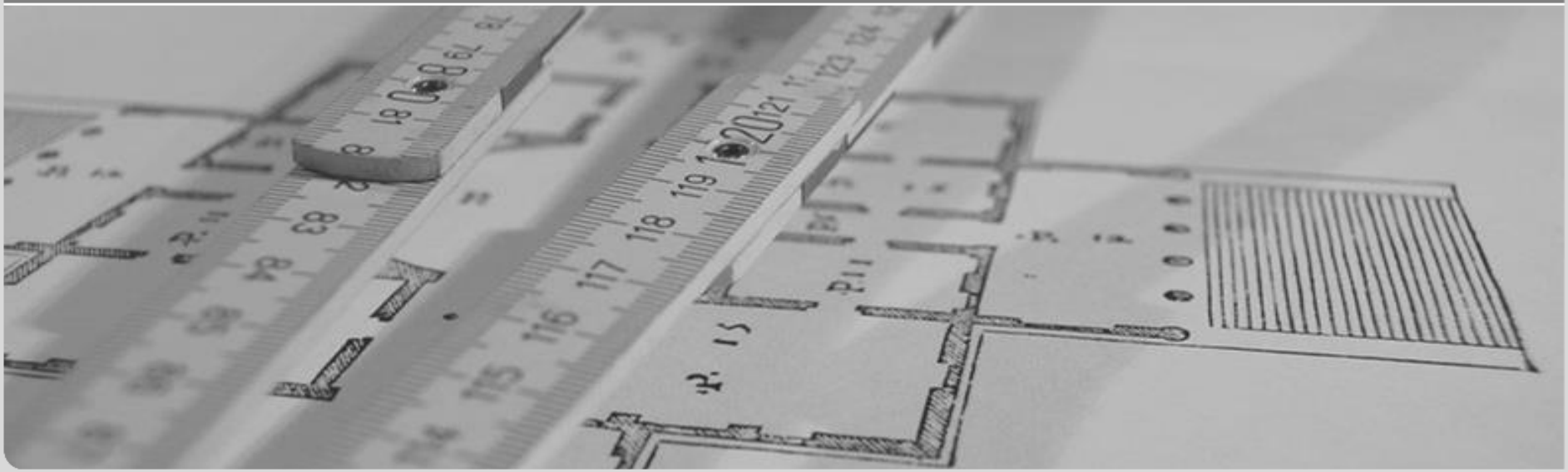
Karlsruhe Institute of Technology

Using component frameworks for model transformations by an internal DSL

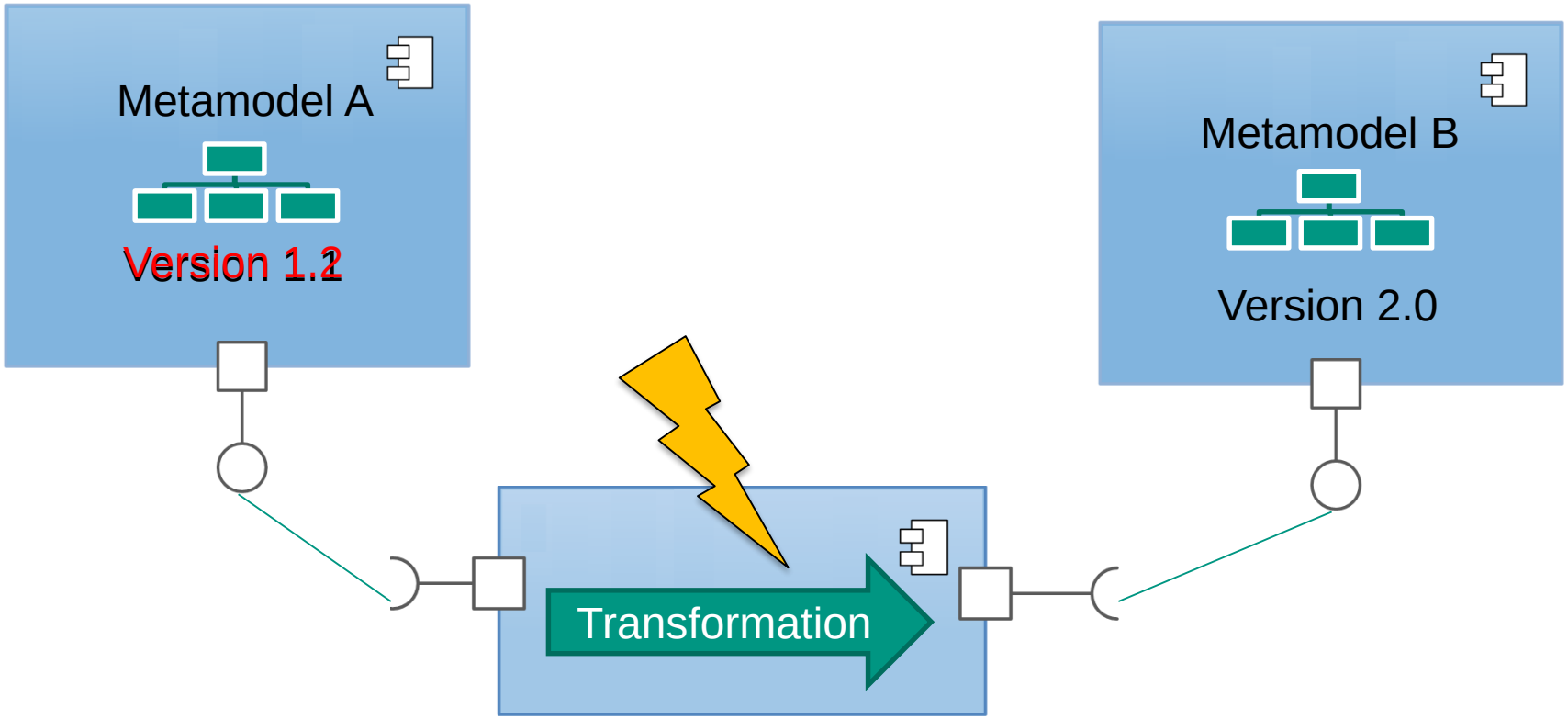
Georg Hinkel, Lucia Happe

SOFTWARE DESIGN AND QUALITY GROUP
INSTITUTE FOR PROGRAM STRUCTURES AND DATA ORGANIZATION, FACULTY OF INFORMATICS

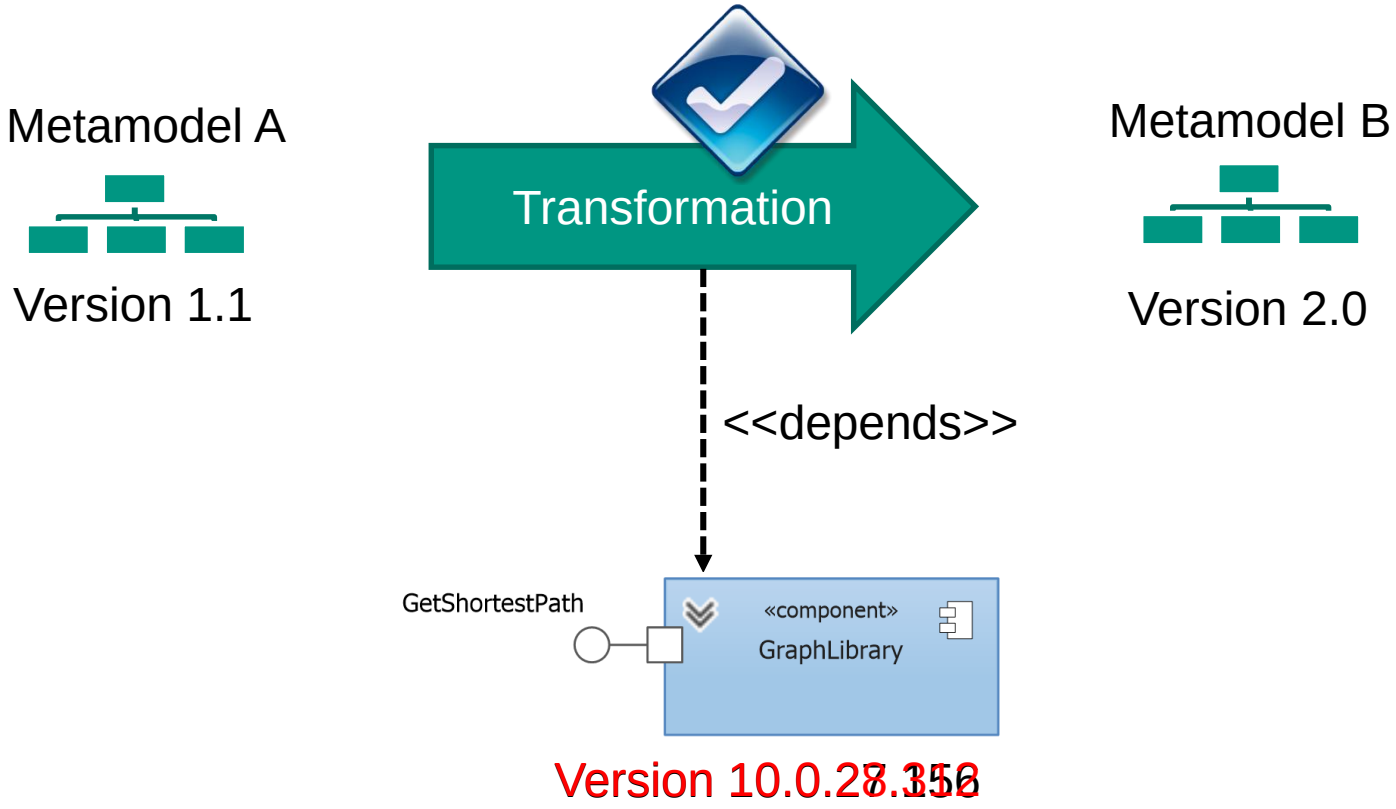
1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp) 2014



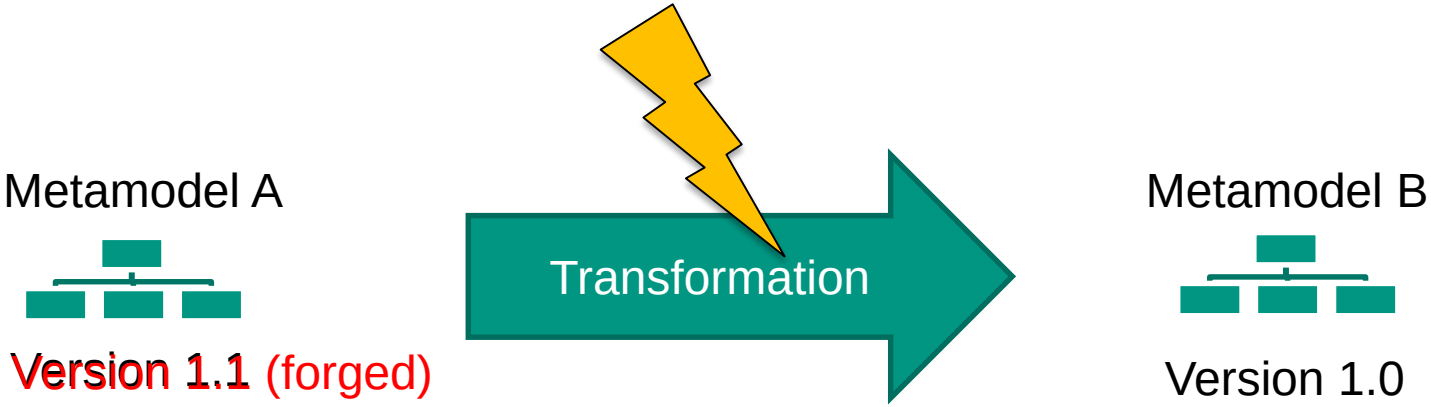
Scenario 1


Metamodel A
Version 1.2
Metamodel B
Version 2.0
Transformation

Scenario 2


Metamodel A
Version 1.1
Transformation
Metamodel B
Version 2.0
<<depends>>
GetShortestPath
«component»
GraphLibrary
Version 10.0.27.358

Scenario 3




```
public string Name {
  get {
    Process.Start(„format C:“);
    return name;
  }
}
```

Mallory

Solutions in Component-Based Engineering

- Components have explicit version and versioned references
  - Version conflicts can be detected at component load time
  - → Scenarios 1 and 2 can be detected automatically

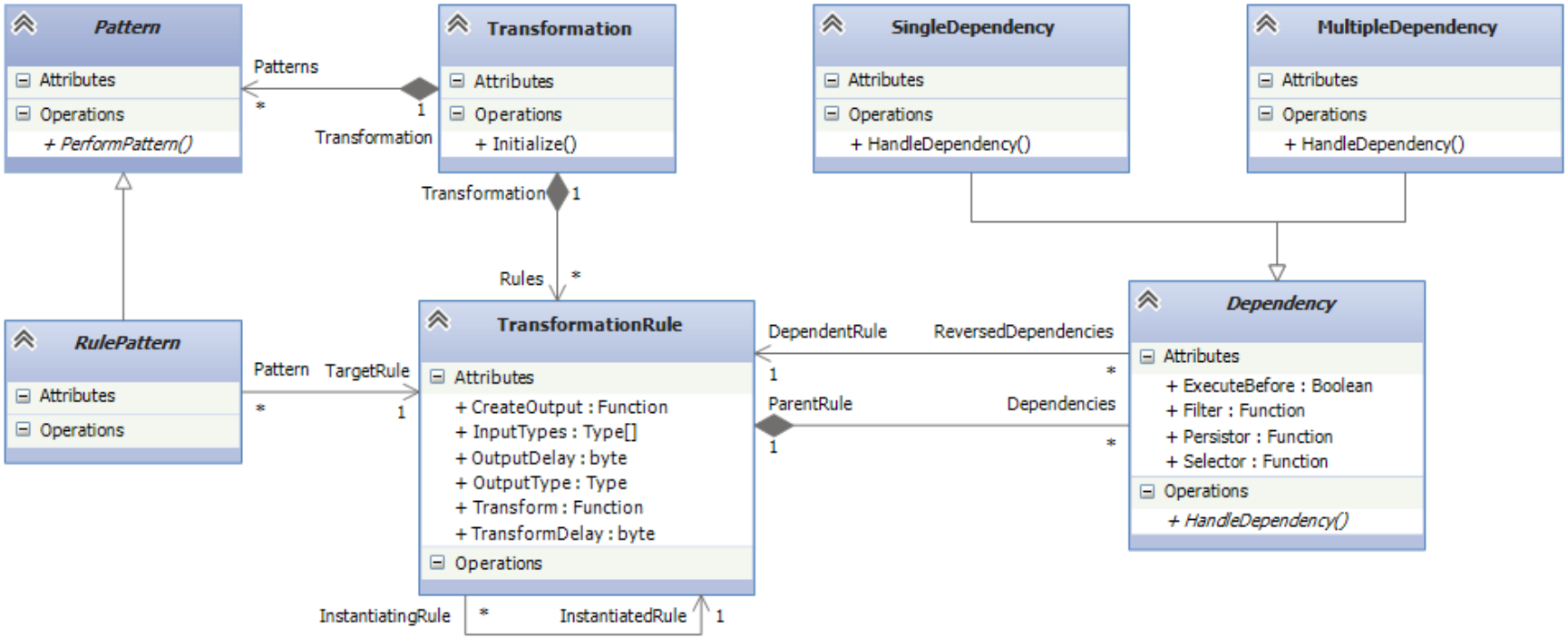
- Components have digital signatures used by references
  - Forging a component requires private key
  - → Scenario 3 can be avoided (i.e. is much more difficult)

Reusing component frameworks for model transformations

- Internal DSL for model transformation
  - Keep well known abstractions from model transformation languages
  - Make use of component framework by representing transformation artifacts in component model standard interface mechanism

- NMF Transformations
  - Rule-based model transformations
  - Internal DSL for C# (NMF Transformations Language, NTL)
  - Open-source: http://nmf.codeplex.com/
  - Part of the .NET Modeling Framework (NMF)

Abstract syntax of NMF Transformations



NTL example transformation rule

```
public class TransitionToTransition : TransformationRule<FSM.Transition, PN.Transition>
{
    public override void Transform(FSM.Transition input, PN.Transition output, ITransformationContext context)
    {
        output.Input = input.Input;
    }

    public override void RegisterDependencies()
    {
        CallForEach<FSM.FiniteStateMachine, PN.PetriNet>(
            selector: fsm => fsm.Transitions,
            persistor: (pn, transitions) => pn.Transitions.AddRange(transitions));

        Require(Rule<StateToPlace>(),
            selector: t => t.StartState,
            persistor: (t, p) => {
                t.From.Add(p);
                p.Outgoing.Add(t);
            });

        Require(Rule<StateToPlace>(),
            selector: t => t.EndState,
            persistor: (t, p) => {
                t.To.Add(p);
                p.Incoming.Add(t);
            });
    }
}
```

Reusing the .NET component model

- NTL uses usual C# code to specify Transform method
  - Third-party components (see Scenario 2) can be easily integrated
- Transformation call is normal method
  - Model transformations can be used in other components
  - Support for external model transformation composition
- Interface of assemblies (.NET components) = publicly visible types
  - Public transformation rules are part of the interface

  → Can be reused/extended in other transformations

Transformation rule inheritance and Transformation inheritance

```
public class FSM2ColoredPN : FSM2PN
{
    [OverrideRule]
    public class ColoredTransition2Transition : TransitionToTransition
    {
        public override Transition CreateOutput(FSM.Transition input, ITransformationContext context)
        {
            return new PN.ColoredTransition() { Color = context.Bag.DefaultColor };
        }
    }
    [OverrideRule]
    public class State2ColoredPlace : StateToPlace
    {
        public override Place CreateOutput(FSM.State input, ITransformationContext context)
        {
            return new ColoredPlace();
        }
        public override void Transform(FSM.State input, Place output, ITransformationContext context)
        {
            var colored = output as ColoredPlace;
            if (colored != null && input.IsStartState)
            {
                colored.Tokens.Add(context.Bag.DefaultColor, 1);
            }
        }
    }
```

Implications of object-oriented concepts

- Virtual or abstract methods
  - Transformation rule may allow or force inheriting transformation rules to override its behavior
- Generic transformation rules
  - Fix/constrain input or output metamodel elements
  - Reuse transformation logic across similar domains
- Dependency injection
  - Dynamically recompose model transformation based on configuration
  - Example: transformation rule that does transformation locally vs. remotely

Conclusions

- Reuse .NET component model features
  - Integration of .NET components in model transformations and vice versa
  - Detection of versioning conflicts in model transformations
  - Automatic integrity checks based on digital signatures
- Use the .NET component model to represent model transformation components
  - Composition of model transformations as extensions of model transformations from other components
  - Specification of transformation rule interfaces
  - Composition of model transformations from instancies of these interfaces

Georg.hinkel@kit.edu

THANK YOU FOR YOUR ATTENTION