VMTS

Tihamér Levendovszky

Supporting Model-Based Software Engineering with Domain-Specific Languages

Budapest University of Technology and Economics
Department of Automation and Applied Informatics
Visual Modeling Languages Research Group
Institute for Software-Integrated Systems, Vanderbilt University

Outline

- Introduction
  - SE on a higher abstraction level
  - Generative Programming
- VMTS
  - Abstract and Concrete Syntax
  - Constraint optimization
  - Animation
  - Optimized Transformations
  - Validated transformations
  - Code-Model/Model-Code Synchronization
  - Domain-Specific Model Patterns

Raising the abstraction level

- Evolution of programming languages
  - Assembly → C → C++/ Java/ C#
- The aims
  - Faster development
    - == compact way to express our aims
  - To avoid steps that can be automated
    - == abstraction level must be increased
  - To develop larger systems
    - == even complex functions must be easy to understand

Generative Programming

- Overview
  - Aims at a narrow domain
  - Models the variability (all possible configurations)
  - Generator takes the desired configuration
- Evaluation
  - Essentially the onle approach really supports reuse
  - Pays off when the generator is used several times
- DSMLs with Code generation is GP!

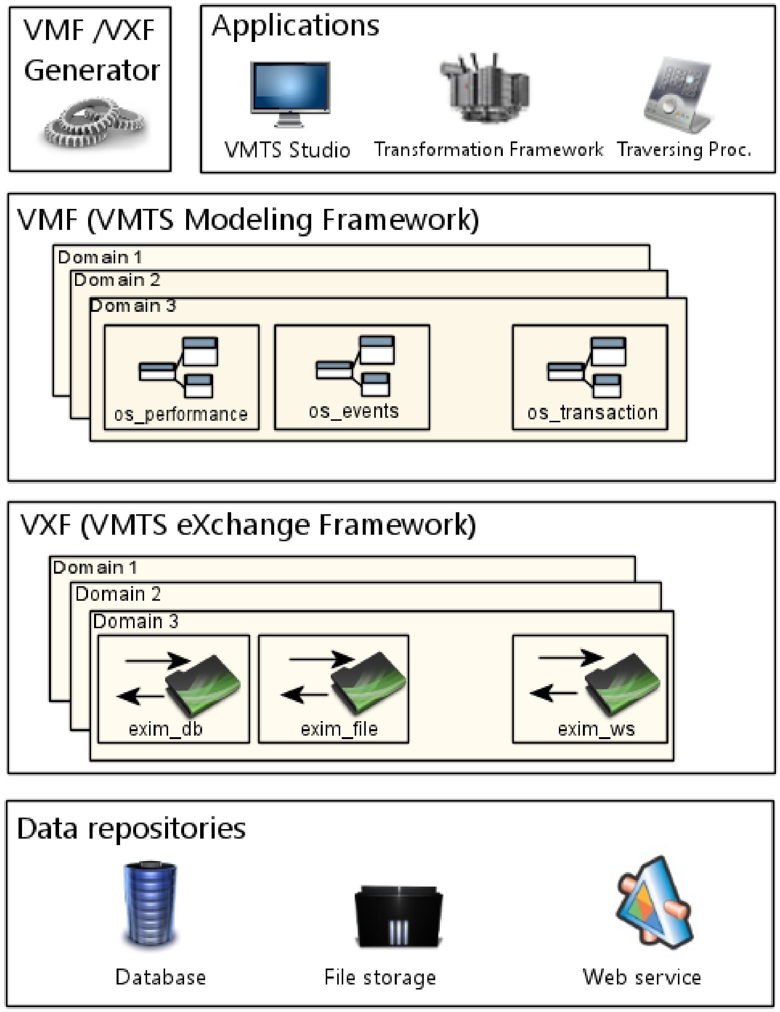
VMTS – Basics

The VMTS Framework

- **V**isual **M**odeling and **T**ransformation **S**ystem
  - Metamodeling and model transformation framework
  - Microsoft .NET-based
  - Metamodels, DSL models, transformations are edited in the same environment
  - Windows Presentation Foundation
  - N-layer metamodeling hierarchy
  - Constraint compiler
  - High-performance transformation engine
  - Animation framework

The VMTS Framework

- Architecture
  - Four layers for flexibility
  - Metamodel-based, auto generated components
  - Performance and customizability
  - Custom Exim for Matlab, and GXL



The VMTS Framework

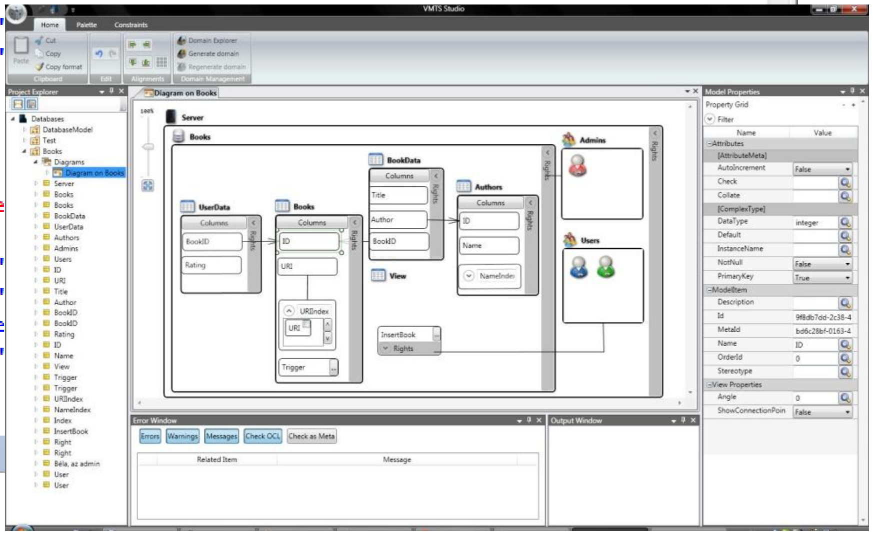
```xml
<!--RadialGrayGradient-->
<RadialGradientBrush x:Key="radialBrushGreyGardient"
      GradientOrigin="0.5,0.5"
      Center="0.5,0.5" RadiusX="0.5" RadiusY="0.5">
    <RadialGradientBrush.GradientStops>
        <GradientStop Color="Black" Offset="0" />
        <GradientStop Color="Black" Offset="0.5" />
        <GradientStop Color="Azure" Offset="0.75" />
        <GradientStop Color="Gray" Offset="1" />
    </RadialGradientBrush.GradientStops>
</RadialGradientBrush>

<!--Start-->
<ControlTemplate x:Key="StartTemplate" TargetType="{x:Type ContentControl}">
    <Grid>
        <Ellipse Height="36.652" Width="36.652"
                 VerticalAlignment="Top" Fill="
    </Grid>
</ControlTemplate>

<!--End-->
<ControlTemplate x:Key="EndTemplate" TargetType
    <Grid>
        <Ellipse Height="36.652" Width="36.652"
                 VerticalAlignment="Top" Fill="
        <Ellipse Height="16" Width="16" Name="e
                 VerticalAlignment="Top" Fill="
    </Grid>
</ControlTemplate>
```

Metamo

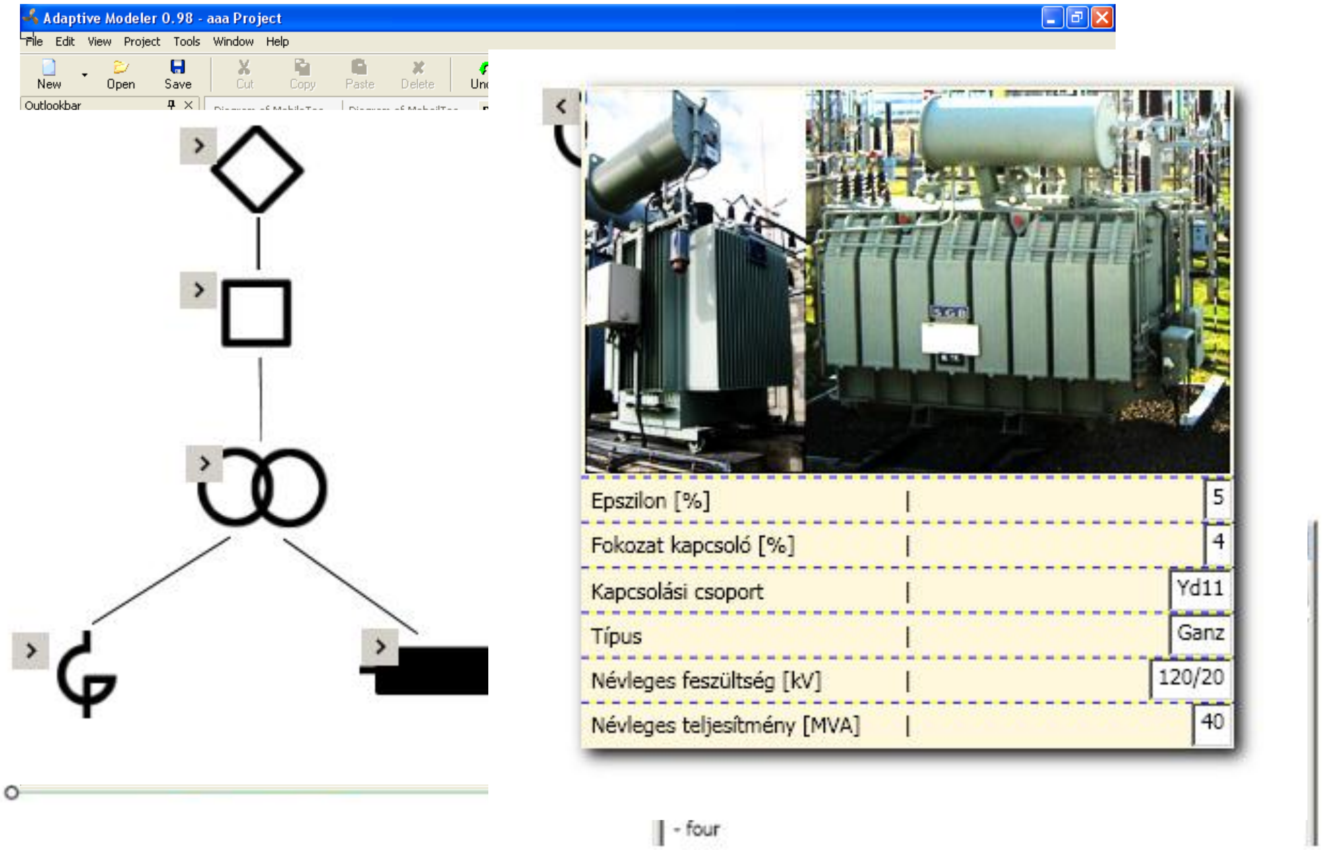
tes



Concrete syntax(XAML)

The VMTS Framework

- Supported domains



Constraint compiler

Constraint compiler

OCL to C#

- Primitive and compound literals

```
'string'; 1; 2.1; True
Sequence{1,2,3,1}; OrderedSet{1,2,3,1}
```

```
"string"; 1; 2.1; true
new List<int>(){1,2,3,1}; new List<int>(){1,2,3,1}.Distinct()
```

- Unary and binary expressions

```
not (1=1); -(5+6)
True  xor  False; (1=1) implies (True or False)
1+2.3; 10 div 4
```

```
!(1==1); -(5+6)
true^false; !(1==1) || (true || false)
1+2.3; 10.div(4), where div is the following extension method:

public static int div(this int self, int other)
{ int rem; return Math.DivRem(self, other, out rem); }
```

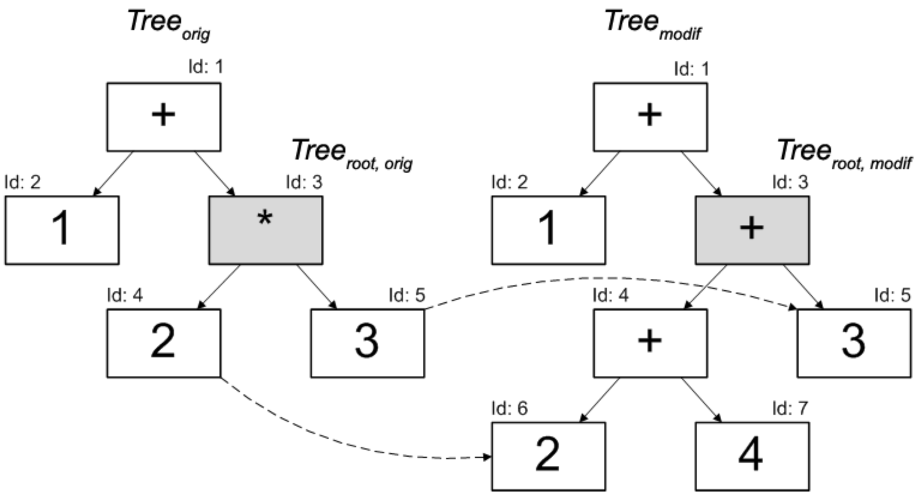
Incremental compilation

- Incremental compilation uses previously produced internal representation of the code (AST, AST-code map)

$$
\begin{tikzcd}
 & OCL \arrow[r, "parse"] \arrow[d] & AST \arrow[rrr, "semantic"', "analysis"] & & & AnalyzedAST \arrow[r, leftrightarrow, "code", "generation"'] \arrow[dll] & C\# \arrow[d] \\
ChangeLog \arrow[r] \arrow[rrr, bend right] \arrow[rrrrr, bend right] & OCL \arrow[r, "parse"] \arrow[d] & AST \arrow[r] & MergedAST \arrow[rr, "partial", "sem.anal."'] \arrow[rrr, bend right] & & AnalyzedAST \arrow[r, leftrightarrow] \arrow[dll] & C\# \arrow[d] \\
ChangeLog \arrow[r] \arrow[rrr, bend right] \arrow[rrrrr, bend right] & OCL \arrow[r, "parse"] & AST \arrow[r] & MergedAST \arrow[rr, "partial", "sem.anal."'] \arrow[rrr, bend right] & & AnalyzedAST \arrow[r, leftrightarrow] & C\#
\end{tikzcd}
$$

Incremental compilation

- Incremental semantic analysis
  - Locates modified vertices in AST
  - Locates unmodified subparts
  - Merges ASTs
- Incremental code generation
  - AST-code mapping



Animation Framework / Simulation

Specification of Visual Languages

- Metamodel
  - What are the elements of the language?
  - Which nodes can be connected by which connections?
  - „Abstract Syntax"
- Appearance model
  - How are the elements visualized?
  - „Dressing up" the elements
  - „Concrete Syntax"

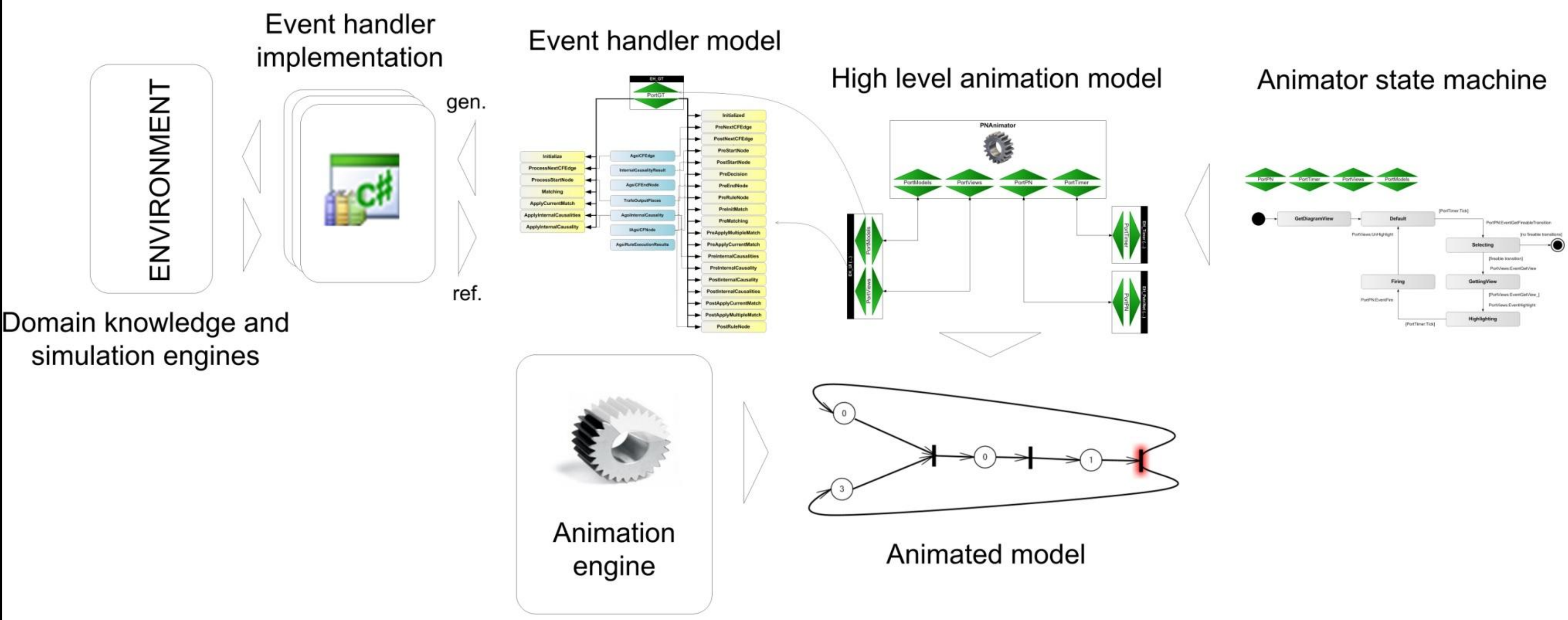
What about the dynamic behavior?

„Animation"

Animation

- Modeling
  - Part of the concrete syntax? Not general enough!
  - Separate model attached to the other two
- Domain-specificity
  - Typically complex dynamic behavior comes from an external system („You don't want to write a MATLAB if you have one")
- We assume this system a „black box"
  - Loosely coupled: event handling
  - VMTS Animation Framework (VAF)

VAF Architecture



Separating animation and **domain-specific knowledge** with **event-based integration**.

VMTS Animation Framework (VAF)

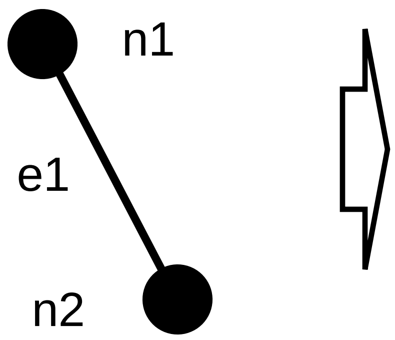
- Event handler model
  - Models the events and the entities
  - Event handlers connect the simulation engines, 3rd party components, and the VMTS UI
- Event driven state machines to describe animation
  - Compose simple events or decompose complex events
- High-level animation model
  - Integration of event handlers and state machines
  - Components passing events through *ports* („fixed length buffers”)

Optimized model transformations

Optimized model transformations

A naive compiled matcher

- Pattern graph => matcher algorithm
- Nested cycles:




```
forach (Node n1 in nodes)
  if (...) //condition examination
  foreach (Node n2 in nodes)
  if (...)
    foreach (Edge e1 in edges)
    if (...) {
          ... //rewriting
    }
```

- Can be highly optimized
  - Matching order
  - Navigation

„The Idea"

- Most graph-rewriting engines optimize rule executions separately
  - Starts the matching from scratch every time
  - Parallel execution
- What about exploiting similarity of patterns?
  - Incremental pattern matching
  - ***Overlapped Rewriting Algorithm (OLRA)***
    - Overlap the matching phase of isomorph parts of similar rules and perform the matching only once

Overlapped matching-problems

- Sequential execution
  - Influencing the execution of the following rules
    - Enabling/disabling matches for them
  - Influencing the final result (attribute conditions)
- Reordering the matching of the rules
  - Matching at once, without execution
- Application conditions : *OLRA susceptibility*
  - The overlapped rules should be sequentially independent for each match
    - Including the attribute conditions
  - The attribute transformations of the rules should be commutative
  - Not so rare as it sounds to be

Property analysis / transformation patterns

Property analysis

- Property analysis of model transformations: formally proving
  - some properties of the transformations (e.g. termination),
  - the mapping between the input and output models
  - Properties of the models when the transformation finishes
- Offline analysis: do not take concrete input models into account, only the definition of the transformation itself is used for analysis
  - advantages: performed only once, results hold for every model
  - Disadvantage: more difficult

State-of-the-art

- General offline analysis methods cannot be provided
  - e.g. termination of a transformation is undecidable in general
- Current approaches for offline analysis propose methods that
  - can be applied for a concrete (type of) transformation,
  - or can be used to analyze a concrete type of property

Our research – goals

- The future goal is to provide fully automated methods for the analysis, this cannot be reached at once
- Our current goal is to automate more and more elements of the analysis process and to combine manual and automated methods

Our research – MTA patterns

- Model Transformation Analysis (MTA) patterns are design patterns for implementing transformations
  - An MTA pattern is well-defined sub transformation pattern that can be reused when implementing a model transformation
  - The motivation (when to apply) and the structure (how to implement) a pattern is documented
  - An MTA pattern (since it is sub transformation) can be pre-analyzed, the result of the analysis will hold for the relevant part of a concrete transformation where the pattern is applied

Our research – MTA patterns

- Concrete MTA patterns have been defined for traversing hierarchical models

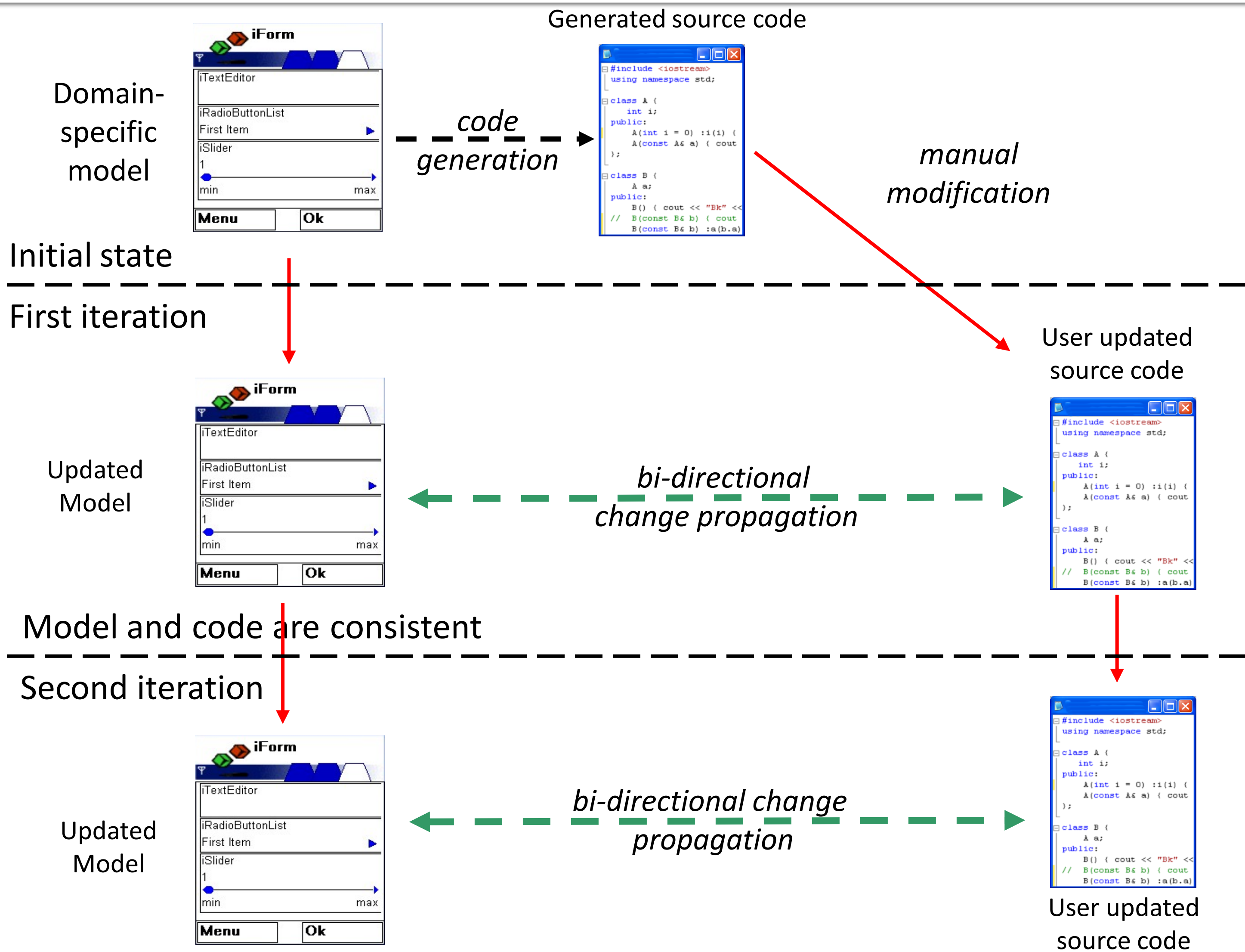
Our research – automated reasoning

- We have introduced the term *assertion*.
  - Assertions are automatically derived from the definitions of model transformations or can be manually provided by model transformation experts.
  - Assertions describe the main characteristics of different parts of the transformations and contain the pieces of information that are relevant for further analysis.
  - An appropriate automated reasoning system can derive the proof of certain properties based on the initial assertions.
- We have proposed a method to automatically generate certain type of assertions and provide the deduction rules for a reasoning system to prove some properties of transformations.

Round-trip engineering

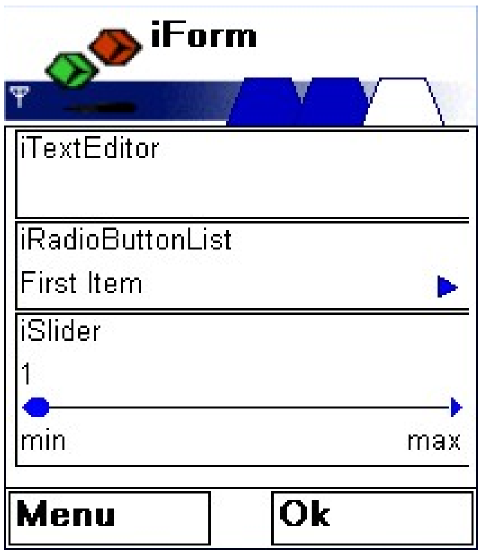
Round-trip engineering

Iterative Model-Based Development

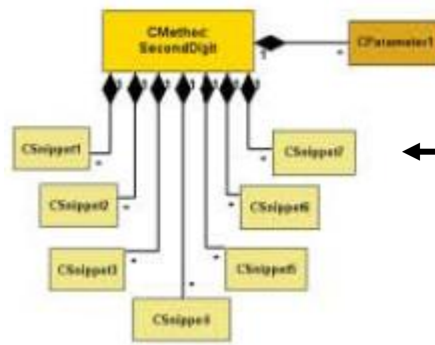


Model-Code Round-Trip Concept

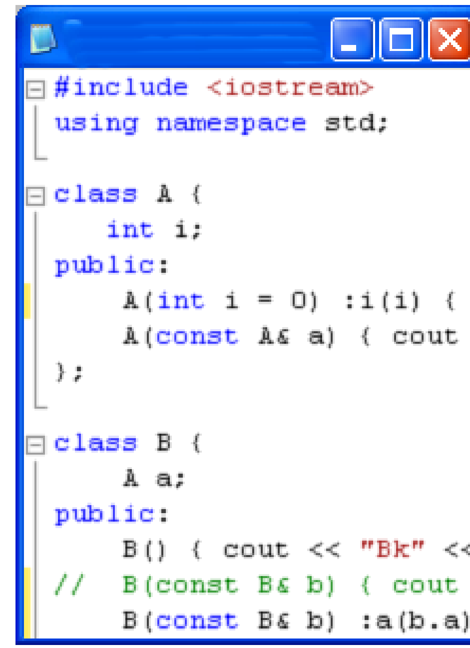
Domain-specific model
(platform independent)



Platform-specific AST
(CodeDOM) model



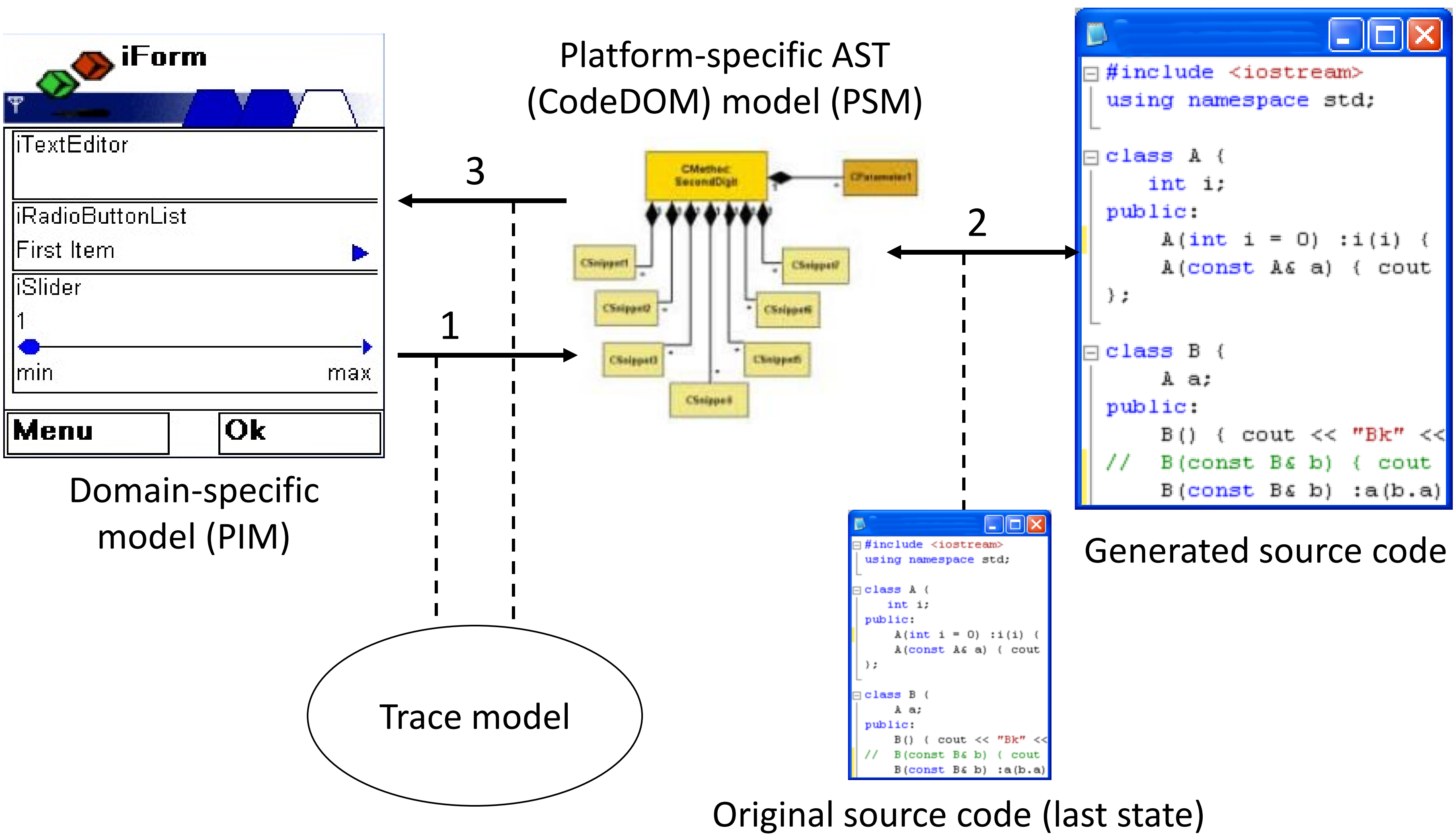
Generated source code



Round-trip 1

Round-trip 2

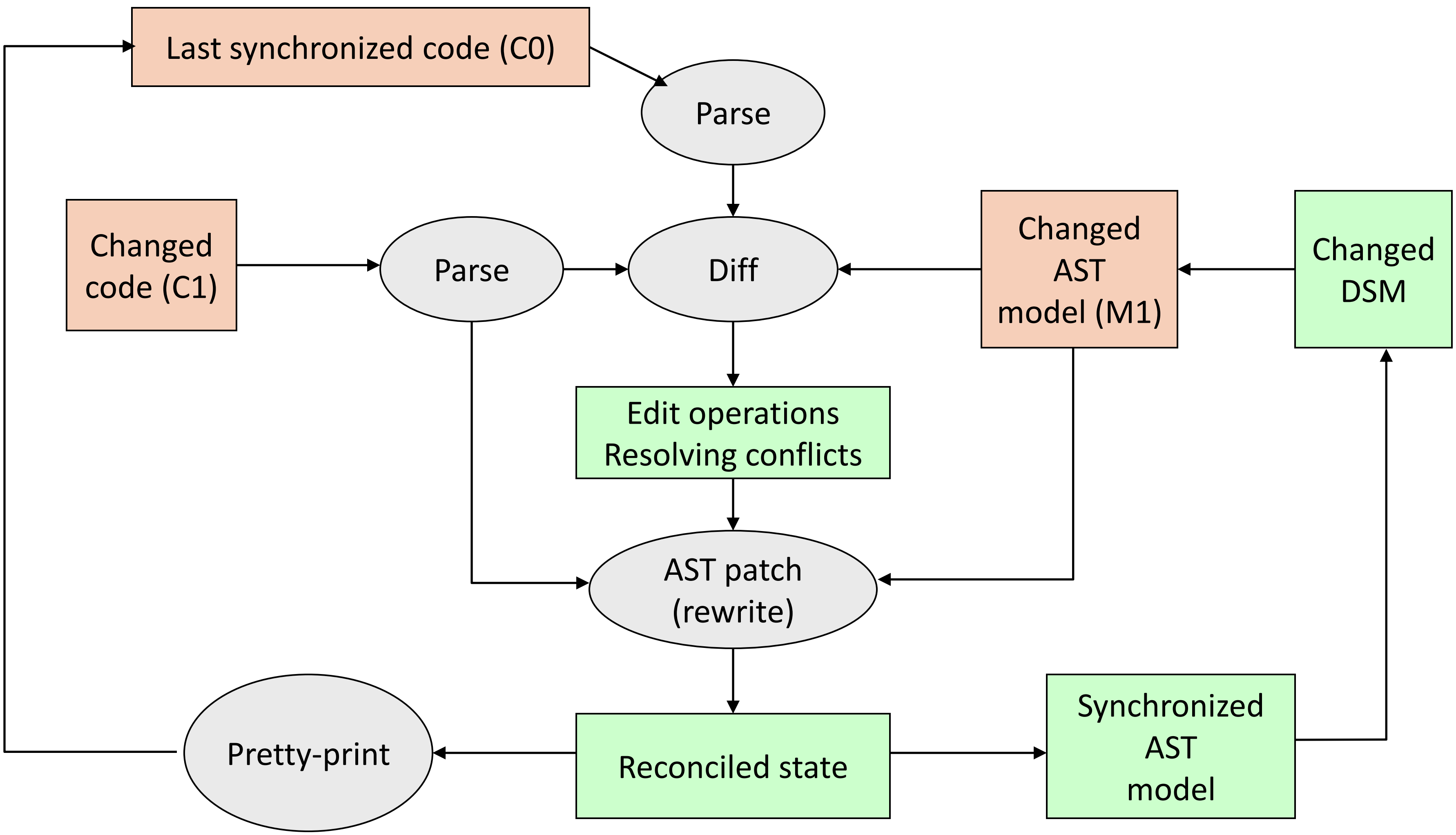
Model-Code Round-Trip Concept



Background of Synchronization

- Incremental synchronization → merging the changes
- Detect changes: differencing
  - Textual (diff tool, general text file)
  - Abstract Syntax Tree (AST) differencing (language dependent)
  - Edit script (the output of differencing, sequence of atomic edit operations: (INS, UPD, DEL, MOV)
- Change propagation: manually or tool-aided
- Modeling the source code with an AST model (that has a corresponding AST metamodel)
  - to describe the platform-specific implementation
  - AST model is comparable to the parsed source code
  - Syntactic elements of the language as atomic modeling elements

VMTS Round-Trip Concept



Conclusions: Pros and Cons

+

- Statement-level incremental synchronization
- Syntactical correctness is ensured
- Free moving between different representations of the system (code and model)
- Enables iterative and incremental development

-

- Low-level synchronization technique, the high-level intentions of the developer should be found out
- Complicated transformation rules (DSL - AST)
- Not trivial, how to handle the semantic conflicts without user intervention
- Preserving comments and formatting info (white spaces) depends on the parser and the pretty-printer

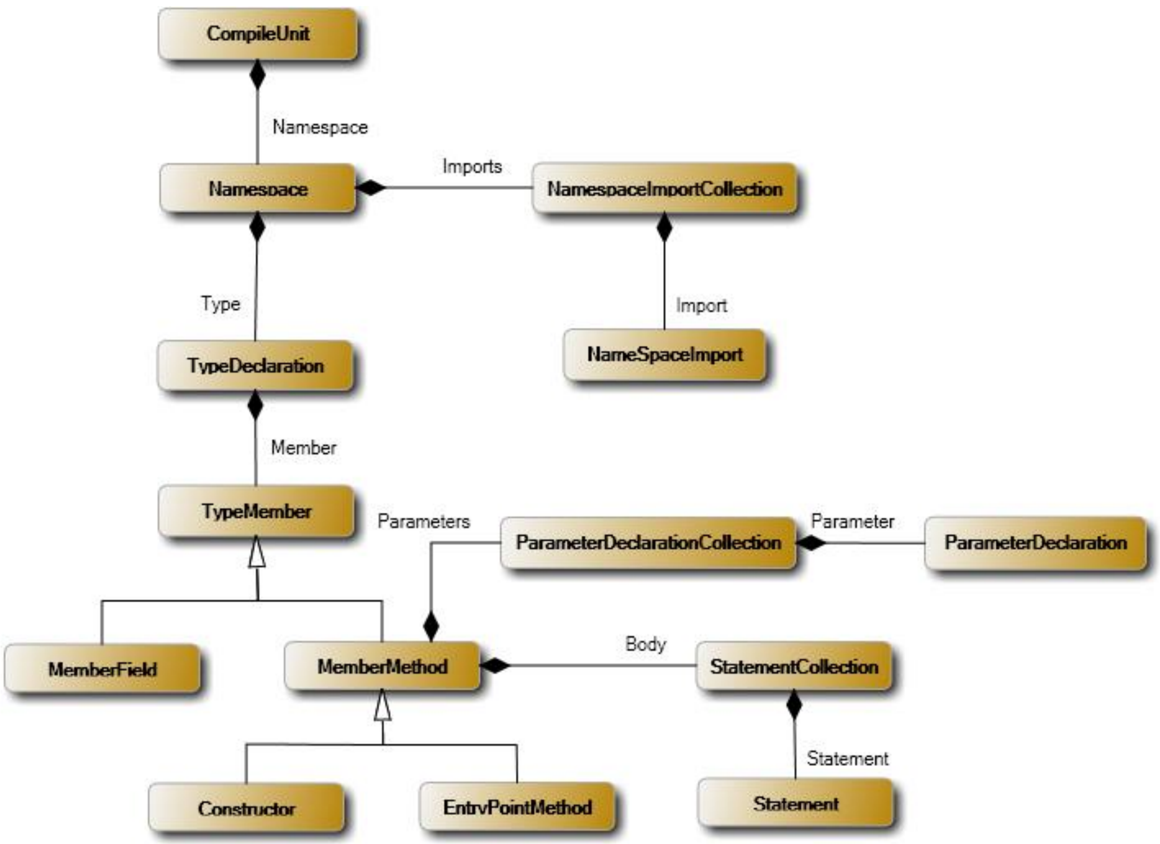
Generalization of the Approach

- Eliminate hand written language specific code
  - parser + glue code (3rd party parser)
  - pretty-printer (Microsoft's CodeDOM)
  - edit script executer (model and CodeDOM tree patching)

- General, language independent algorithms
  - tree differencing
  - conflict resolution

- Solution: modeling the specific objects (AST metamodel)
  - define elements of the AST model
  - specify the textual syntax of each model elements
  - generate the specific code from these models
  - pretty-printer and parser can be constructed

CodeDOM-like Metamodel



Namespace: $Imports "\nnamespace" #Name "{\n" $Type "}\n"
TypeDeclaration: "class" #Name ($Base ? ":" $Base) "{\n" $Member "}\n"
EntryPointMethod: "static void Main(string[] args) {\n" $Body "}\n"

Model synchronization

Model synchronization

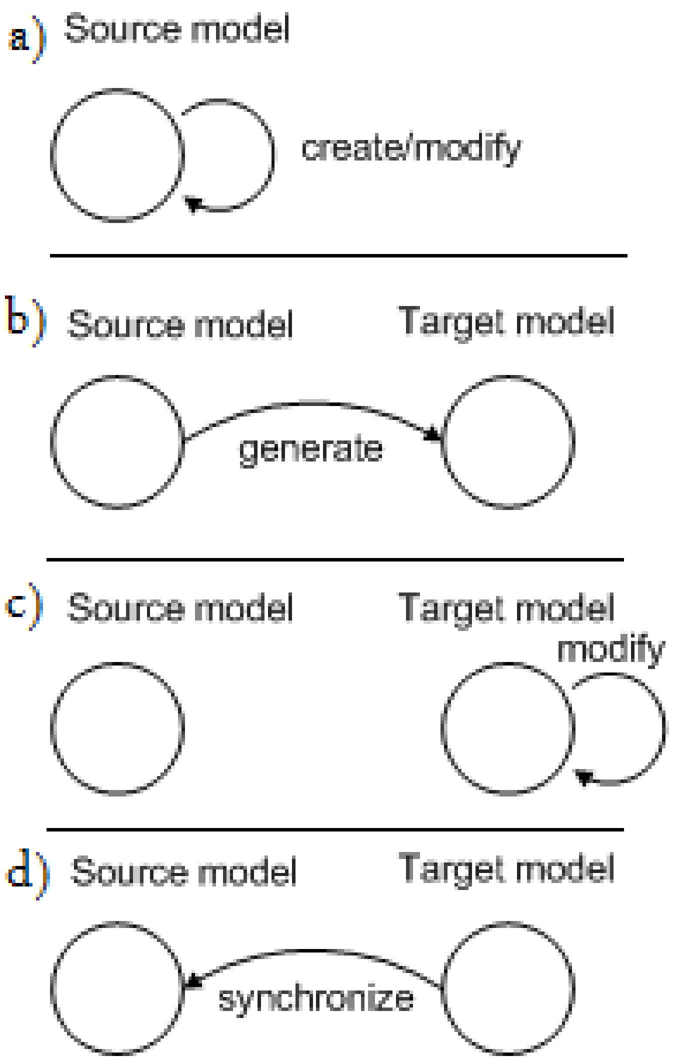
Model-based software engineering

- Developers are working on several models simultaneously
  - E.g., developing mobile applications
    - User interface model (without behavior)
    - Application behavior model (source code)
  - The two models describe different aspects of the same system
  - Entire system is realized by combining these aspects
  - Generation process by model transformations

Motivation

- The developer often wants to change the target artifact
- The target and the source artifact will not be necessarily consistent, synchronization is needed
- The modifications have to be propagated back to the source artifact

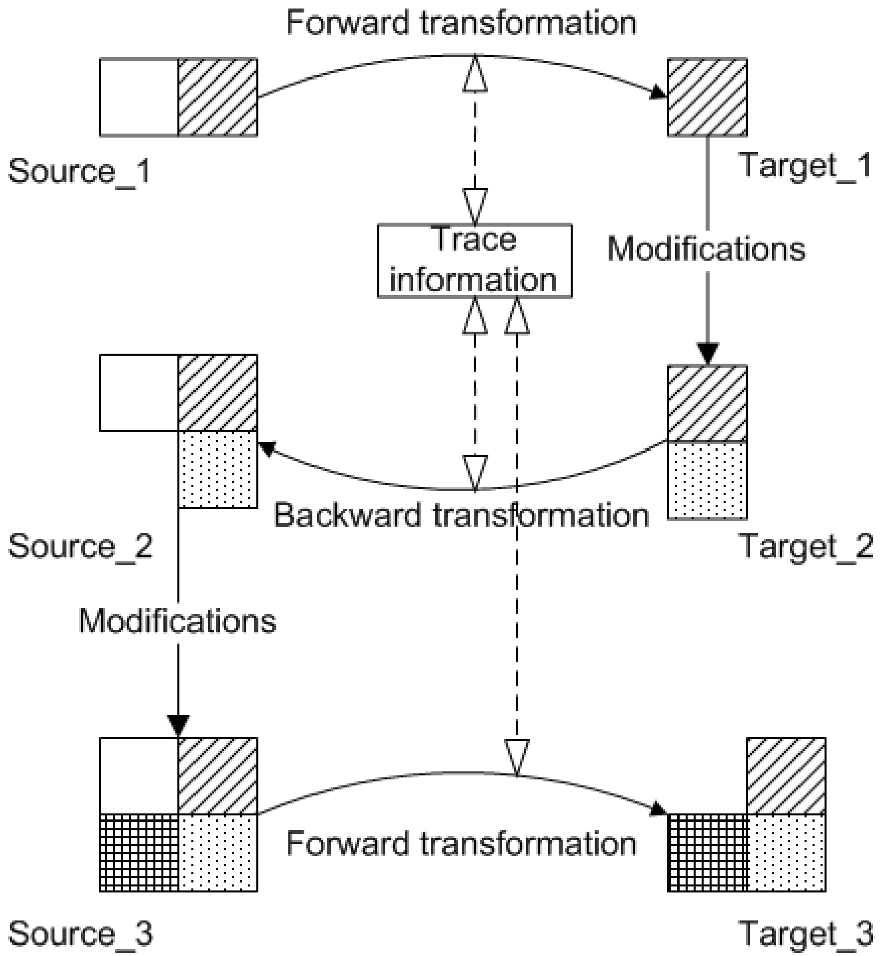
Procedure of the development



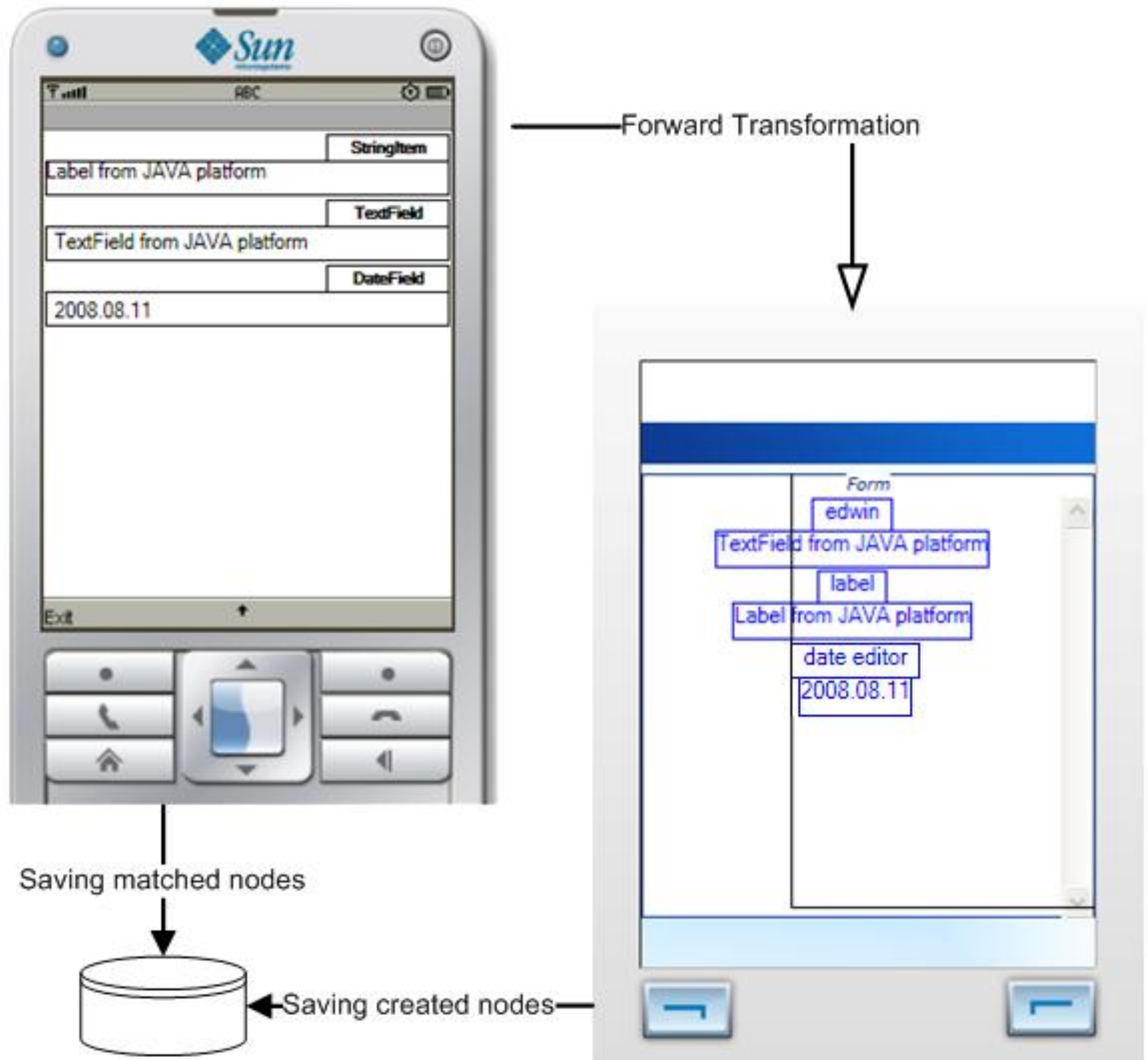
Incremental model synchronization

- The synchronization is implemented as two unidirectional transformations
- Transformation saves trace information during the execution
- The reverse direction uses trace information

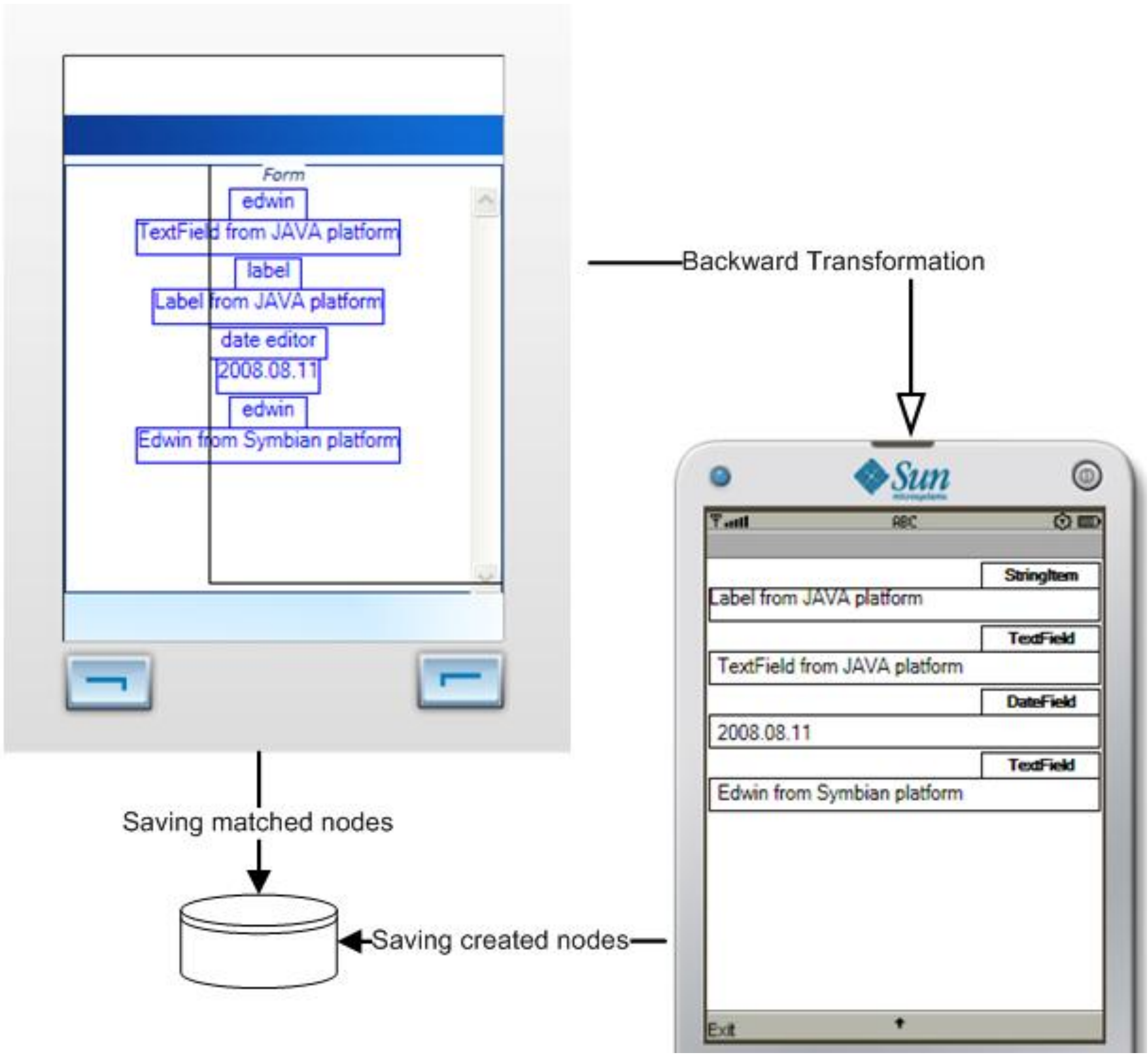
Model synchronization



Case study – Mobile UI synchronization



Case study – Mobile UI synchronization



Design patterns in DSMLs

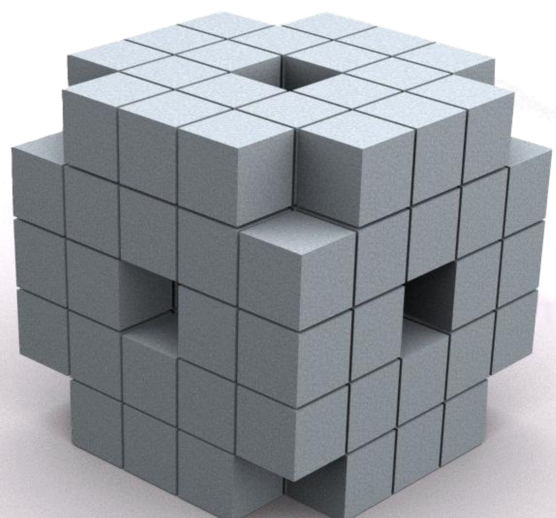
Domain-Specific Model Patterns

Domain-specific model patterns

- Design patterns for DSLs
  - The knowledge of domain experts
  - Solution to well-known domain problems
- Relaxing the instantiation: partial model
  - Incomplete attributes
  - Relaxed multiplicities/cardinalities
  - Transitive containment
  - Constraint profiles
- Relaxing the metamodel

Thank you for your attention!

Thank you for your attention!



http://vmts.aut.bme.hu