Challenges in bootstrapping a model-driven way of software development

–Dennis Wagelaar
–System and Software Engineering Lab

Vrije Universiteit Brussel

Outline

- Context: MDSD
- Case study
- Challenges:
  - Bootstrapping model transformations and language abstractions
  - Evolving a step-wise refinement chain
  - To round-trip or not to round-trip

© 2008, Dennis Wagelaar

Context: MDSD (1)

- MDE is usually demonstrated...
  - ...when it is already in place
  - ...as part of a ready-to-run solution
- "Ready" MDE solutions generally don't do **exactly** what you need, which means:
  - You need to do "post-customisation" on the tool's output,
  - Which can be done by writing your own model transformation definitions

© 2008, Dennis Wagelaar

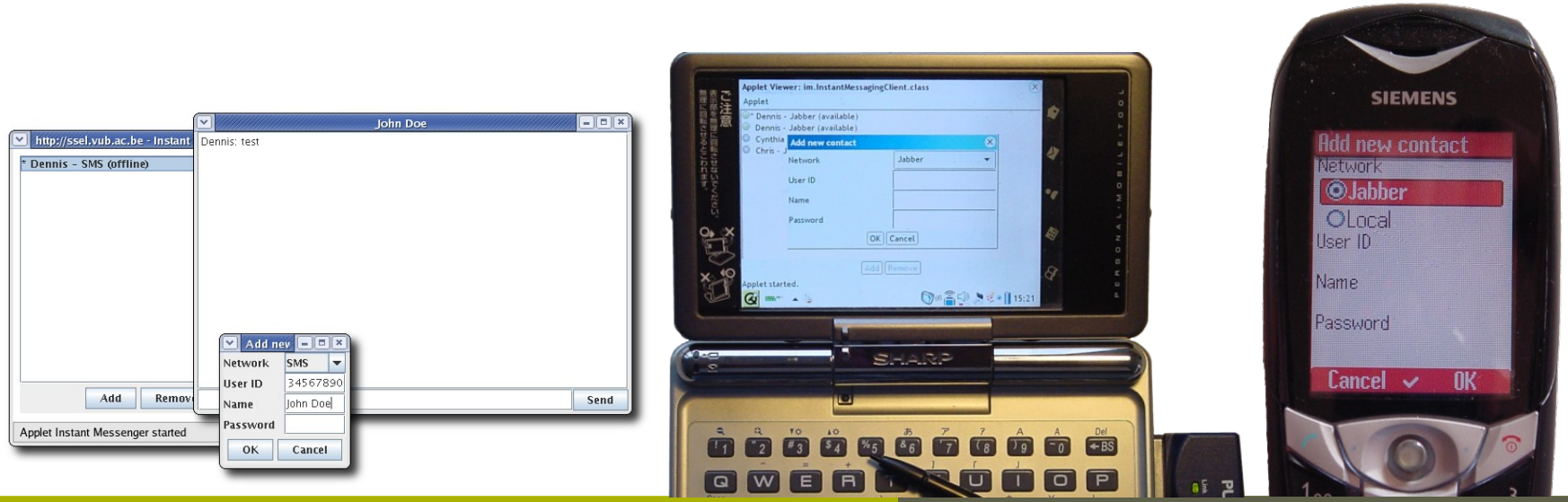
Context: MDSD (2)

- Regardless of whether you use “ready-to-run” tools or a customised MDSD setup:
  - Sooner or later you'll have to develop/maintain your own model transformation definitions
- How Do You Get There?
  - How to bootstrap model transformations and language abstractions?
  - How to evolve a step-wise refinement chain?
  - To round-trip or not to round-trip?

© 2008, Dennis Wagelaar

Case study

- Instant messaging client
  - One core PIM and 7 optional feature PIMs, all in UML 2.x
  - 11 PIM-to-PSM refinement transformations in ATL
  - Targets all Java client platforms



© 2008, Dennis Wagelaar

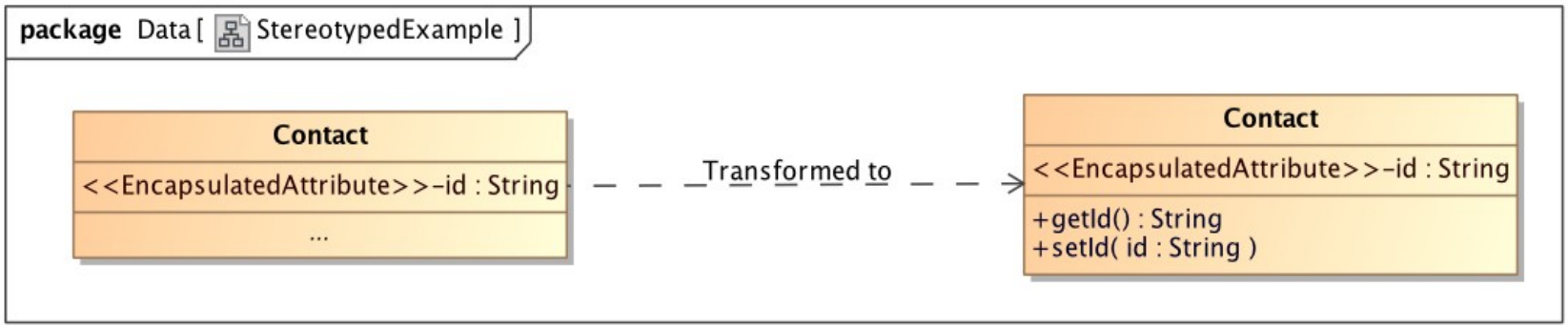
Bootstrapping model transf's and language abstractions

- The instant messenger started out with a UML model and a simple code generator
  - Several recurring patterns in the model:
    - Getter and setter methods
    - Design pattern implementations (observer, abstract factory, ...)
  - Platform-specific API references in UML model:
    - Applet, collection types, AWT, ...
- Use model transformation to automatically generate recurring patterns and platform-specific references
  - Replace recurring patterns by special-purpose language abstractions

© 2008, Dennis Wagelaar

Language abstractions

- UML provides the Profile mechanism to introduce new language abstractions:
  - Define <<EncapsulatedAttribute>> stereotype on top of UML Property
  - Each <<EncapsulatedAttribute>> will be transformed to a private attribute with public getter and setter methods



© 2008, Dennis Wagelaar

Stereotype transformation

Matching stereotyped Property instances

```
module Accessors;
create OUT : UML2 from IN : UML2;
rule PublicPropertySingle {
  from s : UML2!"uml::Property" (
      UML2!"Accessors::EncapsulatedAttribute".allInstances()
      ->select(e|e.base_Property=s)->notEmpty())
  using { baseNameS : String = s.accessorBaseNameS; }
  to t : UML2!"uml::Property" (...),
    getOp : UML2!"uml::Operation" (name <- 'get'+baseNameS,
                                   class <- s.class,
                                   ownedParameter <- Sequence{getPar}),
    getPar : UML2!"uml::Parameter" (name <- 'return',
                                    type <- s.type,
                                    direction <- #return),
    getDep : UML2!"uml::Dependency" (name <- 'Get'+baseNameS,
                                     client <- getOp,
                                     supplier <- s),
    getDepST : UML2!"Accessors::accessor" (base_Dependency <- getDep), ...
}
```


© 2008, Dennis Wagelaar

Stereotype transformation

```
module Accessors;
create OUT : UML2 from IN : UML2;
rule PublicPropertySingle {
  from s : UML2!"uml::Property" (
      UML2!"Accessors::EncapsulatedAttribute".allInstances()
      ->select(e|e.base_Property=s)->notEmpty())
  using { baseNameS : String = s.
  to t : UML2!"uml::Property" (..
    getOp : UML2!"uml::Operation"

    getPar : UML2!"uml::Parameter"
                                    type
                                    direction <- #return),
    getDep : UML2!"uml::Dependency" (name <- 'Get'+baseNameS,
                                     client <- getOp,
                                     supplier <- s),
    getDepST : UML2!"Accessors::accessor" (base_Dependency <- getDep), ...
}
```

Instantiating “getDep” as a Dependency with a stereotype applied to it

© 2008, Dennis Wagelaar

Metaclass transformation

```
module Accessors2;
create OUT : UML2 from IN : UML2;
rule PublicPropertySingle {
  from s : UML2!"accessors::EncapsulatedProperty"
  using { baseNameS : String = s.accessorBaseNameS; }
  to t : UML2!"accessors::EncapsulatedProperty" (...),
    getOp : UML2!"uml::Operation" (name <- 'get'+baseNameS,
                                   class <- s.class,
                                   ownedParameter <- Sequence{getPar}),
    getPar : UML2!"uml::Parameter" (name <- 'return',
                                    type <- s.type,
                                    direction <- #return),
    getDep : UML2!"accessors::AccessorDependency" (name <- 'Get'+baseNameS
                                                   client <- getOp,
                                                   supplier <- s), ...
}
```

Matching EncapsulatedProperty instances

Instantiating “getDep” as an AccessorDependency

© 2008, Dennis Wagelaar

Profiles vs. metamodels

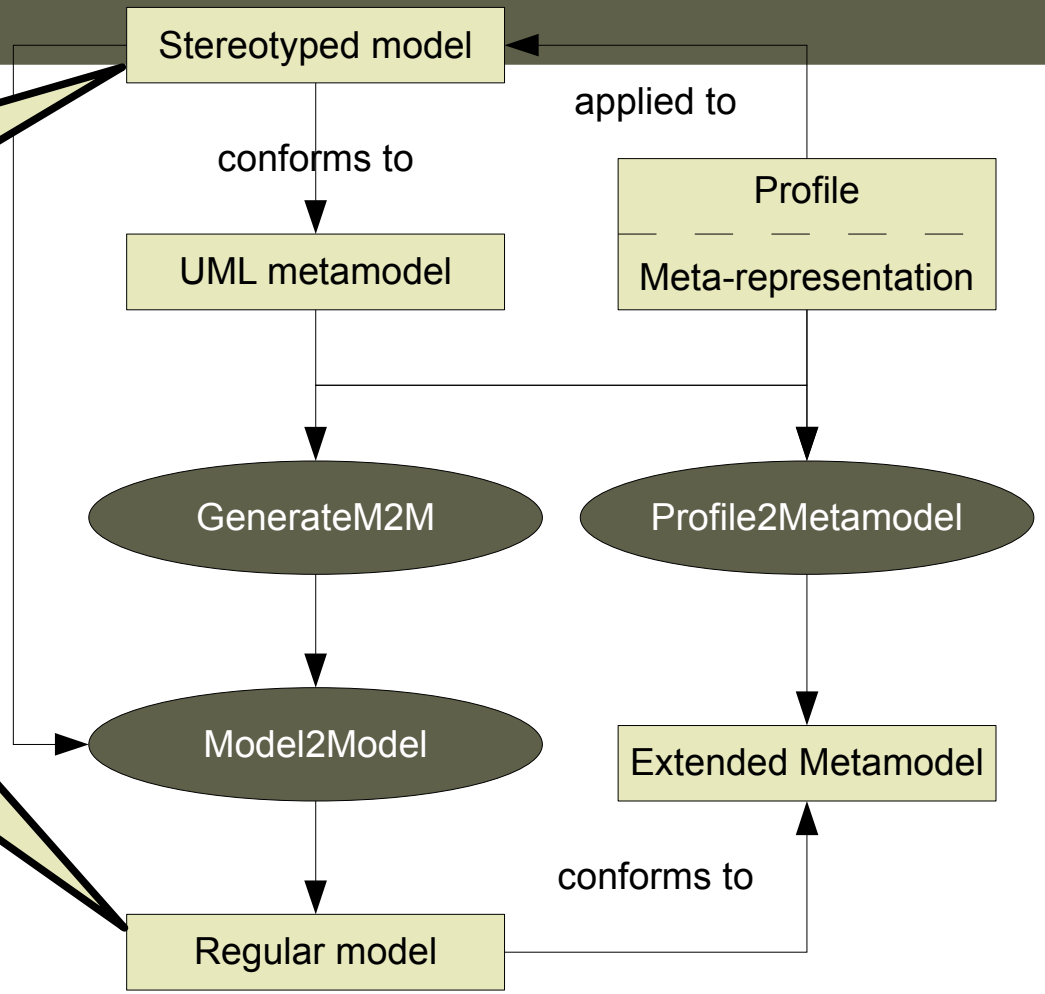
- Profiles allow for easier language extension than meta-models
  - No need to worry about concrete syntax, versioning
- Profiles make model transformation definitions more complex
  - Explicit stereotype instances require more navigation/instantiation
- UML Profile paradox:
  - Easy language extension causes complex model transformation definitions

© 2008, Dennis Wagelaar

Solution?

Translate a stereotyped model...

...into a regular model...



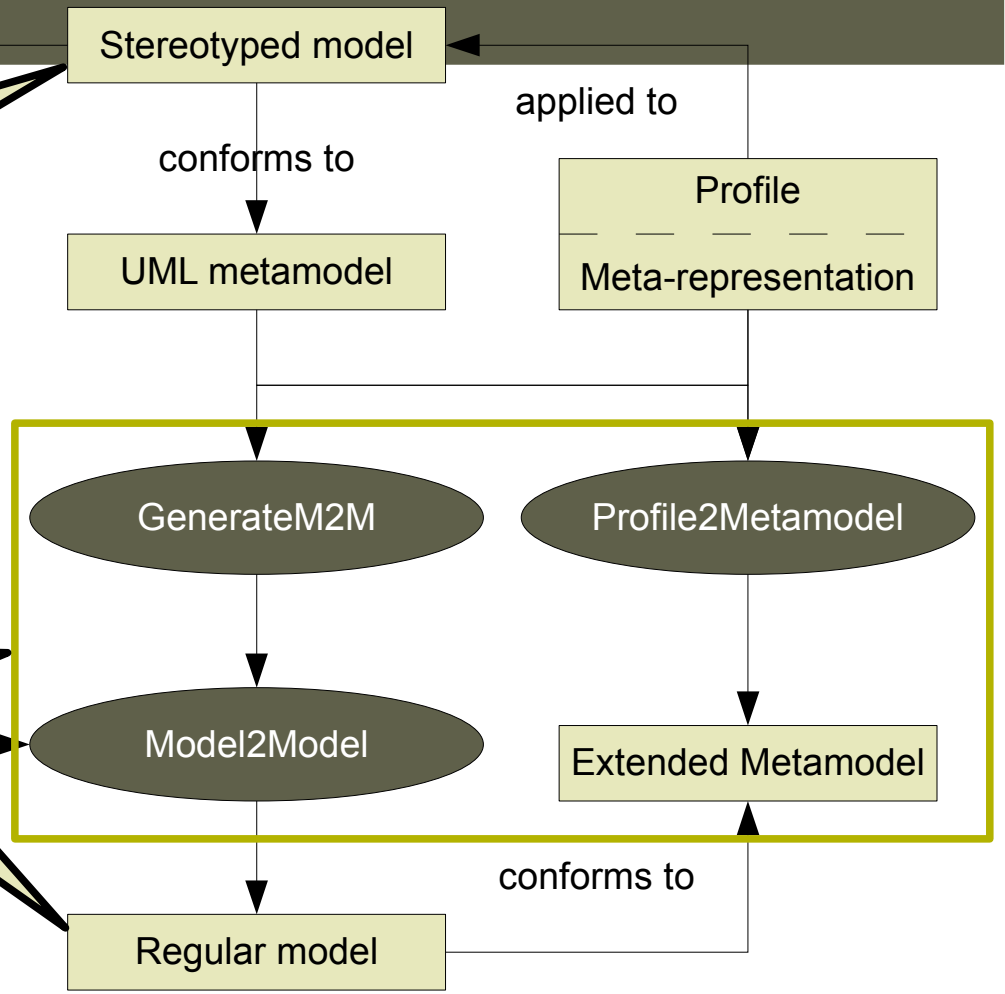
© 2008, Dennis Wagelaar

Solution?

Translate a stereotyped model...

...into a regular model...

...using a generic infrastructure



© 2008, Dennis Wagelaar

Evolving a step-wise refinement chain

- When defining additional refinement transformations on your model, dependencies are introduced
  - Example: Observer transformation depends on result of getter/setter transformation
- Critical pair analysis can help detect dependencies
  - But a detected conflict does not always mean “dependency”
  - And critical pair analysis is a complex computing job

© 2008, Dennis Wagelaar

Evolving a step-wise refinement chain

- Rich meta-classes can make dependencies explicit in the model:
  - Observer transformation requires “Setter” instances
  - Accessors transformation provides “Getter” and “Setter” instances
- By converging complex dependencies into simple, but semantically rich, metaclasses, automated analysis of dependencies becomes much easier

© 2008, Dennis Wagelaar

To round-trip or not to round-trip

- Often, some parts of the software are better edited in the model, others are better edited in the code
  - IDEs for code often have advanced verification/refactoring support, that you'll want to leverage (e.g. Eclipse)
  - Modelling language may not be efficient for expressing (all kinds of) behaviour
- Merging-style incremental code generators seem to provide a solution
  - Manual code changes are not overwritten by the generator
  - But changes to the code are also not propagated back to the model, when applicable

© 2008, Dennis Wagelaar

To round-trip or not to round-trip

- Round-trip engineering (RTE) aims to solve this problem
  - Model(s) and code are kept fully synchronised
  - But RTE is very hard to generalise for any language
  - And RTE is more than just bi-directional transformation
    - Bi-directional transformation definitions are harder to write than uni-directional transformations and are less expressive
- Recent work on RTE
  - Use only forward transformation definition and target model change recordings to do RTE
  - Again: not proven to work in general

© 2008, Dennis Wagelaar

Discussion

Questions?

© 2008, Dennis Wagelaar