The VAMPIRE Approach To Model-Driven Development

Hans Jonkers
Healthcare Information Technology
December 12, 2012
Background

From **model-based** specification to **model-driven** development:

- **1980s:**
  - COLD specification language (*Common O-O Language for Design*)
  - Originally meant for specifying CHILL programs
- **1990s:**
  - SPRINT method (*Specification, Prototyping & Reuse INTegration*)
    - COLD used for axiomatic specifications of data types
    - ProtoCOLD ⊆ COLD: used for executable specifications
    - CoCoNut: control component, acting as a light-weight RTK
  - Applied in development of GFL family of television sets
- **2000-2005:**
  - ISpec interface specification approach
  - Applied in specification of UHAPI
  - Used in CTT courses
- **> 2003:** model-driven development → VAMPIRE
Purpose

Provide a minimalist tool set for model-driven development (MDD):
• Target platform: .NET + C#
• Modeling formalism: MOM (Meta Object Model)
• Model editor: VIDE, integrated graphical modeling environment
• Code generators: heavy-weight, light-weight
• Auxiliary tools: XSD to MOM transformer, documentation generator, etc.

Related approaches:
• Model-Driven Architecture (MDA), OMG
• Eclipse Modeling Framework (EMF), open source
• Domain Specific Languages (DSL), Microsoft
History & Applications

- 2004: first version, applied to ISpec, UHAPI
- 2005: applied to MIP Information Models
- 2006: development of MADE (protocol editor)
- 2007: MADE transferred to Medical Systems, renamed to VAB
- > 2007: used in internal projects, adjustments and extensions made on demand (example models: AsbruModel, MotivaCarePlan, ClinicalStatement)
- > 2009
  - used for implementing standards using ‘reverse model engineering’ (HL7v3, DICOM SR, DICOM IODs, UCUM, PII System Authority)
  - development and implementation of a reference model for computer-interpretable guidelines XGM (eXtensible Guideline Model)

Contributions by (alphabetically):
- Paul ten Brink, Hans Jonkers, Marc Stroucken, Richard Vdovjak
MDD = Data-Driven Development

Traditional application (hand-written code):

Example: guideline completely implemented in code

Separating the knowledge from the code:

Example: guideline represented as network of nodes + engine

Interpretive approach:

run-time data-driven application

Generative approach:

compile-time data-driven application
MDD = Domain-Specific Modeling

Domain-Specific Modeling:
• Creating *abstractions* of the relevant concepts in an application domain
• Representing these abstractions in terms of *concrete objects* (= data)

In other words:

A *model* is a collection of mutually related *concrete abstractions*

Abstractions may be defined at several *metalevels*:

Cf. C#:    `new DateTime()`    `typeof(System.DateTime)`    `typeof(System.Type)`

**Essential**: types and metatypes are also objects!
Casting Models in a Formal Language: MOM

Modeling language:

Formal language to define models (objects, types, etc.)

MOM (Meta Object Model) modeling language design principles:

• Use *object-oriented* concepts.
• Keep the language *as simple as possible*.

MOM = “Object-oriented BNF”:

• \( A ::= “a” \) \( \rightarrow \) value type
• \( A ::= B \ C \) \( \rightarrow \) class type
• \( A ::= B \mid C \) \( \rightarrow \) union type (‘choice type’)
• \( A ::= B \ast \) \( \rightarrow \) list type
• \( A \leftarrow B \) \( \rightarrow \) inheritance
• \( A ::= B \uparrow C \) \( \rightarrow \) references

→ objects are *graphs* rather than *trees*.
MOM: Core Model

Type hierarchy:

- **Object**
  - **ObjectType**
  - **Property**
  - **ClassType**
  - **ValueType**
  - **UnionType**
  - **ListType**

Type properties:

- **ObjectType**
  - **BaseType**
  - **Properties**
- **Property**
  - **ContentsType**
  - **ElementType**
  - **UnionType**
    - **MemberTypes**

Note:
- Properties can be viewed as *partial functions* mapping objects to objects.
- Each object type is itself an object, i.e. an instance of **ClassType**.
MOM: Graphical Notation

ModelElement
  └── Name: String
  └── Namespace: String
      └── Description: String

Node
  └── Step
      └── DeferrableEvents []
          └── Event
              └── Event

RegularStep
  ├── Variables []
  │    └── Variable
  │    └── Action
  └── Entry []
      └── Action
  └── Exit []
      └── Action
  └── DoActivity []
      └── Action

SimpleStep

CompositeStep

supertypes

type being defined

subtypes
Union Types: Why?

Frequently occurring problem in single-inheritance modeling languages:

Type B-or-D cannot be defined like this:

Common solution:

+ constraint: \( \forall x \in E ( x.B = \text{null} \lor x.D = \text{null} ) \)
Union Types versus Supertypes

Union type $T$

- $x \in T \Rightarrow \exists i \ ( x \in S_i )$
- $T$ is defined after $S_i$
- $T$ imposes constraints on $S_i$
- Basetype of $T$ is derived from its member types $S_i$

Supertype $T$

- $\exists i \ ( x \in S_i ) \Rightarrow x \in T$
- $T$ is defined before $S_i$
- $S_i$ imposes constraints on $T$
- Basetype of $T$ is defined by $T$
Union Types: Example

```
ModelElement
  └── Name : String
  └── Namespace : String
    └── Description : String

Expression
  └── InvocationExpression
    └── MethodCall
        └── Object
            └── Expression
        └── Method
            └── Arguments : [Expression]

Action
  └── ModelElement
    └── Name : String
    └── Namespace : String
        └── Description : String
            └── Assignment
            └── MethodCall
            └── EventSend
            └── Conditional
```
Constraints

- There is *no general-purpose constraint language* (like OCL) in MOM
- There are *no built-in constraints*: all properties of object types are optional by definition

Pragmatic approach:
- If constraints are one-off, define constraints in *code* (C#)
- If constraints have to be analyzed, processed, interpreted, etc. define a *domain-specific constraint language* as part of the metamodel

Example
- Attribute conditions in DICOM IODs
Textual versus Graph Representation of Objects

- Visual tools (like VIDE) edit objects in terms of their (in-memory) graph representation.
- VAMPIRE code generators support automatic serialization and deserialization of objects to and from XML.
- Sometimes human-readable textual input/output of objects is desirable (e.g. for entering formulae).

**Question**: how to automatically convert between the textual and graph representations of objects?

- Approach developed by Alex ten Brink, consisting of
  - A *single language* for specifying how objects from a MOM object model are mapped to text and vice versa.
  - A generator that creates a *parser* and *unparser* from a specification in that language.
The Model-Driven Handwork in VAMPIRE

• Working with models:
  – Authoring Models: VIDE
  – Model to Code Transformations
  – Model to Model Transformations
  – Data to Model Transformations
Authoring Models: VIDE
Model to Code Transformations

Model to C# code generators:

- **Heavy-weight (HW) C# code generator:**
  - interface-based: maps model types 1-1 to C# interface types
  - uses hashtable-based implementation of properties
  - supports all MOM metalevels
  - XML serialization support

- **Light-weight (LW) C# code generator:**
  - class-based: maps model types 1-1 to C# class types
  - maps properties in model 1-1 to C# properties
  - no support for MOM metalevels
  - XML serialization support (compatible with HW code generator)

- **Custom model to C# code generators:**
  - written directly in C# using (HW or LW) code generated from models
Model to Model Transformations

- Model to model transformations: mappings from models to models within the same modeling framework
- Written in C# using code generated from models
- Transformations can cover all metalevels:
  - objects to objects
  - objects to types (“lifting”)
  - types to types
  - types to metatypes, etc.
- **Much** easier than using XML and writing transformations in XSLT
Data to Model Transformations

• Necessary to turn legacy data and models in alternative modeling languages into MOM models.

Example:
XsdToMom: Turns XML schemas into object models:

Example:
Model extractor used in standard-driven development:

defines ‘artefact types’ in terms of standard legalese

defines ‘artefact types’ in terms of a formal object model

strongly-typed code to process instances of artefact types
Example: PII System Authority

Resources
- Representation.xsd
- XsdToMom
- XModel.xml
- Representation.mom
- LW code generator
- XmlSerialization.cs
- Generated XML Access
- XModel.cs
- Generated Navigation
- Generated Model

Infomodels
- XModel.mml
- SAInformationModel.cs
- SAMetaModel.cs
- XModel.mom
- ResToSam
- HW code generator
- String Template
- Template.stg
- model to model transformation
- model to LW code transformations
- model to HW code transformations

Metamodel
- SAINformationModel.xsd
- XsdToMom
- SAINformationModel.mom
- SAMetaModel.mom
- LW code generator
- Generated Model

Toolkits
Standard-Driven Development (SDD)

Observations:
- A standard, i.e. the *definition* of a standard, is *data*. If sufficiently formal, that data can be processed and interpreted computationally.
- When working with a standard in practice, manual reproduction of artefacts defined by the standard will inevitably lead to errors.

Basic idea of standard-driven development:
- Turn the standard into structured data that can be interpreted (if that is not already the case) → extract *(meta)*models from the standard
- Generate the necessary software support for using the standard directly from the standard → generate code from the *(meta)*models
Standard-Driven Development Design Pattern (1)

standard definition

- instance definition
  - defined in terms of
    - artefact definition
      - model extractor
        - artefact model
          - code generator
            - support for processing instances of artefact types

- code generator
  - code
    - support for using instances defined in the standard

instance extractor
  - artefact instance
    - is instance of
      - code
        - defined in terms of
          - artefact definition
            - model extractor
              - artefact model
                - code generator
                  - support for using instances defined in the standard
Extensible Standards

An *extensible standard* defines the mechanisms to extend the standard as part of the standard itself (by means of a *metastandard*).

Examples:

- **HL7 version 3**: extensible *messaging* framework
  - RIM (Reference Information Model) defines core concepts.
  - MIF (Model Interchange Format) provides means to define extensions (DMIMs, RMIMs, CMETs) based on RIM.
- **openEHR**: extensible *EHR* framework
  - openEHR Reference Model defines core concepts.
  - ADL (Archetype Definition Language) provides the means to define extensions (‘archetypes’).
Standard-Driven Development Design Pattern (2)

- Metastandard definition
  - Instance definition
  - Artefact definition
  - Metamodel definition

- Support for using instances defined in the standard
- Support for processing instances of artefact types
- Support for processing artefact types

Levels:
- Level 1
- Level 2
- Level 3
Applications of SDD to Existing Standards

- HL7v3 RIM (Reference Information Model)
- HL7v3 CDA (Clinical Document Architecture)
- UCUM (Unified Code for Units of Measure)
- DICOM SR (Structured Reporting)
- DICOM IODs (Information Object Definitions)

More details can be found in:


URL: [http://sdrv.ms/Z9aDBy](http://sdrv.ms/Z9aDBy)

A similar approach (based on EMF) is used in the MDHT (Model-Driven Health Tools) project for CDA Documents.
Applying SDD to New Standards