challenges in domain-specific modeling

raphaël mannadiar

august 27, 2009
outline

1. introduction
2. approaches
3. debugging and simulation
4. differencing
5. evolution
6. (transformations)
7. (dsl engineering)
8. conclusion
introduction

1. introduction

2. approaches

3. debugging and simulation

4. differencing

5. evolution (transformations)

6. (transformations) (dsl engineering)

7. (dsl engineering)

8. conclusion
why not abstract?
generated code less efficient? general purpose languages less expressive?

why abstract?
▷ mapping to develop, maintain, debug... is error prone and difficult
▷ increased productivity compensates for loss in efficiency
▷ domain-specific languages should be less expressive
how is productivity increased?

- user’s mental model of problem is closer to “implementation”
- more intuitive and less error-prone development
  → dsm environment constrains user to create valid domain models
- leverage expertise
  → domain experts play with domain models
  → programming experts play with APIs and frameworks
  → domain, programming and transformation experts play with model-to-artifact transformations
how is productivity increased?

- user’s mental model of problem is closer to “implementation”
- more intuitive and less error-prone development
  → dsm environment constrains user to create valid domain models
- leverage expertise
  → domain experts play with domain models
  → programming experts play with APIs and frameworks
  → domain, programming and transformation experts play with model-to-artifact transformations

→ increased productivity
modeling concepts

why model?

models are cheaper, safer and quicker to build, reason about, test and modify than the systems they represent
modeling concepts

why model?

*models* are cheaper, safer and quicker to build, reason about, test and modify than the systems they *represent*

defining models

a *metamodel* defines a set of entities, associations and constraints that determine a possibly infinite set of *conforming* models
modeling concepts

why model?

*models* are cheaper, safer and quicker to build, reason about, test and modify than the systems they *represent*

defining metamodels

common approaches are *graph grammars* and (augmented) *uml class diagrams*

defining models

a *metamodel* defines a set of entities, associations and constraints that determine a possibly infinite set of *conforming* models
why model?

*models* are cheaper, safer and quicker to build, reason about, test and modify than the systems they *represent*.

defining metamodels

common approaches are *graph grammars* and (augmented) *uml class diagrams*.

defining models

a *metamodel* defines a set of entities, associations and constraints that determine a possibly infinite set of *conforming* models.

defining model semantics

common approach is mapping down to domains with well-defined semantics (*e.g.* mathematics, *statecharts*, python).
dsm vs. code generation

traditional code generation...

not popular because generated code is often awkward, inefficient, inflexible and/or incomplete

→ source domain is too large
→ target domain is too large
dsm vs. code generation

**traditional code generation**

not popular because generated code is often awkward, inefficient, inflexible and/or incomplete

→ source domain is too large
→ target domain is too large

**but!**

**dsm is different**

▷ source domain restricted from all models of all applications to models of applications from 1 domain
▷ target domain restricted from all applications to applications from 1 domain

→ enables generation of complete and optimized artifacts
the “coding community” has mature tools that facilitate

- editing
- debugging
- differencing
- versioning

of text-based artifacts (e.g., code, xml)
the “coding community” has mature tools that facilitate

- editing
- debugging
- differencing
- versioning

of text-based artifacts (e.g., code, xml)

how can these activities and their underlying principles be generalized to DSM?
introduction

2 approaches

3 debugging and simulation

4 differencing

5 evolution

6 (transformations)

7 (dsl engineering)

8 conclusion
generative programming (gp)

basic idea

bring software engineering to the same level of automation as other forms of manufacturing i.e.,

- standardized components (e.g., $\frac{1}{4}$" bolts)
- standardized interfaces (e.g., category B plug)
- customizable assembly lines (e.g., same line for red and blue Corollas)
Generative Programming (GP)

**Basic Idea**

- Bring software engineering to the same level of automation as other forms of manufacturing, i.e.,
  - Standardized components (e.g., \( \frac{1}{4} \)" bolts)
  - Standardized interfaces (e.g., category B plug)
  - Customizable assembly lines (e.g., same line for red and blue Corollas)

**Example**

Instead of coding a LinkedList, an ArrayList and a SyncList, code a List\(<T>\) which can be “instantiated” with arbitrary “configurations”
generative programming (gp)

**basic idea**

bring software engineering to the same level of automation as other forms of manufacturing i.e.,

- standardized components (e.g., $\frac{1}{4}$” bolts)
- standardized interfaces (e.g., category B plug)
- customizable assembly lines (e.g., same line for red and blue Corollas)

**example**

instead of coding a LinkedList, an ArrayList and a SyncList, code a List<T> which can be “instantiated” with arbitrary “configurations”

**gp vs. dsm**

an appropriate technique for implementing domain frameworks
the **object management group**'s (omg) approach to model-driven engineering

**basic idea**

- software development viewed as a series of model refinements where lower and lower level models (referred to as *platform-specific models*) are (semi-)automatically generated from higher level ones (referred to as *platform-independent models*)

- modelers are expected to modify and contribute to generated intermediate models
model-driven architecture (mda)

the object management group’s (omg) approach to model-driven engineering

**basic idea**

- software development viewed as a series of model refinements where lower and lower level models (referred to as platform-specific models) are (semi-)automatically generated from higher level ones (referred to as platform-independent models)
- modelers are expected to modify and contribute to generated intermediate models

**mda vs. dsm**

▷ between UML modeling and dsm...
▷ interaction with intermediate models prevents true raise in abstraction
metamodeling

---

**basic idea**

- complex operations on models and metamodels should not be developed from scratch for every metamodel
- they should take metamodels as parameters
- hence, all metamodels should conform to a *metametamodel*
metamodelling

basic idea

- complex operations on models and metamodels should not be developed from scratch for every metamodel
- they should take metamodels as parameters
- hence, all metamodels should conform to a *metametamodel*

example

one generic tool used as a modeling environment for any metamodel
metamodeling

**basic idea**
- complex operations on models and metamodels should not be developed from scratch for every metamodel
- they should take metamodels as parameters
- hence, all metamodels should conform to a *metametamodel*

**example**
one generic tool used as a modeling environment for any metamodel

**metamodeling vs. dsm**
there is a consensus that metamodeling is the key to empowering model based techniques
1 introduction

2 approaches

3 debugging and simulation

4 differencing

5 evolution (transformations)

6 (dsl engineering)

7 conclusion

rafaël mannadiar  challenges in domain-specific modeling  24/59
simulating a model empowers the modeler to test and reason about its behavior
simulation

**premise**

Simulating a model empowers the modeler to test and reason about its behavior.

**approach 1: hard-coded simulators**

The behavioral semantics of a formalism are hard-coded in a tool that can simulate conforming models.
simulation

premise
simulating a model empowers the modeler to test and reason about its behavior

approach 1: hard-coded simulators
the behavioral semantics of a formalism are hard-coded in a tool that can simulate conforming models

approach 2: rule-based simulators
- rules define “simulation steps”
- simulating equals the sequential (and interactive) application of these rules
- a metamodeling tool can generate a simulation environment from these rules
premise

- error tracking and reproduction are key activities in debugging software
- modern coding tools allow setting/clearing breakpoints, stepping over/into expressions, pausing/resuming execution and reading field values
- these facilities should also be offered by model debugging tools
**premise**

- error tracking and reproduction are key activities in debugging software
- modern coding tools allow setting/clearing *breakpoints*, stepping *over/into* expressions, pausing/resuming execution and reading field values
- these facilities should also be offered by model debugging tools

**current best approaches...**

- deal with textual DSLs only
- instrument code generation rules to store mapping of DSL statements to GPL statements
- instrument code generation rules such that generated GPL code updates DSL variable values
- reuse GPL debuggers (*e.g.*, gdb, jdb) to provide debugging operations at the DSL level (*e.g.*, a breakpoint set in the DSL code will call jdb’s breaking function from the matching line in the generated Java code)
outline

1. introduction
2. approaches
3. debugging and simulation
4. differencing
5. evolution (transformations)
6. (dsl engineering)
7. conclusion
computing differences

**premise**

- means to merge, version and store sequential and parallel versions of models are needed
- means to visualize differences between models are needed
computing differences

premise

- means to merge, version and store sequential and parallel versions of models are needed
- means to visualize differences between models are needed

lexical differencing approaches

- differentiate between textual documents (e.g., code, xml)
- no sense of semantically meaningful and meaningless differences (e.g., layout changes)
- no sense of design-level differences

→ wrong level of abstraction
computing differences...

**model differencing approaches**

1. create some kind of abstract syntax graph (asg) of the models
2. establish matches between both asgs using *unique identifiers* or *syntactic and structural similarities*
3. determine creations, deletions and changes from one asg to the other

metamodel-specific and -independent approaches exist
model differencing approaches

1. create some kind of abstract syntax graph (ASG) of the models
2. establish matches between both ASGs using *unique identifiers* or *syntactic and structural similarities*
3. determine creations, deletions and changes from one ASG to the other

metamodel-specific and -independent approaches exist

unique identifiers
- 100% reliable matching
- tool dependence/lock-in

similarity heuristics
- tool independent
- sensitive to principled versioning
representing differences

premise

given a difference $\Delta$ between two models, how can it be represented?
representing differences

**premise**

given a difference $\Delta$ between two models, how can it be represented?

**edit scripts approaches**

- differences are sequences of invertible operations (e.g. create element, modify attribute) which specify how a model can be procedurally turned into another
- low readability for humans
representing differences

premise

given a difference $\Delta$ between two models, how can it be represented?

edit scripts approaches

- differences are sequences of invertible operations (e.g. create element, modify attribute) which specify how a model can be procedurally turned into another
- low readability for humans

coloring approaches

- overlay 2 models and color differences; more familiar to modeler but doesn’t scale
- color document object model (dom) like view of the model; more compact and scalable
representing differences

premise

given a difference $\Delta$ between two models, how can it be represented?

edit scripts approaches

- differences are sequences of invertible operations (e.g. create element, modify attribute) which specify how a model can be procedurally turned into another
- low readability for humans

coloring approaches

- overlay 2 models and color differences; more familiar to modeler but doesn’t scale
- color document object model- (dom) like view of the model; more compact and scalable

difference models

- differences are models
- enables the use of higher-order transformations to manipulate, apply, merge, invert and represent model differences
- tool-, metamodel- and differencing method-independent
introduction

approaches

debugging and simulation
differencing

evolution (transformations)

dsl engineering

conclusion

1 introduction

2 approaches

3 debugging and simulation

4 differencing

5 evolution

6 (transformations)

7 (dsl engineering)

8 conclusion

raphaël mannadiar

challenges in domain-specific modeling
## Sources of Evolution

### Domain-Driven

- dsls are tightly coupled with their domain
- domain changes can spawn metamodel changes
- these can syntactically and/or semantically invalidate existing models and transformations
sources of evolution

**domain-driven**
- dsls are tightly coupled with their domain
- domain changes can spawn metamodel changes
- these can syntactically and/or semantically invalidate existing models and transformations

**target-driven**
- model transformations may produce artifacts that “interact” with some target platform (e.g. API, device)
- changes in the target may invalidate these transformations and force evolution
sources of evolution

**domain-driven**
- dsls are tightly coupled with their domain
- domain changes can spawn metamodel changes
- these can syntactically and/or semantically invalidate existing models and transformations

**target-driven**
- model transformations may produce artifacts that “interact” with some target platform (e.g. API, device)
- changes in the target may invalidate these transformations and force evolution

**convenience-driven**
- language extensions and new syntactical constructs maybe added to a language
- these typically shouldn’t invalidate existing models
model and model interpreter co-evolution

traditional approach: do it yourself

manually co-evolve models and model interpreters as metamodels evolve
model and model interpreter co-evolution

**traditional approach: do it yourself**

manually *co-evolve* models and model interpreters as metamodels evolve

**current best approaches... (models)**

- distinguish between “easy” and “difficult” metamodel changes
- use higher-order transformations to generate model co-evolution transformations from metamodel difference models
model and model intrepreter co-evolution

traditional approach: do it yourself

manually co-evolve models and model intrepreters as metamodels evolve

current best approaches... (models)

- distinguish between “easy” and “difficult” metamodel changes
- use higher-order transformations to generate model co-evolution transformations from metamodel difference models

only current approach... (intrepreters)

- instrument model co-evolution rules with instructions to rewrite code patterns in coded model intrepreters
outline

1. introduction
2. approaches
3. debugging and simulation
4. differencing
5. evolution (transformations)
6. (transformations)
7. (dsl engineering)
8. conclusion
specifying transformations

with code

- transformations are imperative code programs
- complicates use of higher-order transformations
- intent of transformation may be lost in implementation details
specifying transformations

**with code**
- transformations are imperative code programs
- complicates use of higher-order transformations
- intent of transformation may be lost in implementation details

**with rules**
- rules contain a *pattern*, a *guard* and a *body*
- more modular and abstract than coded transformations
## Specifying Transformations

### With Code
- Transformations are imperative code programs.
- Complicates use of higher-order transformations.
- Intent of transformation may be lost in implementation details.

### With Rules
- Rules contain a *pattern*, a *guard*, and a *body*.
- More modular and abstract than coded transformations.

### With XSLT
- Serialize models to XML and then transform XML using XSLT.
- Awkward transformations due to tree-based nature of XML vs. graph-based nature of models.
- Lacking expressiveness for complex transformations.
- Readability and scalability issues.
- Lacking means of error reporting.
### specifying transformations

<table>
<thead>
<tr>
<th>with code</th>
</tr>
</thead>
<tbody>
<tr>
<td>- transformations are imperative code programs</td>
</tr>
<tr>
<td>- complicates use of higher-order transformations</td>
</tr>
<tr>
<td>- intent of transformation may be lost in implementation details</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>with rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>- rules contain a <em>pattern</em>, a <em>guard</em> and a <em>body</em></td>
</tr>
<tr>
<td>- more modular and abstract than coded transformations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>with xslt</th>
</tr>
</thead>
<tbody>
<tr>
<td>- serialize models to xml and then transform xml using xslt</td>
</tr>
<tr>
<td>- awkward transformations due to tree-based nature of xml vs. graph based nature of models</td>
</tr>
<tr>
<td>- lacking expressiveness for complex transformations</td>
</tr>
<tr>
<td>- readability and scalability issues</td>
</tr>
<tr>
<td>- lacking means of error reporting</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>with pre-/post- conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- pre-conditions express conditions the host model must satisfy for the rule to be applicable</td>
</tr>
<tr>
<td>- post-conditions express conditions the host model must satisfy after the run has been applied</td>
</tr>
<tr>
<td>- declarative approach well suited for transformation <em>bi-directionality</em></td>
</tr>
<tr>
<td>- power contingent on constraint solving facilities</td>
</tr>
</tbody>
</table>
specifying transformations...

- rule-based approach
- left-hand side and right-hand side patterns (which use domain concepts)
- theoretically founded
- possible bi-directionality achievable via *triple graph grammars*
executing rule-based transformations...

default graph grammar semantics

- any applicable rule may run
- stop when no more rules are applicable
- lacking facilities for determinism and scheduling
executing rule-based transformations...

**default graph grammar semantics**
- any applicable rule may run
- stop when no more rules are applicable
- lacking facilities for determinism and scheduling

**structured approaches**
- rule-based approaches become more powerful when control flow and scheduling mechanisms are added
- some tools offer conditions, loops, transactions and hierarchy
- these may be reflection-based or graphical
outline

1 introduction
2 approaches
3 debugging and simulation
4 differencing
5 evolution (transformations)
6 (dsl engineering)
7 (dsl engineering)
8 conclusion
weaving features together

**traditional approach**

1. study the domain
2. extract domain concepts, associations and constraints
3. express these in an augmented class diagram
weaving features together

**traditional approach**

1. study the domain
2. extract domain concepts, associations and constraints
3. express these in an augmented class diagram

**possible future approach: feature weaving**

- **motivation**: a new formalism where notions of state and transition exist may benefit from reusing parts or all of the statechart formalism
- **idea**: inspired from aspect-oriented development where modularly defined concerns are weaved together with core concerns to form complete systems

1. determine basic feature set for “all” dsls (e.g., state-based, continuous time)
2. select basic features of a dsl
3. compose them somehow to yield new dsl

- very modular approach axed on reusability
- synthesized dsls should remain bound to the features composing them allowing for automatic generation of certain artifacts (e.g., basic simulators)
**Outline**

1. Introduction
2. Approaches
3. Debugging and simulation
4. Differencing
5. Evolution (transformations)
6. (DSL engineering)
7. Conclusion

Raphaël Mannadiar

Challenges in Domain-specific Modeling
Recap

- Over the past decades, software development has naturally evolved towards DSM.

- DSM improves productivity by reducing the conceptual gap between the requirements and the solution.

- To replace traditional software development approaches, robust and scalable means to simulate, debug, difference, version, transform and co-evolve models are required.

- DSL engineering may benefit from techniques from aspect-oriented development.
thanks!