Language engineering and Domain Specific Languages

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

1. Defining languages
2. General purpose languages vs domain specific languages
3. Internal and external DSLs
4. What about DSMLs?
5. Technologies

Examples in the remaining labs: you must do the labs!

We quickly revisit some of the ideas that we already saw in the context of defining UML – a general purpose modelling language – but now focus on domain-specific language definition.

How to define language

Recall: languages have syntax, semantics, pragmatics

Language designer defines syntax

for modelling languages, split into:

- abstract (information needed to determine the semantics, e.g. what elements are there, how are they related)
- concrete (what the user actually says, e.g. specific keywords, shapes of graphic elements)

answer questions like:

“what expressions are there in this language?”

“is this expression legal in the language?”

Language designer defines semantics; pragmatics emerge

What does this legal expression mean?

e.g., if I give you a model and a system, does the system conform to the model? Is this system one of the things the model could mean?

Pragmatics are emergent: not defined by the language designer, but by users.

Hold on a moment! Why define languages?

Why is there more than one programming language?

(pause for discussion...)

If our domain is difficult to work with, given existing languages, what are our options?

Are we talking about programming languages or modelling languages, textual or graphical? What's the difference?

Options for programming for a domain

e.g., you're in overall charge of your organisation's software... You could

- do nothing special: after all, your favourite language is Turing complete, it can do anything;
- write some classes and reuse them;
- develop a library;
- develop a framework;
- make your APIs provide a *fluent interface*;
- develop a domain-specific language.

DSLs vs GPLs

Quote and table from Markus Voelter, *DSL Engineering*, `http://dslbook.org`

"DSLs pick more characteristics from the third rather than the second column"

	GPLs	DSLs
Domain	large and complex	smaller and well-defined
Language size	large	small
Turing completeness	always	often not
User-defined abstractions	sophisticated	limited
Execution	via intermediate GPL	native
Lifespan	years to decades	months to years (driven by context)
Designed by	guru or committee	a few engineers and domain experts
User community	large, anonymous and widespread	small, accessible and local
Evolution	slow, often standardized	fast-paced
Deprecation/incompatible changes	almost impossible	feasible

DSL or GPL?

- Java
- Haskell
- UML
- SQL
- HTML
- Makefile
- Use case diagrams
- scientific notation for numbers, e.g. 2.3×10^{17}

Not an absolute binary – can ask "how general purpose?" or "specific to what domain?". How big does a domain have to be before it counts as general? Still, a useful distinction in practice.

Internal vs external DSL

Internal DSL:

- DSL text occurs within a program written in a host GPL
- parsed along with the GPL
- convenient for programmers who know the host GPL
- Example: SQL queries embedded in Java programs

External DSL:

- DSL program is free-standing
- with its own parser etc.
- can suit non-programmers.
- Example: a makefile

Pros and cons of developing a DSL

Pro:

- can get exactly the features you want
- can keep it simple otherwise
- so it can be made concise and usable

Con:

- users won't know it already
- development work!
- especially, maintenance: of the language; of its tools; of the programs in it

How to make a DSL program do something?

Two strategies:

- interpretation: there's an engine that can read the DSL program and do something as a result;
- code generation: there's a translation from the DSL program to something else (and an existing mechanism to make the something else do something).

Which comes first, abstract or concrete syntax?

Abstract: get clean, consistent idea of the concepts to express, and then design the perfect concrete syntax for them – or maybe more than one?

Concrete: work out what you want to say using examples, then systematise and abstract.

Recall the paleo example: we started with just one example in a concrete syntax, and worked out an abstract syntax.

Best of both worlds? use concrete syntax examples to elucidate abstract concepts, then reflect on the concrete syntax and improve it?

Language usability – which involves semantics as well as syntax – is a fascinating area we don't have time for here.

And where does defining the semantics come in?

In practice, you always know the semantics you're trying to express, as you define the syntax. At least roughly...

Making that precise is then very hard...

For programming languages: semantics is usually "what the reference compiler/interpreter does". Can define formally operationally, denotationally, axiomatically...

For modelling languages...?

(Why the difference?)

Abstract syntax for graphical languages

Key problem: not tree-like!

Key solution: metamodelling

meta

We usually say "a metamodel is a model of the model" but what we mean is more like

"a metamodel is a model that describes all the possible models"

Just think about class diagrams for a moment:

model of a system describes how the objects can be configured: a running instance of the system conforms to the model, is an instance of the model;

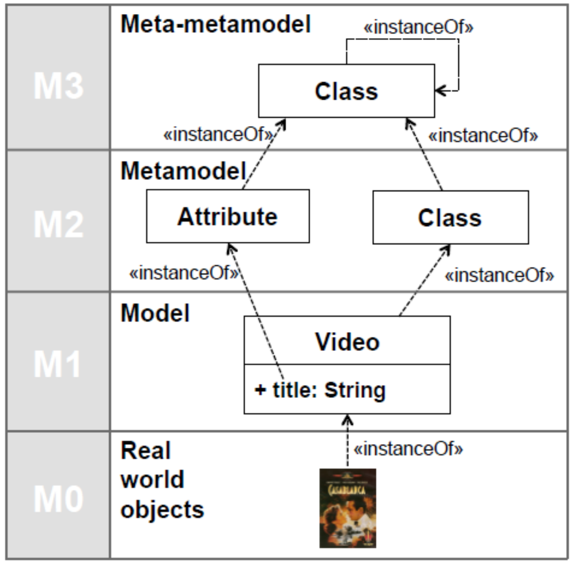
"the system described by the model" is approximately all the configurations the objects can be in, according to the model

The model constrains those configurations: only some are legal.

metamodel of a modelling language is what we get when we want the models to be the configurations. It tells us which models are legal. A model conforms to a metamodel. The metamodel is the model of the modelling *language*.

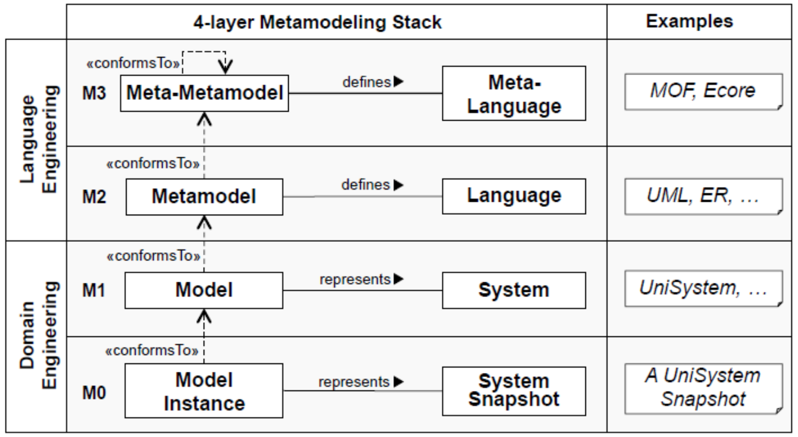
metamodel defines *abstract* syntax

OMG 4-level metamodel hierarchy



Picture from Brambilla et al., *MDSE in Practice*

In more detail:



Picture from Brambilla et al., *MDSE in Practice*

Connecting to the Eclipse world

OMG's hierarchy aims to be language-neutral.

Eclipse tools implement it in Java, supporting facilities like generating Java code from a model, serialising it in XML, etc.

EMF, the Eclipse Modeling Framework, supports those tasks.

ecore is a metamodel, used within EMF, for defining models held in Eclipse. Corresponds to (part of) MOF in OMG terms.

Sketch of one way to define a DSML

1. abstract syntax: define metamodel using ecore
2. concrete syntax: define diagrammatic elements etc. e.g. using EMF/GMF tools or EuGENia
3. semantics: (e.g.) by generating Java code.

To allow people to use it, generate a model editor in Eclipse (with validation etc.).

Lab coming up.

Sketch of one way to define a textual DSL

1. concrete syntax: define a grammar using xtext
2. abstract syntax: generated from the grammar (can be expressed in ecore)
3. semantics: (e.g.) by generating Java code.

To allow people to use your language, generate an editor in Eclipse (with validation etc.)

Lab coming up.

Code generation using xtend

xtend is "essentially Java on steroids with awesome expressive notation that lets you express your logic in super elegant ways not possible in Java itself. Xtend compiles to Java source so anything that requires code written in Java can make use of code written in Xtend and Xtend can use anything written in Java so it's totally seamless." Ed Merks

It's much easier to write code generation in xtend than in Java.

We may see a little of this in the labs, but won't have time to go into much detail.

Why do we discuss DSLs and DSMLs together?

You might think these were just two different choices: what are they doing in the same lecture?

In practice the line between them is blurred *by the very separation of concrete and abstract syntax we've discussed*.

Abstract syntax: an abstract model, saying what's in your language.

Concrete syntax: how the user sees it, could be graphical or textual, or something else.

You saw in the OCL lab that there are several (many!) "ecore editors", working with different presentations of an ecore model, and Eclipse happily switches between them: it's just concrete syntax...

What is a programming language anyway?

Maybe it's a model... (Everything's a model!)

"Any sufficiently advanced technology is indistinguishable from magic."

Arthur C. Clarke, Third Law.

Caution

Arthur C. Clarke's second law also applies.

"The only way of discovering the limits of the possible is to venture a little way past them into the impossible."

Currently, you can distinguish DSL engineering technology from magic by the flakiness of the tools.

Will that ever go away? Well... maybe.*

* I am neither elderly nor distinguished enough to fall into the trap of saying it's impossible. [Look up Clarke's first law.]

Conclusion

We could very easily have an entire course on domain specific languages.

Further reading (beyond what's required for this course!)

- Martin Fowler, *Domain specific languages*
- Ralf Lämmel, *The Software Languages Book*
- Markus Voelter, *DSL Engineering* `http://dslbook.org/`
- Lorenzo Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*
- Brambilla et al., *MDSE in Practice* `https://www.sites.google.com/site/mdsebook/`