Modeling: From CASE Tools to SLE and Machine Learning

Jean-Marc Jézéquel

▶ **To cite this version:**

Jean-Marc Jézéquel. Modeling: From CASE Tools to SLE and Machine Learning. Bertrand Meyer. The French School of Programming, Springer, pp.1-22, In press. hal-04080311

HAL Id: hal-04080311
https://hal.science/hal-04080311v1

Submitted on 24 Apr 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modeling: From CASE Tools to SLE and Machine Learning

Jean-Marc Jézéquel
Univ Rennes, Inria, CNRS, IRISA
jezequel@irisa.fr

September 29, 2022

Abstract

Finding better ways to handle software complexity (both inherent and accidental) is the holy grail for a significant part of the software engineering community, and especially for the Model Driven Engineering (MDE) one. To that purpose, plenty of techniques have been proposed, leading to a succession of trends in model based software developments paradigms in the last decades. While these trends seem to pop out from nowhere, we claim in this article that most of them actually stem from trying to get a better grasp on the variability of software. We revisit the history of MDE trying to identify the main aspect of variability they wanted to address when they were introduced. We conclude on what are the variability challenges of our time, including variability of data leading to machine learning of models.

1 Introduction

Software suffers from a very striking paradox. One the one hand it is so easy to write simple programs that a 6 years old child can do Logo or Scratch programming right after a few minutes of training. On the other hand, it is so difficult to write complex ones that basically nobody is able to write large, bug free programs[1]. At the scale of a 100 lines program, the method or the programming language used does not really matter much, and if you fail, you can just start over again at very little cost. However it is well-known from Fred Brooks' days [3], that writing a 100,000 line program is much more difficult than 1000 times the effort of writing a 100 line program.

[1] Even most real life "proven" programs still have bugs in corner cases, stemming from unexpected conditions.

We can then identify 3 dimensions of complexity in building software:

Inherent complexity of software This is due to software roots in Alan Turing's theory of universal computation: Even extremely short and simple programs may be impossible to prove or even have undecidable properties (termination).

Complexity due to scale Within this dimension, Fred Brooks [3] identifies two types of software complexity: *essential* and *accidental* complexity. On the one hand, *essential complexity* is software complexity that is inherent to the problem targeted by the software or to the solution provided by the software. This inherent complexity can arise, for example, from the variety of input events and data that must be correctly handled by the software, or from the critical functional and quality concerns that the software must address. For example, software controlling a commercial aircraft is inherently complex from both a problem and solution perspective. On the other hand, *accidental complexity* comes from the use of inappropriate technologies, which leads to significant human effort being spent on developing the software.

Complexity due to uncertainty Uncertainty in software development comes from many sources. First it comes from the problem domain: requirements, including business or legal rules and human expected behavior are generally incomplete and do evolve over time. Assumptions about the world the software interacts with are typically quite rough, usually implicit, and do not take into account all corner cases. Uncertainty can also come from the execution platform, either inherently (hardware faults or even simple network delays) or accidentally due to e.g.; misinterpretations or changes in APIs. This is not even counting with cyber-attacks and other malicious tampering with the software.

Dealing with the inherent complexity of software requires reasoning about the software, using techniques ranging from full formal proofs to lightweight mechanisms such as Design By Contract [31] or Unit Testing [19].

Due to the limited capacity of human mind (nobody can fully understand a program made of one million lines of code), dealing with the complexity due to scale can only be achieved through abstraction and modularity. If a program is properly broken into 10 modules, each made of 10 submodules, each made of another 10 submodules, and each of these leaf modules containing 1000 lines of code, then we can both fully understand any given module and how this one fits with the other 999, provided we have an abstract understanding of those (*what* they do, not *how* they do it).

Dealing with the complexity due to uncertainty requires a proper anticipation of eventual changes, by identifying and isolating (or at least reducing the coupling

with) the parts that could change. This is the idea of Separation of Concerns, along with the explicit management of variations, along the two dimensions of space and time for respectively variants and versions. Deciding which variant should be chosen can be done at different stages in the software lifecycle: at requirement time, design time, compile time, load time, JIT time, or even runtime [18].

If you consider variability management of essential complexity as the main concern in software development, it is interesting to look back at the history of trends in Software Engineering to see what particular problem of variability management these trends actually wanted implicitly or explicitly to address. In this paper we roughly identify 4 generations, that are outlined in the next sections:

- 1st Generation: CASE tools
- 2nd Generation: Model Driven Architecture
- 3rd Generation: Separation of Concerns with Models, Aspects and Features
- 4th Generation: Domain Specific Languages & Software Language Engineering

Each of these sections first starts with the presentation of the code idea, then highlights its good points, and finally list its drawbacks (the Bad and the Ugly).

We conclude with what could be the challenges of the next generation: the integration of Models & Data.

2 First Generation: CASE tools

2.1 The Idea

Computer Assisted Software Engineering (CASE) tools were born in the 80's, with features such as consistency checking, validation and code generation starting to be used in some industries such as telecommunications.

The telecom industry (actually mostly the European one) indeed started to use a number of so-called Formal Description Techniques, from SDL (Specification and Description Language) to Estelle or Lotos (Logic of Temporal ordering of events). SDL and Estelle were based on the notion of extended state machines, whereas Lotos was based on process algebra. These languages featured a fairly well-defined syntax (both textual and graphical in the case of SDL, see Figure 1) and semantics, and were thus amenable to simulation and early attempts to scale model-checking to industrial strength.

Once confidence could be built into a given specification through simulation and validation, some CASE tools allowed engineers to use code generation to get

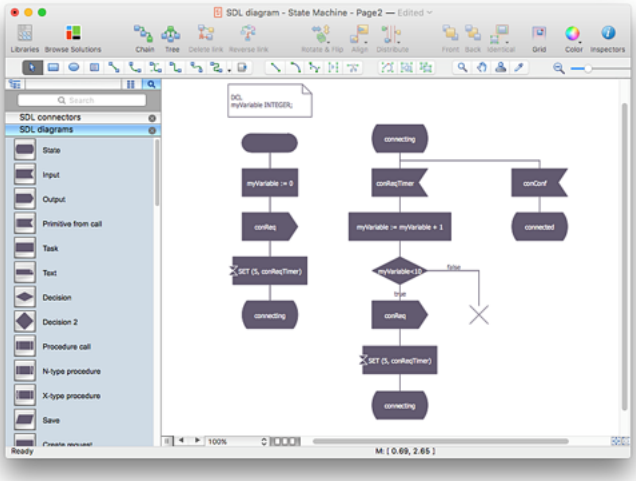

Figure 1: A CASE Tool for SDL

an instant implementation of their algorithms and protocols. As a matter of fact, one of my early contribution to this field was to build such a code generator for Estelle [23], targeting the early distributed computers of that time, ranging from dedicated machines such as the Intel hypercubes to more mundane networks of Unix computers.

2.2 The Good

The promises of this approach were great: you could program these very complex distributed computers at a high level of abstraction, with a high level of confidence in the validity of your code because the simulation/validation/model-checking could be performed on the exact same source code. What was actually achieved in software engineering terms was a clear separation of the *essential* complexity (the specification of a protocol) from the *accidental* complexity of the implementation, thus making it easier to evolve the specification to meet new requirements [20].

2.3 The Bad and the Ugly

There were however two main drawbacks to this approach. The first one is due to the highly abstract and somehow mathematical nature of these Formal Description Techniques. It was in fact quite difficult to train large numbers of telecom engineers to use these formalisms. SDL was kind of OK, but Lotos was considered as way

too baroque for C programmers, with Estelle and its Pascal-like syntax somehow in between.

The second drawback, which is probably the worst one, is linked to code generation. Code generators at that time were black boxes, heavily protected by the CASE tool vendors. In some cases, the generated code perfectly fitted the engineering needs, and thus several successful uses of this technology were documented (and leveraged by the vendor marketing departments). However in many other cases there were difficult engineering constraints to care of, such as speed, code compacity, memory footprint, memory usage, and in many cases interface with legacy software or firmware. These constraints would each have required the (functional) code to be generated in slightly different ways, but that was impossible with black box code generators.

In some cases, engineers tried to circumvent the problem by adding a post-processing step after the code has been generated. For example, I remember the case at a large telecom company that was using an SDL code generator and had the issue of a state machine transition that took too long to execute. The consequence was that some real time constraints could not be met. So the engineers broke the transition into two blocks, yielding the control to the scheduler in the middle. That worked for meeting the real time constraints. However there was a catch that was only seen much later after a catastrophic crash of the system. By breaking the transition, that in SDL semantics was atomic, the engineers involuntarily introduced new behavior with a different interleaving of event processing that in some rare cases led to this crash.

In the end, one size fits all black box code generation was rapidly abandoned in the industry. The bottom line was that it was not worth the trouble.

3 The Time of Model Driven Architecture (MDA)

3.1 The Idea

In [32] the OMG Architecture Board described its Model Driven Architecture (MDA) vision to support interoperability with specifications that address integration through the entire systems life cycle: from business modeling to system design, to component construction, to assembly, integration, deployment, management, and evolution. MDA had the primary goal to separate the fundamental logic behind a specification from the specifics of the particular middleware that implements it. This would allow rapid development and delivery of new interoperability specifications that use new deployment technologies but are based on proven, tested business models. In the OMG vision, organizations would use MDA to meet the integration challenges posed by new platforms, while preserving their investments in existing

business logic based on existing platforms. The term platform is used here to refer to technological and engineering details that are irrelevant to the fundamental functionality of a software component.

Main concepts in MDA are:

CIM: a Computation Independent Model focuses on the context and requirements of the system without consideration for its structure or processing.

PIM: a The Platform Independent Model focuses on the operational capabilities of a system outside the context of a specific platform (or set of platforms) by showing only those parts of a complete specification that can be abstracted out of that platform.

PSM: a Platform Specific Model augments a Platform Independent Model with details relating to the use of a specific platform.

PDM: a Platform Description Model describes set of subsystems and technologies that provide a coherent set of functionality through interfaces and usage patterns, e.g.; CORBA, Java/EJB, C#/.NET etc.

Model Transformations are automated ways of modifying and creating models.

Early on, model transformations were developed within modeling tools using proprietary languages. Such a method was unfortunately far from being reusable and robust. Model transformations were increasingly seen as vital assets that must be managed with sound software engineering principles: they must be analyzed, designed, implemented, tested, maintained and be subject to configuration management. For the same reason that domain know-how should not be tied to a particular platform, it was considered critical that model transformations were not prisoners of a given CASE tool. So the OMG launch a RFP on MOF QVT (Query/View/Transformation) in order to define a vendor independent Model Transformation language.

3.2 The Good

Since domain specific models are less likely to change rapidly than platform-specific ones, the MDA core idea made a lot of sense: it should be possible to capitalize on platform-independent models (PIM), and more or less automatically derive platform-specific models (PSM) —and ultimately code— from PIM through model transformations [2].

According to [32], that vision would provide:

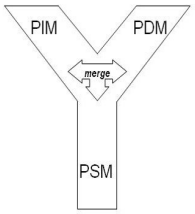

Figure 2: Y-development

- Portability, increasing application re-use and reducing the cost and complexity of application development and management, now and into the future.
- Cross-platform Interoperability, using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions.
- Platform Independence, greatly reducing the time, cost and complexity associated with re-targeting applications for different platforms —including those yet to be introduced.
- Domain Specificity, through Domain-specific models that enable rapid implementation of new, industry-specific applications over diverse platforms.
- Productivity, by allowing developers, designers and system administrators to use languages and concepts they are comfortable with, while allowing seamless communication and integration across the teams.

3.3 The Bad and the Ugly

In its most idealized form, illustrated in Figure 2, the MDA would combine a PIM with a PDM to automatically provide a PSM. The reality however is a little bit more complex than this idealized vision, which probably explains why the MDA never quite took real momentum.

First, MDA is mostly a forward engineering approach in which models are transformed into implementation artifacts (e.g. code, database schema, software configuration scripts, test cases) in one direction via a fully or partially automated generation step. When not everything is captured in the source models, some modification of the generated code has to be carried out manually to take into account the missing concerns. That rapidly becomes a nightmare from the maintenance point of view, even if some vendors came up with tricks to alleviate the burden of keeping the manual modifications in synch with the generated code.

Second, in some business areas involving fault-tolerant, distributed real-time computations, there was a concern that the added value of a company not only lies in its know-how of the business domain (the PIM) but also in the design know-how needed to make these systems work in the field (i.e. the transformation to go from PIM to PSM). In some cases, the transformation was much more complex than the PIM. A lot of knowledge was actually captured in such a transformation, but it was done at the wrong level of abstraction because there was very little support for a proper separation of concerns in transformation languages such as QVT. One possible approach would have been to actually fulfill the Y-shaped approach presented in Figure 2, with an explicit PDM and a much simpler model transformation which would "just" implement a model composition between the PIM and the PDM to get the PDM. However no one ever managed to model a non trivial platform in such a way that it could be use in that purpose.

Finally, we can see that the effort to build dedicated transformation languages such as QVT or ATL has been a dead end. These language never really captured a wide audience. Their most advanced features (such as semi-declarative style) were at best difficult to use, while their navigation capabilities and higher order features are now available in general-purpose languages such as Java or Kotlin, leaving very little interest in writing new model transformations with QVT-style languages.

4 Separation of Concerns with Models, Aspects and Features

4.1 The Idea

As discussed in the previous section, MDA had the goal to separate the fundamental logic behind a specification from the specifics of the particular middleware that implements it. Platform dependency is indeed one important concern to separate from the core functionality of an application. But that is not the only one. In any complex application, there are many other concerns that must be dealt with, including performances, fault tolerance, concurrency, distribution, data persistency, energy consumption, safety, security, user experience, and many more.

The term *separation of concerns* has probably been coined by Dijkstra as far back as 1974 [9]:

> *Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study*

> *it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained —on the contrary!— by tackling these various aspects simultaneously. It is what I sometimes have called* **"the separation of concerns"***, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.*

As Dijkstra already pointed out, separation of concerns naturally leads to the idea of aspects that can be analyzed separately, which is precisely what we are trying to achieve with modeling. My preferred definition of a model is indeed [21]:

> *A model is the abstraction of an aspect of reality for handling a given concern.*

Note that the Aspect Oriented Programming community had a much narrower definition of an aspect as being only the modularization of a cross-cutting concern [12]. If we indeed have an already existing "main" decomposition paradigm (such as object orientation), there are many classes of concerns for which a clear allocation into modules is not possible (hence the name "cross-cutting"). Examples include both allocating responsibility for providing certain kinds of functionality (such as logging) in a cohesive, loosely coupled fashion, as well as handling many non-functional requirements that are inherently cross-cutting, e.g.; security, resource management, etc. But purely outside the programming world [39], there was an acceptance for a wider definition where an aspect is a concern that can be modularized. In that sense an aspect becomes quite close to the concept of a feature, as in Feature Oriented Programming [1].

Indeed, once we have identified and modularized our concerns as model level aspects [42], we still need to manage which variant of which concern goes into which product. This led to the development of the Software Product Line concept [37], and in particular to the pervasive feature diagram notation, as illustrated in Figure 3, to describe and explicitly manage variability[2].

4.2 The Good

So really modeling is the activity of separating concerns in the problem domain, an activity often called *analysis*. If solutions to these concerns could be described

[2]Taken from https://commons.wikimedia.org/w/index.php?curid=25197577

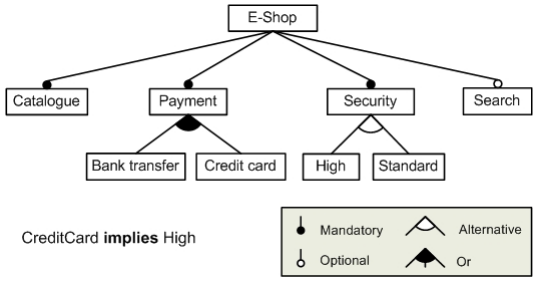

Figure 3: A Feature Model Example

as aspects, the design process would then be characterized as a weaving of these aspects into a detailed design model [16]. This is actually what software designers have been actually doing forever. Most often however, the various aspects were not *explicit*, or when there were, it was in the form of informal descriptions. So the task of the designer was to do the weaving in her head more or less at once, and then produce the resulting detailed design as a big tangled program (even if one decomposition paradigm, such as functional or object-oriented, is used). While it could work pretty well for small problems, it is a major headache for bigger ones.

Note that the real challenge here is not on how to design the system to take a particular aspect into account: there is a huge design know-how in industry for that, often captured in the form of design patterns. Taking into account more than one aspect as the same time is a little bit more tricky, but many large scale successful projects in industry are there to show us that engineers do ultimately manage to sort it out (most of the time).

The real challenge in a product-line context is that the engineer wants to be able to change her mind on which version of which variant of any particular aspect she wants in the system. And she wants to do it cheaply, quickly and safely. For that, redoing by hand the tedious weaving of every aspect is not an option.

In this context, *Model Driven Engineering* (MDE) can be seen as a methodology for mechanizing the process experienced designers follow by hand [43]. The idea is that when a new product has to be derived from the product-line, we can automatically replay the design process, just changing a few things here and there [33].

MDE technologies, such as executable meta-modeling [35], support development of models that capture software functionality and properties at different levels of abstraction and from different perspectives, as well as rigorous analysis of models, and transformation of models into software artifacts that serve specific development purposes.

4.3 The Bad and the Ugly

Cleanly separating concerns of a system is not always completely straightforward, but its difficulty is more or less proportional to the inherent complexity of the problem at hand. But once concerns have been separated into aspects, even simple ones as in the AOP view (i.e. oblivious and cross-cutting), one still needs to compose them.

While weaving a single aspect is pretty straightforward, weaving a second one at the same join point is indeed another story. When a second aspect has to be woven, the initial join point might not any longer exist because it could have been modified by the first aspect advice. If we want to allow aspect weaving on a pair-wise basis, we must then define the join point matching mechanism in a way that considers these composability issues. However, with this new way of specifying join points, two new problems arise (see [21] for a detailed explanation with the example of sequence diagrams):

- It is in general difficult (or even statically undecidable [4, 25]) to identify join points when the patterns we are looking for are based on the properties of the computational flow.
- The composition of the advice with the detected part cannot any longer be just a replacement of the detected part by the advice: we also have to define more sophisticated compositions operators.

More generally, the problem is that nobody ever came up with composition operators with good mathematical properties such as commutativity and associativity. That makes the tool support for these approaches extremely difficult and costly to build.

It probably means that there is no hope for a fully general-purpose, meta-model independent, model-level aspect weaver. Still, it should be possible to develop aspect weaving software components handling several aspects of aspect weaving, from general-purpose model-level pattern matching [38] to automated support for composing models written in a particular language (through a definition of model composition behavior in the metamodel defining the language [14]), to specializable model composers [17]. These aspect weaving components could then be customized and combined to build domain specific or even project specific aspect weavers.

5 Domain Specific languages and Software Language Engineering

5.1 The Idea

The principle of *separation of concerns*, and its derivatives, *Modularity* and *Abstraction* are the keys toward mastering the different dimensions of complexity of software, including its inherent variability. But as discussed above, it is extremely difficult to come up with powerful general-purpose composition mechanisms working at the level of software modules (functions, classes, aspects, components, etc.).

One very powerful idea [34] is to lift the composition at the language (or metamodel) level. Each concern of a complex software could then be described using its own relevant Domain Specific language, and their composition worked out one and for all by defining how those DSL could be composed.

The idea of using DSL is far from being a new one. As far back as 1972, Dijkstra (again!) said in his ACM Turing Lecture:

> *Another lesson we should have learned from the recent past is that the development of 'richer' or 'more powerful' programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally.*
>
> *I see a great future for very systematic and very modest programming languages.*

This quote is thus often cited by proponents of Domain Specific Languages (DSL), which are indeed modest languages specifically designed for a single purpose. Such languages provide developers with abstractions that are directly related to the application-specific concerns they are addressing in their software projects. For example, there are modeling languages for expressing security, distributed real-time systems, and user interface concerns, in addition to languages that are specific to e.g. the automotive software domain.

These DSLs tend to be smaller, focus on a particular domain, serve a smaller user community, and evolve at a faster rate than traditional modeling or programming languages, since the concern domains are continually evolving. The engineering of these languages thus presents special challenges when compared with the development of traditional languages. The MDE community [6] has developed a rich integrated collection of generative technologies that are based on standardized metamodeling facilities such as OMG's MetaObject Facility (MOF) [36]. These technologies use metamodels to facilitate and drive the generation of parsers, com-

pilers, code generators, checkers, simulators, and other integrated development environment services.

The need for proper tools and methods in the development of software languages recently led to the emergence of the Software Language Engineering (SLE) research field which is defined as "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages" [26].

SLE makes it possible to isolate the variability coming from the problem domain in the sense that domain experts are given DSL for them to directly express their requirements and change them as often as they wish.

5.2 The Good

New DSLs can nowadays easily be developed using a language workbench [13], a "one-stop shop" for the definition of languages and their environments [41]. The main intent of language workbenches is to provide a unified environment to assist both language designers and users in, respectively, creating new DSLs and using them.

The idea of using multiple DSL to solve a problem can even be applied to itself, as exemplified with the Kermeta Language Workbench [22] which is implemented as a Mashup of Meta-Languages. Modern language workbenches such as Xtext [10], Spoofax [24] or Monticore [27] thus typically offer a set of meta-languages that language designers use to express each of the implementation concerns of a DSL, along with tools and methods for manipulating their specifications.

One of the current trend in SLE is to consider more and more languages as first-class entities that can be extended, composed, and manipulated as a whole. Melange [8] is such a meta-language for modular and reusable development of DSLs. Leveraging the model type theory [40], Melange provides an algebra of operators for extending, restricting, and assembling separate DSL artifacts (see Figure 4).

Beyond helping the development of a single DSL, a new challenge in SLE is also to co-develop a set of related DSLs, each aiming at capturing a specific aspect of the same system, but seldom in a fully orthogonal way. For example, a system engineer may need to analyze a system property that requires information scattered in models expressed in different DSLs. Supporting this coordinated use of multiple DSLs leads to what we called the globalization of modeling languages [5], by analogy with world globalization in which relationships are established between sovereign countries to regulate interactions (e.g., travel and commerce related interactions) while preserving each country's independent existence.

Globalized DSLs aim to support several critical aspects of developing complex

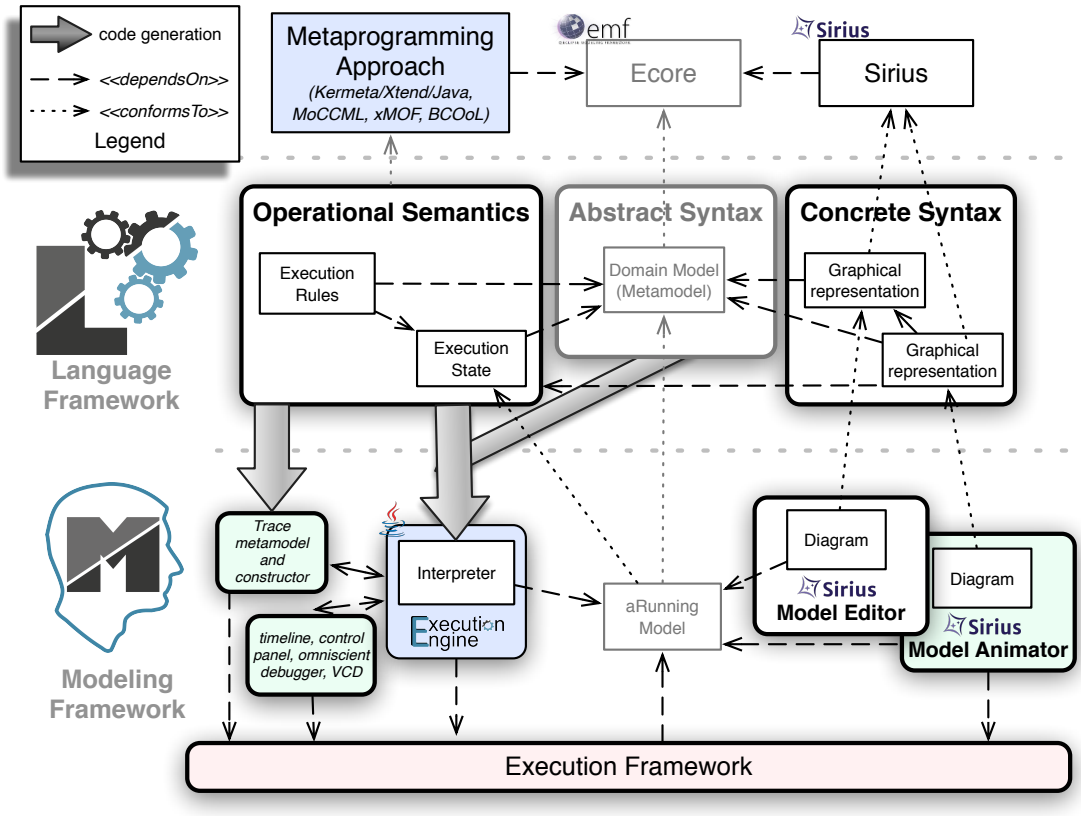

Figure 4: The SLE Vision as implemented in Melange

systems: communication across teams working on different aspects, coordination of work across the teams, and control of the teams to ensure product quality. The way different DSLs can be related ranges from simple interoperability, to collaboration (e.g. the same "event" could appear in a class diagram, a sequence diagram and a statechart), and finally composition (e.g., weaving and merging). The later is made possible when the builder of the DSLs has a full control over their semantics (and assuming they should be much simpler than the semantics of a general-purpose language such as Java) so that she can define at language level how they could be composed together [29].

The GEMOC Studio[3] is an example of a workbench supporting the globalization of modeling languages. It provides generic components through Eclipse technologies for the development, integration, and use of heterogeneous executable modeling languages. This includes, among others:

- metaprogramming approaches and associated execution engines to design and execute the behavioral semantics of executable modeling languages,
- efficient and domain-specific execution trace management services,
- model animation services,
- advanced debugging facilities such as forward and backward debugging (i.e. omniscient debugging), timeline, etc.
- coordination facilities to support concurrent and coordinated execution of heterogeneous models
- an extensible framework for easily adding new execution engines and runtime services

5.3 The Bad and the Ugly

"Software languages are software too" [11] and, consequently, they inherit all the complexity of software development in terms of maintenance, evolution, user experience, etc. Not only do languages require traditional software development skills, but they also require specialized knowledge for conducting the development of complex artifacts such as grammars, metamodels, interpreters, or type systems. While this is still far from being an easy endeavor, there is already some progress compared to the past. Indeed, during the 60's–70's you really had to be a genius to fully develop a new language. During the late 80's–90's, it became feasible for

[3] http://gemoc.org/studio.html

mere PhDs. In this millennium, it is usually taught at master level, or even before. Maybe tomorrow anybody will be able to hack her own language.

With respect to the globalization of DSLs, many challenging issues remain. Relationships among the languages will need to be explicitly defined in a form that corresponding tools can use to realize the desired interactions. Requirements for tool manipulation are thus another topic that will be a focus for future work in the area of DSL globalization.

6 A New Challenge: Models & Data

In all the previous sections, the models we have been discussing were explicit models that engineers were supposed to have created at some point. These engineering models are devoted to support the definition and representation of a targeted system [28]. Engineering models are *prescriptive* during the design process of a system, and become *descriptive* once the system is built.

But engineering models are not the only kind of models that are useful nowadays. Scientific models are for instance representations of some aspects of a phenomenon of the real world [15]. Based on established scientific knowledge, they are primarily used to explain and analyze a phenomenon of interest: they are first and foremost *descriptive* models. These models are validated or rejected by experiments, which play an analogous role as tests for engineering models. But once validated, these models can also become *predictive*. They are indeed typically used to predict future behavior of the phenomenon of interest through computer ran simulations, which make them somehow become engineering models.

Recent advances in computing power has also enabled a new spring for machine learning (ML) techniques, fostering the possibility to automatically infer models from data instead of laboriously crafting them. ML models are obtained through inductive reasoning principle, i.e., generalization from specific cases. ML models can be *descriptive* of a current or past relationship, *predictive* when given some hypothetical input data, or *prescriptive* if they are used in a larger system to make decisions.

Of course, it would be ridiculous to use machine learning to build models from well-known domains: we already quite well know the law of gravity, or what the model of a bank account is. We do not need to learn them from observations, i.e. apples falling from trees or sets of banking transactions. We however see it as an interesting challenge of our time how we can smoothly combine these three types of models (engineering models, scientific models and ML models) to build innovative systems. In particular, the kind of variability we must deal with here is the variability of data. Indeed, these models cannot be fully known before the data

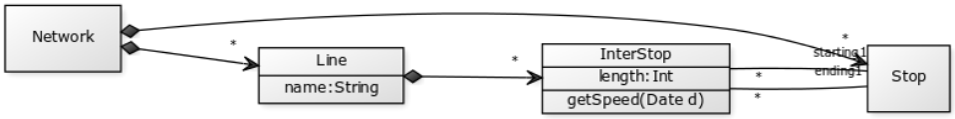

Figure 5: A Simplified Bus Network Model

is acquired, either for calibration (case of scientific models) or synthesis (case of machine learning).

A possible direction for integrating these models would be to articulate them around the concept of time, i.e. Past, Present, Future. Let's take the simplified example of a bus network, as illustrated in Figure 5, and concentrate on the $getSpeed()$ method of the class InterStop, that yields the speed of a bus between two adjacent stops at a specific date d.

- Conceptually speaking, if d is in the past, the $getSpeed()$ method is a simple lookup of the relevant data in the database of recorded bus timetables that would have been designed as a part of the engineering model for the bus management application.
- If d is *now*, then it means that to get the speed of a bus on a particular InterStop, I should have access to some kind of model reflecting the current state of the bus system, aka model at runtime.
- If d is in the future, then I have to find a way to accurately predict the future. Of course predictions are always difficult[4], but there exist several techniques to help us do that. First we can use an explicit (scientific) model bringing some mathematical approximation of the distribution of bus speeds across the city, using for instance some linear regression of predefined parameters such as time of the day, road quality, number of pedestrian crossings etc. After some calibration with past data, we might or might not get accurate predictions over the future. Alternatively we can also try to learn bus speeds out of raw data using one of the myriads of Machine Learning techniques we now have at our disposal, with all the well-known caveat of these techniques.

What is interesting here is that to predict the future, the model is no longer fixed but needs to be continuously changed to incorporate new knowledge stemming out of new data. As engineers we now have to cope with a brand new dimension of variability: the variability brought by the continuous flow of new data entering our system.

[4]Especially when they concern the future (Woody Allen).

We are currently exploring these ideas in the context of modeling the public transportation infrastructure of the Rennes Metropolis, with the hope of providing an integrated tool supporting past, present and future views of what's going on there, including support for what-if scenarios (e.g.; what if road works happen here and here and bus have to be deviated, or what if we change the itinerary of that line etc.) [30].

Many other groups are working along the same lines around the world [7], and there is no doubt that this opens the way for a renewed interest in software and system modeling.

Acknowledgement I would like to thank all the members of the DiverSE team at Inria/IRISA for fruitful discussions on the topic of this paper and for their feed-back on its early versions.

References

[1] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering*, pages 125–140, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[2] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In *Proceedings of UML 2003*, San Francisco, United States, Oct. 2003.

[3] F. P. Brooks. *The Mythical Man-Month*, volume 1995. Addison-Wesley Reading, MA, 1975.

[4] W. Cazzola, J.-M. Jézéquel, and A. Rashid. Semantic join point models: Motivations, notions and requirements. In *SPLAT 2006 (Software Engineering Properties of Languages and Aspect Technologies)*, Mar. 2006.

[5] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, 2014.

[6] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. R. Steel, and D. Vojtisek. *Engineering Modeling Languages*. Chapman and Hall/CRC, Nov. 2016.

[7] B. Combemale, J. Kienzle, G. Mussbacher, H. Ali, D. Amyot, M. Bagherzadeh, E. Batot, N. Bencomo, B. Benni, J.-M. Bruel, J. Cabot, B. H. C. Cheng,

P. Collet, G. Engels, R. Heinrich, J.-M. Jézéquel, A. Koziolek, S. Mosser, R. Reussner, H. Sahraoui, R. Saini, J. Sallou, S. Stinckwich, E. Syriani, and M. Wimmer. A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems. *IEEE Software*, page 9, 2020.

[8] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 25–36. ACM, 2015.

[9] E. W. Dijkstra. *On the Role of Scientific Thought*, pages 60–66. Springer New York, New York, NY, 1982.

[10] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[11] J.-M. Favre. Languages evolve too! changing the software time scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 33–42. IEEE, 2005.

[12] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Oct. 2000.

[13] M. Fowler. Language workbenches: The killer-app for domain specific languages, 2005.

[14] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, Annapolis, MD, USA, 2007.

[15] P. Gerlee and T. e. a. Lundh. *Scientific Models*. Springer, 2016.

[16] W. M. Ho, J.-M. Jézéquel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of 1st ACM International Conference on Aspect Oriented Software Development, AOSD 2002*, Enschede, The Netherlands, Apr. 2002.

[17] A. Jackson, O. Barais, J.-M. Jézéquel, and S. Clarke. Toward a generic and extensible merge operator. In *Models and Aspects workshop, at ECOOP 2006*, Nantes, France, July 2006.

[18] J. Jézéquel. Reifying configuration management for object-oriented software. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998*, pages 240–249. IEEE Computer Society, 1998.

[19] J. Jézéquel, D. Deveaux, and Y. L. Traon. Reliable objects: Lightweight testing for OO languages. *IEEE Softw.*, 18(4):76–83, 2001.

[20] J.-M. Jézéquel. Experience in validating protocol integration using Estelle. In *Proc. of the Third International Conference on Formal Description Techniques, Madrid, Spain*, November 1990.

[21] J.-M. Jézéquel. Model Driven Design and Aspect Weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, 2008.

[22] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Software and Systems Modeling*, 14(2):905–920, 2015.

[23] J.-M. Jézéquel and C. Jard. A multi-processor Estelle to C compiler to experiment distributed algorithms on parallel machines. In *9th IFIP International Workshop on Protocol Specification, Testing and Verification, University of Twente*, The Netherlands, Netherlands, 1989.

[24] L. C. Kats and E. Visser. *The spoofax language workbench: rules for declarative specification of languages and IDEs*, volume 45. ACM, 2010.

[25] J. Klein, L. Hélouet, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, Mar. 2006. ACM.

[26] A. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[27] H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.

[28] E. A. Lee. Modeling in engineering and science. *Commun. ACM*, 62(1):35–36, Dec. 2018.

[29] M. Look, A. Navarro Pérez, J. O. Ringert, B. Rumpe, and A. Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical

Systems. In *Globalization of Modeling Languages Workshop (GEMOC'13)*, volume 1102 of *CEUR Workshop Proceedings*, 2013.

[30] G. Lyan, J.-M. Jézéquel, D. Gross-Amblard, and B. Combemale. DataTime: a Framework to smoothly Integrate Past, Present and Future into Models. In *MODELS 2021 - ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems*, pages 1–11, Fukuoka, Japan, Oct. 2021.

[31] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, Oct. 1992.

[32] J. Miller and J. Mukerji. Model driven architecture. https://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, 7 2001.

[33] B. Morin, O. Barais, and J.-M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *2nd International Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, Germany, 2008.

[34] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J. Jézéquel. Weaving variability into domain metamodels. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, volume 5795 of *Lecture Notes in Computer Science*, pages 690–705. Springer, 2009.

[35] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML'2005*, Montego Bay, Jamaica, Oct. 2005.

[36] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification Version 2.5. OMG Document No. formal/2015-06-05, http://www.omg.org/spec/MOF/2.5/, June 2015.

[37] K. Pohl, G. Böckle, and F. van der Linden, editors. *Software Product Line Engineering*. Springer, 2006.

[38] R. Ramos, O. Barais, and J.-M. Jézéquel. Matching model-snippets. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA, October 2007.

[39] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD*, pages 11–20. ACM, 2003.

[40] J. Steel and J. M. Jézéquel. On model typing. *SoSyM*, 6(4):401–413, 2007.

[41] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, and G. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proc. SPLASH*, pages 95–111, 2014.

[42] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML profile for software product lines. In *Proceedings of the Fifth International Workshop on Product Familly Engineering (PFE-5)*, RENNES, France, 2003.

[43] T. Ziadi and J.-M. Jézéquel. *Product Line Engineering with the UML: Deriving Products*, pages 557–586. Number ISBN: 978-3-540-33252-7. Springer Verlag, 2006.