Are Models the DNA of Software Construction?
A Controversial Discussion

Friedrich Steimann
Universität Hannover
Institut für Informationsysteme
Fachgebiet Wissenbasierte Systeme
Appelstraße 4, D-30167 Hannover
steimann@acm.org

Thomas Kühne
Technische Universität Darmstadt
FG Metamodeling
Hochschulstr. 10
64289 Darmstadt
kuehne@informatik.tu-darmstadt.de

INTRODUCTION

MDA is advocated as the next step in software construction. It builds on the availability of a powerful modelling language and model compilers that translate models into executable code. In the following Dr. Con—known as a harsh critic of UML and as a sceptic of the feasibility of MDA—and Dr. Pro—a believer in the MDA vision—are discussing whether MDA is fundamentally flawed from the beginning or represents the most promising new development paradigm we work on today.

GRAPHICAL VS. TEXTUAL: WHEN DOES THE CHOICE MATTER?

Dr. Con: MDA starts from a weak basis—graphical models—and aims at the highest possible goal: the delivery of sound production level code. We know that in theory, graphical and textual notations are equivalent (at least as regards their expressive power), but in practice graphical notations are usually far more cumbersome than textual ones. Admittedly, they aid comprehension if kept simple, but they quickly get convoluted when it comes down to the core of real problems (not some very high abstraction thereof). Why should something as weakly developed as a graphical modelling language help solve a problem all other approaches have failed to?

Dr. Pro: To do justice to this important legibility problem one needs to distinguish *structural* from *behavioural* information. Graphical models excel in conveying information of the former kind, while admittedly have been known to be challenged for the latter. I think we can agree on the fact that a visual rendering of structural information enables quicker reception of information compared to sequential parsing of text where, e.g., one-to-many relationships, have to be expressed and then re-recognized through repeatedly referring to the same identifier (the "one" case). In this sense, indeed "one picture says more than a 1000 words". With respect to behavioural information the situation is not that clear cut, yet not a lost case for graphical approaches either. While there is certainly no point in having an iconic programming language mimicking all the traditional features of today's programming languages, surely there is value in presenting and expressing behaviour in the form of interaction diagrams (sequence and collaboration types) and state diagrams. When it gets down to the nitty-gritty, small grained behaviour, one may still use the more high-level diagrams as navigation aids to get to textual behaviour description and/or refrain from descending to such fine grained behaviour at all. In a source model we will mainly just select predefined behaviour, but not spell it out. Transformation engineers will still need to get their hands dirty here and there, specifying which code needs to be generated for given high-level model elements/annotations. The modeller, however, may "stay graphical" at all times.

Dr. Con: If graphical syntax were the better form of expressing programs, we would long be using it. Instead, what we find is people using degenerate state machines (degenerate to the extent that all transitions to one state are labelled with the same event so that they are actually flow charts without loops and subroutine calls), decorating states with so-called actions that are more or less SMALLTALK blocks in disguise. This is really one step back from structured programming.

Dr. Pro: It has not been that long ago when all we had to support programming was ASCII editors. It is quite understandable that something like "graphical programming" will take its time to receive optimal support and widespread adoption. Many organizations all over the world are already using "graphical programming", i.e., various MDA techniques, improving their productivity. The examples you refer to are first steps towards the MDA vision, which should be judged by its true potential, not its incarnations during its infancy.

Dr. Con: But there is an intrinsic problem with graphical approaches: Either the models are simple and readable but then only capture functionality too trivial to be useful, or they faithfully capture complex functionality but then are next to illegible.

Dr. Pro: I already pointed out that—given powerful transformations, which inject a lot of low-level behaviour code—

an actual MDA modeller wishes, and is able, to stay at a high level of abstraction. Such models are both legible and rich in terms of specifying functionality. The functionality might only be expressed as a choice from a fixed set of available low-level behaviours, but nevertheless the information is there. This shift—from "programming" to "configuration", i.e., from "hand-crafting a single piece of software" to "selecting a fitting piece from an existing choice"—is a persistent trend in the history of computer science when it comes to increasing the productivity of software creation.

Dr. Con: What if your desired choice is not among the choices offered? Then you are stuck.

Dr. Pro: The modeller will be stuck, i.e., not be able to continue on his own. Still, after the transformation engineer has provided any missing choice, the modeller may continue. It follows that MDA implies new development roles and opens up the way for new development paradigms.

Dr. Con: So it all comes down to a very high level language that—no matter whether graphical or textual—can automatically be transformed to some lower-level, platform-specific specification?

Dr. Pro: That's right. The usual emphasis on *graphical* models is not core to the approach. Essential is the idea to use high-level descriptions which are solution independent, or in MDA speak "platform independent". Ideally, the high-level descriptions should resemble *analysis* models rather than more solution oriented *design* models.

SOUNDS FAMILIAR: WHY SHOULD IT WORK THIS TIME AROUND?

Dr. Con: MDA is not the first go at making software construction more productive; code generation tools and forth generation languages (4GLs) for instance have been around for quite some time. But practice has shown that code generation works poorly. If it worked well, we would make the source of the code generator a new programming language and turn the generator into a native compiler. But it isn't like that. We need to make changes to the generated code, partly because it is just one bit off what we needed (and what we need precisely just cannot be generated), partly because there is code that is for good reasons not generated (an algorithm for instance), simply because the target language is much better suited to express it than the source formalism of the generator. No one would accept a programming language that, in order to produce useful programs, would require hand coding in assembler as an additional exercise.

Dr. Pro: There are applications of code generation which tremendously increase the productivity of the people employing it. It is just a matter of time before this kind of development will be more or less commonplace. Point taken, we still have to learn a number of lessons until we know how to use the generation paradigm optimally. That is why some of the issues you address are relevant, yet they are not insurmountable. In the future we'll learn to make use of, e.g., parameterization, to tweak our transformations so that they have *exactly* the desired effect and do not just approximate it. We will also learn how to best balance the distribution of behaviour in terms of expressing it in the model and/or shifting it to the transformations. For one thing, we won't generate algorithms from scratch. We will either just select existing ones, or compose new ones from building blocks that are known to work to together. For instance, many sorting algorithms can be expressed by a very general approach which is just parameterized with two reduction strategies. "Tournament Sort" for instance was found by a certain combination of reduction strategies[1]. With respect to your last sentiment, I fully agree. This is not an argument against MDA, though, since the modeller will never have to get in touch with "assembler". It is the transformation engineer who might.

Dr. Con: History has shown that the kind of 4GL you seem to aspire for didn't work too well. In my view, we had the perfect 4GL (it was called SMALLTALK), but as everyone knows we are all doing JAVA now and find ourselves writing most stereotypical source code most of our time. Try to find a modelling language that is both more expressive and more productive than SMALLTALK, then we can start talking.

Dr. Pro: The strength of SMALLTALK was the "as small a language as possible" and "as rich a library as needed" approach. Relying basically just on assignment and message-send as two primitives, everything else was deferred to the SMALLTALK library. This way one could extend the language in almost arbitrary ways. This is what MDA is about as well. You may extend your modelling language—and thus your expressiveness—in arbitrary ways, if you have a mechanism of defining what the new concepts mean. In SMALLTALK you define a method to elaborate what, e.g., "while" means. In MDA you define a new transformation (or add an aspect to an existing one), defining what the new concept means. To use your words, we have already "started talking"! By the way, what better indication for the viability of MDA could there be than the fact that using JAVA we, "write most stereotypical source code most of our time"?!

COMPILING MODELS VS. COMPILING PROGRAMS: WHAT DIFFERENCE DOES IT MAKE?

Dr. Con: Are you not mixing up two different kinds of transformation? One kind of transformation goes from platform-independent to platform-specific. The other kind of transformation goes from high-level to low-level description. The former adds no real information, whereas the lat-

[1] A. Kershenbaum, D. Musser, A. Stepanov, *Higher Order Imperative Programming*, Rensselaer Polytechnic Institute Computer Science Department, 1988.

ter has to invent something (make informed guesses) and thus is a truly creative process. In fact, I would conjecture that PIM to PSM transformation is very much like applying a native compiler to source code or, in other words, that the PIM could run on a virtual machine and the PSM is its equivalent platform-specific executable.

Dr. Pro: Superficially the two kinds of transformations appear to be different, however, from a MDA perspective they can be unified. First, when a compiler translates source code to byte or machine code, there are choices to be made (i.e., information to add) as well. Yet these choices, e.g., how to realize a procedure call are so clear cut that the compiler can make them for you without your advice. If the translation starts from something more complex than today's source code, there are more options and choices to make, e.g., in which particular way you would prefer a "Dictionary" data structure to be realized. Once a choice is made though, there is no "creativity" involved. Thus, the two kinds of transformation are really not that different: the code templates and translation rules embodied in a compiler basically capture how to transform a high-level description into something executable given some supporting hardware. In comparison, starting from a problem based description (PIM) and gradually moving towards a more solution oriented one (PSM) represents the same process. I say "gradually" since from the original PIM to the final PSM it will usually take multiple transformation steps were two subsequent models in the transformation chain assume the roles of PIM and PSM respectively, so „platform dependence" really is a relative property rather than an absolute one. A platform is essentially a "realization infrastructure" (just like hardware), hence the further you approach a certain realization support, the more solution oriented and low-level you get. In both cases, a) you increase the specificity of platform dependence and b) you add more low-level detail.

Dr. Con: But how does a model compiler know how to realize a "Dictionary" data structure? I may express a choice, but what is the magic that implements my choice? Using programming languages I have to spell out how a specific Dictionary works. Why can I skip that step with models?

Dr. Pro: Transformations have to add something that is directed by the models, but not contained in them. Adding this information is what you referred to as a "creative process", but really it's the transformation engineer who needs to be creative once, and then the modeller may just indicate which solution choice is appropriate. In other words, "engineer the transformation once, apply it everywhere".

FROM ABSTRACT TO CONCRETE: HOW TO CLOSE THE SEMANTIC GAP?

Dr. Con: How much redundancy is in a program? Given that the most trivial bug can alter program behaviour to an extent that makes it useless, I would assume not very much. Now given that almost nothing of a production level program can be omitted or changed without violating the specification, the same information content must be captured by the model. This would imply that either

1) the modelling language is exceedingly more expressive than any programming language known today, or that
2) models are (nearly) as complicated as the programs they produce, or that
3) only few programs can be generated from models.

Assuming that the first is not the case and that the second is not what we want, we must conclude the third. In other words: if a model is significantly simpler than the program it produces, and if redundancy in a program is low, then I would assume that there are far fewer meaningful models than meaningful programs, meaning that there are many useful programs that cannot be generated. After all, programming is a complex matter not because our linguistic means are inappropriate, but because our problems are extremely complex. So how can we expect to be able to take the complexity out without losing precision?

Dr. Pro: You are right in observing that there is no noise in programs. We cannot simply remove or alter pieces of programs and still expect them to work. However, there is redundancy in that many high level concepts (say n-ary associations) are repeatedly realized using lower level features (say references) in the same way. Instead of manually creating those repeated patterns of code, we should just specify that we want that particular pattern applied to a number of occasions. In that sense, a modelling language drawing from many such predefined realization patterns, which are enacted by transformations, is indeed "exceedingly more expressive than any programming language known today". That is why your second implication does not hold in the case of MDA. Granted, during its infancy MDA will have to live with your third implication that "only few programs can be generated from models". But even initially "few" will equal "very many" and gradually one will learn what the boiler-plate code of the currently too complex applications is and will then be able to capture it through standard transformations guided by model annotations. Let me give you an example: when hand-coding machine code you may use various different ways doing subroutine calls (with various protocols for using the registers, stack, etc.). You may think that calling a subroutine is a challenging task that needs to be creatively resolved on a case by case basis. Soon, however, you will learn that you can reduce all your variety to perhaps two basic cases (e.g., synchronous and asynchronous) and then it is enough to signify which case you need and generate all the low-level actions required in each case.

Dr. Con: I don't think you got my point. For any given number of function points, there are far more different programs than there are different models. Given that the transformation of a model to a program leaves the number of function points unaltered, mapping is either ambiguous, or not onto. This turning of knobs you mention (the choice or parameterization of transformations) should either not alter the functionality, or should be part of the specification (and hence also be found in the model).

Dr. Pro: The very fact that the mapping from a model to a program is ambiguous gives the model its value! It can express the same number of function points without getting bogged down in realization choices. Spelling out all the realization choices a) makes it difficult to see the functionality for all the realization of it and b) can be platform dependent and hence subject to change. The actual realization choices—removing the ambiguity—are partly expressed within the model, using a so called "marking model", and are partly expressed outside the model as "turning of knobs" choices regarding the transformation. You may regard the latter as "compiler options". Today's compilers also allow some choice of translation strategy, e.g., optimize for code length or execution speed. Model compiler will feature a lot more of these "knobs" since much more solution specific choices need to be made. In summary, you may create *all* your different programs from a *combination* of one model and various transformation choices.

A TRANSFORMATION-BASED APPROACH: DOES IT REALLY MAKE SENSE?

Dr. Con: So what you propose is really an MDK (Model Development Kit) analogous to the JDK or .NET, and a "glue" language that allows the model engineer to put it all together. Do you have any other examples than 1:n-relationships where this could work? I could imagine integrating collection-valued fields into any programming language!

Dr. Pro: If you want to call it "MDK" that's fine with me as long as you don't confuse the approach with a simple "library usage" paradigm. MDA is not about including prefabricated parts, it is about applying boiler-plate realization strategies automatically. Sure you may add "collection-valued fields" to your favourite programming language, but where do you stop adding features? MDA is about using a core language whose expressiveness can be extended, e.g., by using annotations (marks) and associated realization strategies. This way you are neither stuck with some level of language expressiveness nor do you end up with a never ending featurism for a given language. By the way, examples over and above 1:n-relationships are numerous. A few typical exemplars are events, which need to be realized with messages, distribution, often calling for hand written transaction schemes, and persistence, also typically requiring the mechanical addition of regular code fragments to domain classes.

Dr. Con: It seems you are advocating an extensible library of modelling constructs (various kinds of classifiers, associations, calls, state transitions, etc.) each coming with a set of transformations the modeller can pick from. The transformations are all functionally equivalent, but differ in a) the target platform and b) non-functional properties such as efficiency etc. Whenever I (as the modeller) miss a) a modelling construct or b) a transformation rule, I turn to a transformation engineer to have it manufactured for me. Of course, all elements of the library are designed to go together well, i.e., they can be combined freely.

Dr. Pro: You've got it!

Dr. Con: Is UML such a library? And if so, what are the modelling language primitives, the analogues of SMALLTALK's assignment and message passing? Although we finished the "graphical vs. textual" debate earlier on, it strikes me that especially assignment, cannot be efficiently expressed graphically. You will agree that "*new*" and "*delete*" constraints on links are really cumbersome. How can that lack be made up for?

Dr. Pro: The UML certainly can be used for MDA purposes, which is not to say that it is already optimally equipped for this purpose. With respect to your second question, let us not repeat the discussion we had earlier on concerning the granularity of behaviour you want to express with your models. Again, even if the models were just structuring and complementing plain old programming code, MDA would be a huge step forward. To which extent behaviour is also expressed by models, e.g., with interaction diagrams, state charts, etc. and with which level of granularity is a discussion which is interesting but whose outcome certainly does not endanger the viability of the overall approach.

Dr. Con: I would assume that domain-level entities (classes such as *Account*, *Document*, or *PatientRecord*) are not part of the library?

Dr. Pro: Why not? If they are of general use across applications, surely it makes sense to support various realization choices through predefined transformations.

Dr. Con: Ok, what you say sounds nice but I still don't buy it. You are basically implying that programming is just about selecting a number of prefabricated behaviours and then combining them. This is not the case, though. Just recently I came across a very simple problem. From a document comprised of a list of paragraphs, all paragraphs starting with a given word had to be printed in their order of appearance. For two selected paragraphs that were non-consecutive in the original document, the first sentence of the first intermittent paragraph and the last sentence of the last intermittent paragraph also had to be printed to mark the omission. While the actual problem was in fact a little trickier than that, the chances that the solution even of its simplified version would be readily available in any library are very low, as are the chances that this kind of iterator is ever needed again (which is why it isn't in the library in the first place). My own work experience tells me that every non-foobar application comes with countless idiosyncratic problems of this kind, and I would suspect that all attempts to parameterize model elements or transformations so that they can cover every conceivable peculiarity to otherwise stereotypical patterns is doomed to failure.

Dr. Pro: Let me pick up your usage of the word "pattern" and rephrase your statement to "The idea of design patterns is silly. Every application is different and the reduction of a programmer's work to a standard set of design patterns is doomed to failure". Of course, the success of design patterns tells us that even though all applications are different and in fact often require very specific algorithms, there is still a large amount of regularity to be exploited (and, in fact, should be *enforced* as it is often better to use a well-proven design pattern solution than some custom ad-hoc realization strategy). Specific algorithms, such as the one you describe, need not necessarily go into a transformation library. They may also be expressed in the source model using the available ways of describing behaviour. This implies neither that there is no regularity in applications to exploit at all, nor that the algorithm should not be realized by a transformation even if it does not appear to be of general utility.

Dr. Con: I would conjecture that with a language like SMALLTALK an optimum has been reached in the trade-off between expressiveness and flexibility. As it turns out, only few of the languages ranked above level 20 in Capers Jones' famous language productivity list[2] could be considered general purpose. For instance, trying to formulate the above selection procedure as an SQL statement would certainly make me wish I had started the project in assembler.

Dr. Pro: I do agree that SQL would be a bad choice but then SQL is not UML and is not supported by predefined and extendable model transformations either.

FINAL SPEECHES: WHAT IS THE VERDICT?

Dr. Con: Graphical and other high-level languages are *only good* for people who cannot program. This is not because they are more expressive (in the sense that they are capable of expressing complex things in a simple manner), but because they oversimplify. In order to be able to make a useful program (one that meets its users' expectations) from a model, the model must be so complicated that

a) people who cannot program cannot understand it and
b) people who can program would rather write a program than draw the corresponding model, in part because fundamental notions such as assignment may only be poorly supported graphically.

Even though (over)simplification can be useful at times, it is certainly not a sufficient basis for creating a satisfactory end product.

Dr. Pro: Let's rephrase your first sentence to "graphical and other high-level languages are *ideal* for people who cannot program". And this is fantastic, since we do not want people to program anymore. We need to stop them wasting time and money by reproducing boiler-plate code and regular realization patterns, time and again; all this in an error-prone fashion, often resulting in suboptimal solutions. People who cannot program will still be able to understand a model specifying their desired solution, since it will be built on and rely on a multitude of known realization strategies. I do not need to know how to best implement a particular *n*-ary association. All I need is to know that I want it. People who can implement such and more complicated realizations will not be modelling but rather develop new transformations. While it is true that it is still on open issue how to optimally specify behaviour at the lowest level of granularity with graphical models, this certainly does not represent a stumbling block for the MDA vision as such. Abstraction—as opposed to (over)simplification—well used, is definitely useful and in fact, our only known basis to master the ever-growing demands on constructing complex software in more reliable and productive ways.

[2] *http://www.theadvisors.com/langcomparison.htm*