By permission of Cambridge University Press this paper is reproduced from the book **From semantics to Computer Science; Essays in Memory of Gilles Kahn**, *to be published early in 2009.*

The tower of informatic models

Robin Milner

University of Cambridge

Abstract: Software science has always dealt with models of computation that associate meaning with syntactical construction. The link between software science and software engineering has for many years been tenuous. A recent initiative, *model-driven engineering* (MDE), has begun to emphasize the role of models in software construction. Hitherto, the notions of 'model' entertained by software scientists and engineers have differed, the former emphasizing meaning and the latter emphasizing tool-based engineering practice. This essay finds the two approaches consistent, and proposes to integrate them in a framework that allows one model to *explain* another, in a sense that includes both implementation and validation.

This essay is dedicated in admiration to the memory of Gilles Kahn, a friend and guide for thirty-five years. I have been struck by the confidence and warmth expressed towards him by the many French colleagues whom he guided. As a non-Frenchman I can also testify that colleagues in other countries have felt the same.

I begin by recalling two events separated by thirty years; one private to him and me, one public in the UK. I met Gilles in Stanford University in 1972, when he was studying for the PhD degree—which, I came to believe, he found unnecessary to acquire. His study was, I think, thwarted by the misunderstanding of others. I was working on two different things: on computer-assisted reasoning in a logic of Dana Scott based upon domain theory, which inspired me, and on models of interaction—which I believed would grow steadily in importance (as indeed they have). There was hope to unite the two. Yet it was hard to relate domain theory to the non-determinism inherent in interactive processes. I remember, but not in detail, a discussion of this connection with Gilles. The main thing I remember is that he ignited. He had got the idea of the domain of streams which, developed jointly with David MacQueen, became one of the most famous papers in informatics; a model of deterministic processes linked by streams of data.

The public event, in 2002, was the launching workshop of the UK Exercise in Grand Challenges for Computing Research. It identified eight or so Grand Challenge topics that now act as a focus for collaborative research; part of their effect is to unite researchers who would otherwise never have communicated. Before the workshop we

had no doubt that Gilles Kahn was the one to invite as keynote speaker. We knew his unique combination of encouragement and probing criticism; just what we needed for the event. And so it turned out. His view of the future of computing, and his cautionary remarks about artificially created goals, are well-remembered. Equally important were his enthusiasm, and his encouragement to aim high.

Purpose

The purpose of this essay is to suggest, in simple terms, how to harmonise the scientific content of informatics with its engineering practice. Such an exposition should help informaticians[1] both to coordinate their work and to present it to other scientists and to the wider public. It should also clarify the nature of informatics alongside, but in contrast with, the natural sciences.

In attempting this general exposition, let us avoid terminology that is either new or technical. Of course, each instance of modelling—such as engineering a distributed system, or modelling by intelligent agents, or optimizing target code, or verifying a program—has its own technical terms; but the terms used for each are unlikely to cover all instances. And we should minimise the extra term-baggage used to fit these instances together into a whole, even if we use terms that are imprecise.

Models

All scientists and engineers will agree that they work with models and modelling. The word 'model' is both verb and noun. Used as a verb, 'M models R' usually means that M that represents (some aspects of) R, a reality. Used as a noun, 'M is a model' means that M may represent one or more unspecified realities; for example, differential equations model many realities. R and M may be singular: R a specific ship, e.g. the Queen Mary, and M a specific plastic replica of R. Or they may be plural; R a family of realities, such as national economies, and M a pack of concepts and equations that represent these economies. We shall avoid the singular kind of model, and say that a model comprises

A family of *entities*; and
What these entities *mean*.

This is not a formal definition; it is rather a challenge to seek a notion of model that unites the engineering and scientific aspects of informatics. The purpose of the 'definition' is to insist that a model should not merely enumerate its entities, as in a syntactic definition, but should describe how they work, i.e. their meaning. On this basis we shall propose how models should relate to each other.

The imprecise term 'meaning', synonymous with 'semantics' or 'interpretation', is meant to be inclusive. If the entities are automata or programs or processes, it includes their activity or behavior; if the entities are the sentences of a logic it includes the truth-valuation of those sentences via an interpretation of the function and predicate

[1] Loosely speaking, informatics is a synonym for computer science, and hence informatician (or informaticist) is a synonym for computer scientist. The 'info' words have an advantage: they express the insight that informatic behavior is wider than what computers do, or what computing is.

symbols[2]; if the entities are differential equations, it includes the interdependence of the variables and their derivatives. But the term 'meaning' also includes an informal description of how the entities work.

We avoid a sharp distinction between 'model' and 'reality'.[3] We may wish to say that a reality, say 'clouds and their movement', is an *extremal* model: it doesn't represent anything else. But some realities—e.g. a plastic replica of a ship—are models of other realities, so it is a mistake to say that *all* realities—considered as models—are extremal.

Our informal definition admits artificial realities as well as natural ones; thus it includes all engineered artifacts. In particular, it includes 'computers and what their screens display'; a model of it can then be 'assembly programs and their semantics'. Part of what contrasts engineering with natural science is that the realities are artificial in the first case and natural in the second.

Explanation

The phrase 'model *of* ...' needs discussion. The '*of*' relationship, between a model and the reality it explains, is central to the whole of science; the same relationship holds in any engineering discipline between a model and the reality it explains. Just as we say that Newton's laws *explain* the movement of bodies with mass, so we can say in informatics that a model consisting of programs and their meaning *explains* the reality of computers and what their screens display.

The artifacts of informatics are not always (physical) realities; they can also be (and are more often) syntactic or symbolic. In fact, they are models. What distinguishes the science of informatics is that its artifacts demand explanation at many levels. Consider Fortran, one of the most influential models in computing history. The model 'Fortran and its behavior' (even though this behavior was informally described) explains the model 'assembly programs and their behavior'; at least, it explains those assembly programs that are the translations of Fortran programs. By transitivity, Fortran also explains the real behavior of the computers on which these assembly programs run. As we shall see later, Fortran—and other symbolic constructions—demand explanation at still higher levels; we begin to see why the term 'tower of models' is appropriate. We shall also argue that the towers have breadth as well as height.

One may feel uneasy about this use of the term 'explanation', because realities normally precede the models that explain them. But informatics deals with artifacts, whether real or symbolic, so the explanation often precedes what its explains. One may then be happier to use 'specify' rather than 'explain'. But there are cases where the artifact precedes the explanation; if a reverse-engineer succeeds in reconstructing the lost Cobol source-code of a legacy assembly-code program, then she would find it

[2]Logics are models in the sense of this essay, so we should compare the way we use the term 'model' with the logicians' usage. The striking difference is that a logician speaks of models *of a logic*, and that we (so far) take a logic to be a model *of something else*. The logicians' usage is close to what we call 'meaning'. Thus the two usages differ but can be reconciled.

[3]In this paper, we use 'reality' to mean 'physical reality'. This is for brevity, rather than from a conviction that all reality is physical.

natural to call the former an explanation of the latter.[4]

We shall therefore assume that *explanation* is the principal relationship—with many manifestations—that cements the tower of models that we call informatics. Using near-synonyms we can express 'model A explains model B' in many ways; for example

> model A *represents*, or *specifies*, or *abstracts from*, model B; or
> model B *realises*, or *implements*, or *refines*, model A.

As a simple illustration, suppose that B is a programming language, whose behavior is defined in one of the usual ways. One way is by structured operational semantics (SOS), which is formal; another way is informal—a description in natural language of how the runtime state changes. An example of the latter is the original description of Algol60, an outstanding example of lucid prose.

Now let A be a specification logic, such as **Z**; its entities are sentences, and its meaning defines the interpretation of each sentence. Then an explanation of the language B by the logic A comprises—for each program P of B—a set S of A-sentences concerning the operations, types, variables, inputs and outputs of P. The explanation provides a way to prove that each sentence of S is satisfied by the behavior of P as described in B. If S pre-exists P then it may be called a specification of P. It is unlikely to determine P uniquely; the larger the set S, the more accurately is P determined.

Combination

The entities in a model need not be all of the same kind. Consider a model of the flight of informatically controlled aircraft. This heterogeneus model combines at least three parts: a model of the real world (locality, temperature, wind speed, ...) in which the planes fly; an electro-mechanical model of the systems to be controlled; and a specification (or explanation) of the controlling software. Consider also a model of humans interacting with a computer; the model of the human components may involve human attributes such as belief or sensation, as distinct from the way the computer is described. These two examples show the need not only to combine informatic models, but to combine them with others that are not informatic.

Such *combination* is best seen as a construction, not a relationship; it combines the entities of different models, with extra behavioral description of how they interact. Combinations abound in informatics. Further examples: hybrid systems mix differential equations with automata; coordination systems combine different programming languages via shared data structures; and a distributed programming language may be combined with a networking model to create a model of a pervasive system.

Towers

Let us declare a *tower of models* to be a collection of models built by combination and related by explanation. A tower may be tall and thin, or short and broad. Breadth

[4] We tend to use 'A explains B' as an abbreviation for 'A-entities explain B-entities'. This allows us to dodge the question of how many A-entities are involved in explaining each B-entity. This surely varies from case to case, but for this essay we shall use the abbreviated form 'A explains B' for all cases.

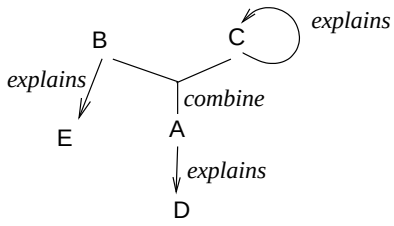

Figure 1: A possible tower of models

can arise partly via combination, and partly because explanation is a many-many relation: different aspects of a model B may be explained by different models A_1, A_2,...; equally, a model A may explain different models B_1, B_2,.... However, a tower with no explanations —one that is very short— is of little use.

What role does such a tower play in informatics? Natural sciences pertain to realities that are given. These sciences are anchored in the real world; much of what a natural scientist does is to validate or refute models of this reality by experiment. In contrast, except at the lowest level of silicon or optical fibres, informatics builds its own realities; also, crucially, it builds symbolic models to explain these realities at many levels, shading from reality to abstraction with no sharp distinction between the two. Such levels are—roughly in order from less to more abstract—computers themselves, memories, networks, low-level programming, high-level programming, algorithms, programming frameworks (object-oriented, neural, intelligent agents), program transformation, specification languages, graphical representations, logics, mathematical theories, ... There are many models at each of these levels.

Correspondingly, every model addresses a certain class of *clients*: those for whom its explanations are intended.[5] In the natural sciences many models are designed for the scientist, but models designed for the general public are also essential for public understanding. Clients for informatic models span a huge range of concerns and ability, ranging from the many millions of private end-users, through executives in client companies, through the technical staff in such companies, through suppliers of custom-assembled products, through programmers of support software, through software architects, down to academic engineers and theorists.

No-one could attempt to describe the *whole* tower of informatic models. Although the science of informatics has advanced enormously in its sixty years, new technologies continually increase the possible realities, and therefore increase the challenge to build models that explain them. But the notion of a tower, ever incomplete and ever growing, provides a path that the growth of our science can follow; we may build *partial* model-towers, each representing a particular field, with models designed for different clients and cohered by explanation.

Figure 1 shows a possible structure for a small model-tower. A is a combination of B with C; A explains D; B explains E; C explains itself. To see that self-explanation

[5]More generally, every model has a distinct purpose. For example, a single client may use different models of flight-control software for different forms of analysis.

makes sense, recall that ‘M explains N’ is a short way of saying that the entities of model N—say programs—may be explained (e.g. specified) by entities of model M—say a logic. A good example of ‘C explains itself’ is provided by the refinement ordering invented at Oxford; to refine a specification is to constrain its non-determinism. Thus a coarser specification explains each of its refinements. Such a notion of refinement is also built into Eiffel, an object-oriented language for specification and programming.

For M to explain N, there is no need to require that *every* entity of N is explained by entities of M. For example flowcharts explain some programs, but not those with recursion. When we want more precision we can talk about *full* or *partial* explanations; and the latter should allow that only some entities of N are explained, or that only some aspects of each entity are explained.

Now recall that different models are designed for different clients. For example, if M is designed for senior executives then we may expand ‘M explains N’ into the statement ‘M explains N *for* senior executives’. In the example pictured above, suppose B consists of certain differential equations, and C is a process calculus; then the combination A explains hybrid systems. However, B is designed to explain only E, the electronic component of A, to control engineers who need not be familiar with process calculus. An important function for a model tower is to cohere the models designed for different clients.

Examples

The variety of explanations may create unease; can we not formally define what ‘explanation’ means? Not yet: this paper aims to arouse discussion of that very question. To feed the discussion, here are some examples that illustrate the variety of explanations. In each case we outline the structure of a small model-tower. To establish firm ground our first example, though elementary. will be defined precisely; it indicates that model-towers can be rigorous. The other examples are treated more informally; indeed, a main purpose of models and their relationship is to allow informal insight into how software science may be integrated with software engineering.

Programs We consider both specification and implementation for a fragmentary programming language; this yields the small tower shown in Figure 2. Research over the past four decades ensures that the same can be done for realistic languages; but as far as I know these two activities—specification and implementation—have not previously been presented as instances of the same general notion, which we are calling ‘explanation’.

Let $X = \{x_1, \ldots, x_n\}$ be a fixed set, the program variables. Let V be a set of values, say the real numbers. A map $m : X \to V$ is called a memory; let M denote the set of memories. Consider three models:

> Programming language P. An *entity* p is a sequence of assignment statements like $x_1 := 3x_1 + 2x_2 - 4$. The *meaning* of p is a function $\mathcal{P}[\![p]\!] : M \to M$, and is defined in the standard way.

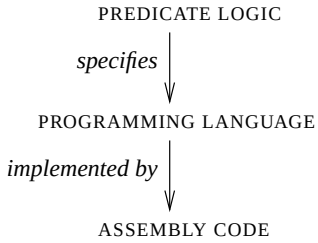

Figure 2: a small tower of models for programming

Assembly code C. An *entity* c is a sequence of instructions of the form add, mult,..., load_v, fetch_x, store_x where $v \in V$ and $x \in X$. These instructions manipulate a memory $m \in M$ and a stack $s \in V^*$ in the familiar way, defining the *meaning* of a code c as a function $\mathcal{C}[\![c]\!] : M \times V^* \to M \times V^*$.

Predicate logic L. An *entity* ϕ is a logical formula with free variables in X and bound variables distinct from X. The *meaning* of ϕ is a map $\mathcal{L}[\![\phi]\!] : M \to \{\mathsf{true}, \mathsf{false}\}$; this is a standard notion, called by logicians a *valuation* of ϕ in M.

To implement P we define a compiler Comp that translates each assignment $x := e$ into a sequence of stack operations, in the standard way. The implementation is validated by a theorem stating that if $\mathcal{P}[\![p]\!]m = m'$ then $\mathcal{C}[\![\mathsf{Comp}(p)]\!](m, s) = (m', s)$ for any stack s. Thus the implementation has a formal part—the compiler—relating entities, and a semantic part relating their meanings.

To explain P by L also involves a formal part and a semantic part. The formal part is a relation which may be called 'satisfaction', denoted by $\models$, between programs p and pairs ϕ, ϕ' of logical formulae. If we write $p \models \phi, \phi'$ as $\models \{\phi\}p\{\phi'\}$, we recognise it a 'Hoare triple'; a sentence of Hoare's well-known logic. In that logic such a triple may be proved as a theorem, written $\vdash \{\phi\}p\{\phi'\}$. The explanation is validated by relating these formal triples to their meaning; it asserts that

$$\text{Whenever } \vdash \{\phi\}p\{\phi'\} \text{ and } \mathcal{P}[\![p]\!]m = m', \text{ then } \mathcal{L}[\![\phi]\!]m \Rightarrow \mathcal{L}[\![\phi']\!]m' \ .$$

Thus we have seen how explanation may consist of a formal part, in this case a compiler or a logical proof, that may be executed by a tool, and a semantic part that gives meaning to the work done by the formal part. Both parts are essential to modelling.

Electrical installations The left-hand diagram of Figure 3 shows a small tower that coheres two models of an electrical installation; one for the architect and home-owner, the other for the scientist. Architects understand a model of requirements for an electrical installation in a house in terms of performance—heating, lighting, refrigeration etc—also taking account of locality, maintenance, cost and other factors. In these terms they specify the appliances, and home-owners also understand this specification. On the other hand the appliance designs are explained by electrical science, which more

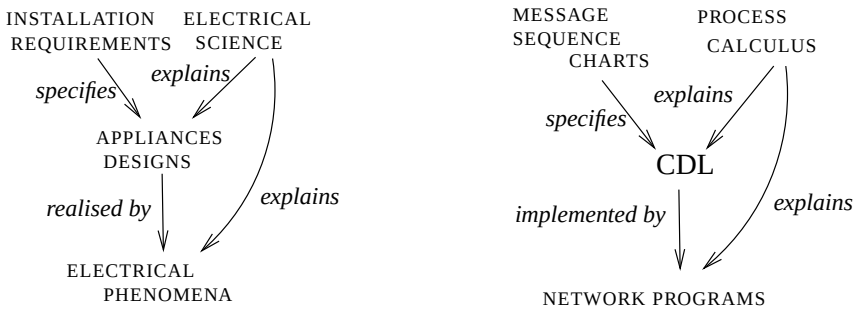

Figure 3: Model towers for electrical installations and for network programs

generally explains the full range of electrical phenomena. The left-hand diagram shows these model relationships. An important aspect of the example is that a single model—the appliance designs—is explained differently for two different kinds of client: in one case the architect or home-owner, in the other case the electrical engineer or scientist.

Business protocols An analogous tower is shown in the right-hand diagram of Figure 3; it concerns a software platform for the computer-assisted enaction of business processes. Such a platform is being designed by a working group of the Worldwide Web consortium (W3C). The workhorse of this platform is the Choreography Description Language (CDL), which has been designed and intensively analysed by the working group. This collaboration allowed a rigorous explanation of CDL in terms of process calculus, which also explains network programs; the implementation of CDL in a low-level language can thus be validated.

The clients of CDL are application programmers, and their concerns are different from those of the scientists, who are clients of the process calculus. They differ again from the concerns of executives in the client companies; these executives in turn understand message-sequence charts, a simple graphical model that represents at least some of the communicational behavior defined in CDL.

Before leaving this example, it is worth mentioning that the right-hand tower extends naturally both 'upwards' (more abstract) and 'downwards' (more concrete). Downwards, the low-level network programs are realised by a combination of physical computers and physical networks; upwards, a process calculus can be explained by a special kind of logic.

The airbus Our final example applies rigorous modelling to a safety-critical system. After the failed launch of the Ariane 5 spacecraft, the Institut National de Recherche en Informatique et en Automatique (INRIA) in France—of which Gilles Kahn was then Scientific Director—undertook to analyse the failure. The analysis was informative. As a consequence, Kahn proposed that INRIA should conduct rigorous analysis for the informatic aspects of the design of the forthcoming Airbus.

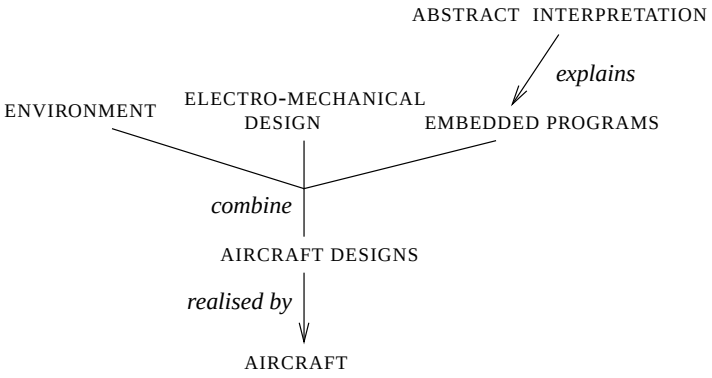

Figure 4: A simplified model tower for aircraft construction

Such an analysis can often be based upon a specification in logical model, perhaps a temporal logic; the logical sentences are chosen to characterise the desired behavior of the embedded software (the program model). This software model has to be combined—as remarked earlier—with an electro-mechanical model of the plane, as well as a model of the plane's environment. Thus we arrive at a tower like that shown in Figure 4; of course this is only a simplification. The method chosen for analysis, based upon *abstract interpretation*, can be seen as a refinement of the logic-based approach. An abstract interpretation of a program is a simplification of the program, omitting certain details and making certain approximations, with the aim of rendering detailed analysis feasible. Such abstraction is essential in situations where the state-space is very large; but, to be sound, it must not conceal any undesirable behavior. Thus, instead of choosing a fixed specification, one may choose an abstraction specifically to match those aspects of behavior that are essential. In the case of the Airbus, by a careful choice of different abstractions, the analysis required to validate the embedded programs was reduced to the extent that, with the assistance of programmed tools, it could be completed.

The Airbus example illustrates that explanations and their validation can be customised within the framework of a tower of models. It also illustrates the importance of programmed analytical tools.

This concludes our examples. It is a good moment to answer a possible criticism of our notions of model and explanation. The criticism is that whenever a model is defined, the meaning of its entities—which are often symbolic—has to be expressed in some formalism; thus a model does no more than translate between formalisms, and our search for real meaning leads to an infinite regress.

Our answer is in two parts. First, every model-designer clearly has *some* meaning in mind. A programming language, or a logic, or a process calculus, or a graphical representation is never just a class of symbolic entities; its intended behavior is always described, even if informally. Thus it is clearly inadequate to call such a class a model

in itself. Second, as we move from entities to meaning within a model, or indeed as we move from the entities of model B to those of model A which explains B, we typically move from a specialised entity-class to a class belonging to a more general model. This can be seen clearly in all our examples; e.g. CDL is more specific than a process calculus, so in explaining it we move to a model for which there is already a body of knowledge.

Our examples show that scientific work in informatics consists not only in designing models, but even more in relating them to each other. The former is essential, but only the latter can provide the coherence that will enable both other scientists and the general public to grasp the quality of informatics.

Divergent approaches

Increasingly, over sixty years, informatic theory and applications have diverged from each other. At present, their relationship is almost tangential; few people understand or practice both. On the one hand an impressive range of theoretical concepts, inspired by applications, have emerged and been developed. Here is an incomplete list, roughly in order of discovery:

> universal machines, automata theory, formal language theory, automation of logics, program semantics, specification and verification disciplines, abstract interpretation, object-orientation, type theories, process calculi, neural nets, temporal and modal logics, calculi for mobile systems, intelligent agents, semi-structured data, game-theoretic models, quantum computing, separation logic, ...

On the other hand the industrial production of software has developed sophisticated processes for implementation and management, stimulated by a huge and growing market demand. These two developments, theoretical and industrial, are largely independent of one another. It is hard to see how this could have been avoided. No scientific and technological discipline in human history has been so rapid or so global. Responding to hungry demand has preoccupied the industrial wing of informatics; competition in the market has made it virtually impossible to win contracts while maintaining a rigorous standard of validation, or even documentation, of software. On the other hand, building models for rigorous analysis is a daunting intellectual challenge, especially as technological advance continually creates startling new realities—such as pervasive or ubiquitous systems—to be modelled.

Despite the frequent delay and technical failure of software systems, and despite the fact that future software systems—especially pervasive systems—will be larger and more complex than ever, there is a danger that this disconnection between software engineering and analysis becomes accepted as a norm. There is evidence for this acceptance. For example, in a report entitled *The Challenge of Complex IT Systems* produced jointly by the UK's Royal Academy of Engineering and the British Computer Society, the phenomenon of defective IT systems was examined in detail. Many cases were studied, and valuable recommendations were made from the point of view of management and the software production process. But virtually no mention was

made of the need for, and possibility of, rigorous analysis based upon fully developed theories.

Rapprochement?

Paradoxically, while the need for scientific system analysis has been neglected by some, there is currently a strong drive in other quarters to base software development and production on models. This trend has been called 'model-driven engineering' (MDE). The academic research carried out under this heading is very varied, and the author is not equipped to summarise it. An optimistic view is that, while the MDE approach may appear superficially incompatible with the framework proposed here, the approaches differ only in terminology and emphasis. One purpose of the present essay is to induce a rapprochement between the approaches; such a rapprochement is not only possible, but absolutely necessary, for informatics to gain its proper position as an integration of scientific understanding with industrial construction.

An extreme form of MDE has, as its prime concern, that software production be based upon automatic tools which transform one class of syntactic entities into another. Sometimes such an entity-class, e.g. the syntax of a programming language, is called a model. This conflicts with the notion of model proposed here; and while terminology is conceptually unimportant, such a conflict may inhibit rapprochement. In this particular case, it may inhibit the union that I am advocating between science and engineering, since the scientific sense of 'model' lays emphasis on the notion of meaning, which is absent from syntactic entities in themselves.

The MDE research community also—in varying degrees—seeks this union. This essay has proposed that the union can be found via a notion of model which associates meaning with every syntactic entity, and via a notion of explanation between models that includes not only a transformation between syntax-classes, but also a representation between their corresponding meaning-classes. Both the transformation and the representation can be highly engineered as tools; thus the union itself can be engineered! This essay will have achieved its goal if it promotes a constructive debate, involving both engineers and scientists, upon the notion of 'model'.

Acknowledgements I am grateful to all those, including Gilles Kahn himself, with whom I have discussed these questions, and to the anonymous referees for their helpful comments. They have made my suggestions (which are hardly original) seem plausible to me, even though there are many difficulties to be overcome before convergence between the engineering and science of informatics becomes real. Next steps on the path must include reports of detailed case studies, with detailed bibliography, that appear to contribute to the convergence.