A Taxonomy for Computer Science

Hanno Wupper and Hans Meijer
Computing Science Institute
Faculty of Mathematics and Informatics
University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, NL
Hanno.Wupper@cs.kun.nl Hans.Meijer@cs.kun.nl

'To design is to invent a formally provable statement'

Abstract

We try to capture the essence of information technology and computer science, arguing that information technologists have the same principal goal as all technologists: to create machines with certain properties. To achieve this, they formalize the problem, i.e. abstract the properties into a specification and invent or develop a schema, i.e. an abstraction of the machine's structure. Subsequently, it is their principal task to *prove* that the schema satisfies the specification. Computer scientists develop mathematical and physical means to support or even enable that task. From this, the principal research questions of computer science may be derived.

From this viewpoint, we try to propose a consistent set of notions together with a consistent terminology, which may clarify the relation of information technology and computer science to other scientific disciplines and also give rise to new ideas about computer science education.

Keywords

informatics, taxonomies, academic requirements

Acknowledgements

We wish to thank Henk Barendregt, Guy Debrock, André van den Hoogenhof, Toine Tax, Huub van Thienen, Frits Vaandrager, Tom van Weert, and all other colleagues who contributed to this paper with their ideas, criticism and sharp questions.

1. Introduction

In computer science there exists only limited terminological agreement. Already for its very name one has the alternatives 'computing science' and 'informatics'. Fundamental terms like 'state' or 'automaton' or 'specification' or even 'program' may have different meanings in different contexts. Often such terms are not explicitly defined but simply borrowed from common language, or else are defined quite differently. It may very well be the case that only the term 'bit' qualifies for having a universally accepted meaning.

This situation is an inevitable reflection of the fact that there is no common understanding of the core of this particular branch of science—let alone a basic understanding among the general public.

This paper tries to answer the question what information technology[1] is all about and what computer science has to do with it. In doing so, it proposes a consistent collection of clear and unambiguous terms for notions which are essential for the core of the discipline. In fact, these notions are the important issues, not the terms: we try to establish a set of fundamental notions, for which we hope to find consistent terms. We do not claim to arrive at particular new results, apart from an overall insight in the field: information technologists and teachers and students of computer science may find these notions helpful to disentangle complex achievements of computer science and re-use their constituents in other contexts.

> Many achievements of computer science are presented as 'paradigms'. An example is 'object orientation': its value is beyond doubt, but for information technologists who did not grow up in the object oriented culture it may be difficult to see what it is all about. Guided by our notions they may find, among others: a *specification method* that can be useful in many contexts; a number of *language issues* in specification and programming languages; and an amount of *theory* (abstract data types, inheritance, etc.) which can be useful as a guideline even for someone who has to develop Fortran or Cobol programs.

Our method is to begin with a basic, abstract and general observation, which is subsequently refined and detailed. This observation is based on a development cycle, which is related to the various forms of the ubiquitous *software life cycle* of software engineering, but tries to unify and normalize them.

> It seems that in particular software engineering has produced an abundance of points of view, of terms and of notions which sometimes look all alike and all different. What is the difference between a program and an algorithm? What is a system? When is it appropriate to call something a module?

The framework presented here should help to bring some order in the terminological chaos. However, an equally important goal is the generalization of the development cycle to the whole area of information technology. In fact we believe that every aspect of information technology and computer science has its place in our framework. Of course this very much depends on the understanding of the notions 'computer science' and 'information technology', and indeed we take these notions in the widest possible sense: it is the science of and technology of the use of computers and computer applications, the creation of such applications, the specification of customers' desires, the proofs of correctness of applications, etc.

Going a step further, we are even convinced that our framework may be applied *mutatis mutandis* to a much larger segment of science and technology. The discovery and (experimental or theoretical) verification of laws of nature and the creation of technological products from that scientific knowledge may be described and classified in a totally analogous way. However, apart from the fact that other more established fields of science may not be in need of such a clarification, we will not explicitly make such claims in this paper.

Summarizing, we try to propose a consistent set of notions with respect to which all important aspects of computer science can be understood, together with a consistent terminology—which, however, may nevertheless rouse a debate. Moreover, we hope

[1]With 'information technology' we explicitly refer to the actual, professional, manufacturing of applications etc.

that this set of notions may greatly clarify the relation of information technology and computer science to other scientific disciplines and also give rise to new ideas about computer science education.

We shall employ —and explain in due course— three different frameworks: a formula, a diagram (the one resembling the software life cycle), and the tetrachotomy *theory*, *method*, *language* and *tool.*

2. The formula: an information technologist's principal task

We begin with two fundamental notions which refer to the real world.

Definition. A *machine* is some physical[2] object which has been intentionally constructed from certain parts for some well-defined purpose.

Definition. A *property* is a physical phenomenon which can be ascribed to a physical object.

A machine is said to *have* a property to express the proposition that that property is ascribed to that machine.

> Simple examples of an object's properties are its weight, speed or colour. However in the context of information technology one is mainly interested in complex, dynamic properties like 'controlling a nuclear reactor's temperature' or 'navigating an aeroplane'. Here we are primarily interested in externally observable properties. In particular the property 'structure' will be given a special rôle below.
>
> We deliberately propose to restrict the term 'machine' to objects of the real world[3]. A watch is a machine and so is a computer with or without software, but in our view a formula or a computer program is not. One's interest in abstract entities like programs must ultimately stem from one's wish to make a machine have properties such that a certain goal is achieved.
>
> In information technology, functional correctness and efficiency have traditionally been considered important properties of (machines executing) computer programs. Other properties like time-constraints or reliability which first seemed too difficult to deal with (and seemed delegated to other disciplines such as 'reliability engineering') have recently begun to admit being reasoned about in a more uniform framework.

Whether a machine actually has all desired properties is the problem of *adequacy*.

> This problem may be very hard or even impossible to decide. One may simply not know particular laws of nature, or be blocked by fundamental inaccuracies or uncertainties of measurement, but even if the physics are easy, the machine may still be too complex. Moreover, *discovering* a property is generally much more difficult than *deciding* one. In the sequel we shall divide the problem of adequacy into three subproblems: that of *meaning*, that of *structure* and that of *(formal) correctness.*

Next we discuss a basic notion which is the descriptive mirror image of the one called 'property'.

Definition. A *specification* is a statement of properties, in some suitable language.

A specification is said to *state* a property, viz., to be obeyed or fulfilled.

Whether a specification states certain properties is the problem of *meaning*.

[2]or 'substantial', see (Bunge), 1977

[3]nevertheless acknowledging its metaphorical use as in 'Turing Machine'

Subsequently a machine is said to *fulfil* a specification if for each property from a sufficiently well-defined collection, the machine has that property whenever the specification states it.

A specification states the properties of an existing or desired machine. In the latter case the specification may become a *contract*.

Whether a machine fulfils a specification is the problem of *acceptance*.

Given a specification, the problem of *adequacy* may now be decomposed into the problems of *acceptance* and that of *meaning*.

Our fourth basic notion is the descriptive counterpart of a machine and the concrete counterpart of a specification—in the same way as a property is the abstract counterpart of a machine.

In the term 'schema' we want to capture the description of the (detailed) structure of a machine. According to the Oxford English Dictionary *structure* is 'the mutual relation of the constituent parts or elements of a whole determining its particular nature or character'. In other words, a machine's structure is the specific or even characteristic way in which it is assembled from its parts—and a schema is a description of such a structure.

Molecules, for instance, are 'machines' assembled from atoms by means of 'chemical binding'. Chemistry studies the structure of molecules, taking (only) the properties of atoms into consideration, and not looking inside them (as suggested by their very name). On the other hand, chemistry is not primarily interested in living cells, computer chips, plastic toys, etc., which are in turn 'assembled' from particular molecules. Whereas many sciences, like chemistry, seem to confine themselves to a more or less homogeneous class of parts, computer science studies a whole hierarchy of parts like gates, processing elements, instructions, programs, languages, computers, information systems, control systems and networks.

Definition. A *schema* (of arity $n \geq 0$) is a pair (s, X) where s is a sequence of n specifications and X is a structure description containing n numbered 'place holders' for components.

A schema must be sufficiently detailed, such that it is at least feasible to *make* the machine.

A machine is said to *be assembled from* a collection of n parts conforming to a structure description X if it is faithfully built according to X, with each part set in the position of the correspondingly numbered place holder.

A machine is said to *be a realization of* a schema (s, X) if it is assembled conforming to X from a numbered collection of n machines which fulfil their correspondingly numbered specifications in s.

Whether a machine is a realization of a schema is the problem of *structure*.

Note that a machine contains particular parts where a structure description only contains place holders, to which a schema assigns requirements by way of specifications. The schema treats the parts as 'black boxes' of which only properties are prescribed, not the internal structure—while the structure itself is a 'glass box'.

In summary we now have that a machine has or is supposed to have certain properties; that a specification is a description of (those) properties; and that a schema is a description of the (precise) structure of a machine, relative to the specification of the properties of its parts.

A schema is said to *satisfy* a specification if every machine which is a realization of the schema fulfils that specification.

This leads to the prospected formula: a schema (s, X) satisfies a specification S if any machine that can be built according to X using parts fulfilling s, fulfils S. Formally,

$$(\forall i\text{: } 1..\#s.\ \ m_i\,\textbf{\textit{fulfils}}\ s_i\,) \Rightarrow (m\ \textbf{\textit{assembled}}\ X)\ \textbf{\textit{fulfils}}\ S \qquad (*)$$

Since specifications and schemata are mathematical objects (descriptions of properties and structure), we may now capture the essence of an information technologist's task in mathematical terms. In order to build a machine with certain properties he or she must produce a specification *S*, a schema *Y* and a proof *p*, such that *p* is a proof that *Y* satisfies *S*. Then, if *S* states the desired properties, the desired machine is a realization of *Y*.

> The proof *p* may be any mathematically acceptable proof, including fully formal proofs as well as sufficiently precise reasoning. In fact, if information technologists discuss their product, be it finished or in development, with colleagues or their superior or a customer, they will primarily argue in favour of their schema satisfying the specification. This argument can only be interpreted as a try to establish a proof, however sloppy it may be.
>
> Actually, the only alternatives to proofs seem to be a *belief* in some form of higher authority (based on power or experienced knowledge or intellect) or an *insight* by which the truth of a statement is immediately clear.

Whether a schema satisfies a specification is the problem of *correctness*, which is particularly hard because of its *complexity*.

A schema is a description of the structural composition of parts of which specifications are given. This enables us to treat most of the complexity of the adequacy problem in the mathematical domain: only the meaning of the specification and the physical realization of the structure defined by the schema cannot be dealt with mathematically.

Moreover, it admits a reduction of the complexity problem by hierarchical decomposition. One may initially restrict the schema to large parts which are optimally chosen to allow simple specifications and an easy structure description and then repeat the same activity for the parts. This is referred to as the 'Chinese Box Principle', where the box is composed of boxes which are composed of boxes, *etc*.

This procedure avoids the problem to capture all complexity at once, but only if and when the specifications of parts hide (abstract from) details which exist only locally. One other gain is that parts may be reused (spatially as well as temporally). A striking example is the programmable computer, where the 'architecture' or 'programmer's model' is a relatively simple specification of a very complex part and one 'only' needs to deal with the complexity of the other part, viz. the program.

To summarize: in order to establish that a machine has a collection of properties, one must come up with a schema and a specification, and establish that the machine is a realization of the schema, the schema satisfies the specification and the specification states the properties.

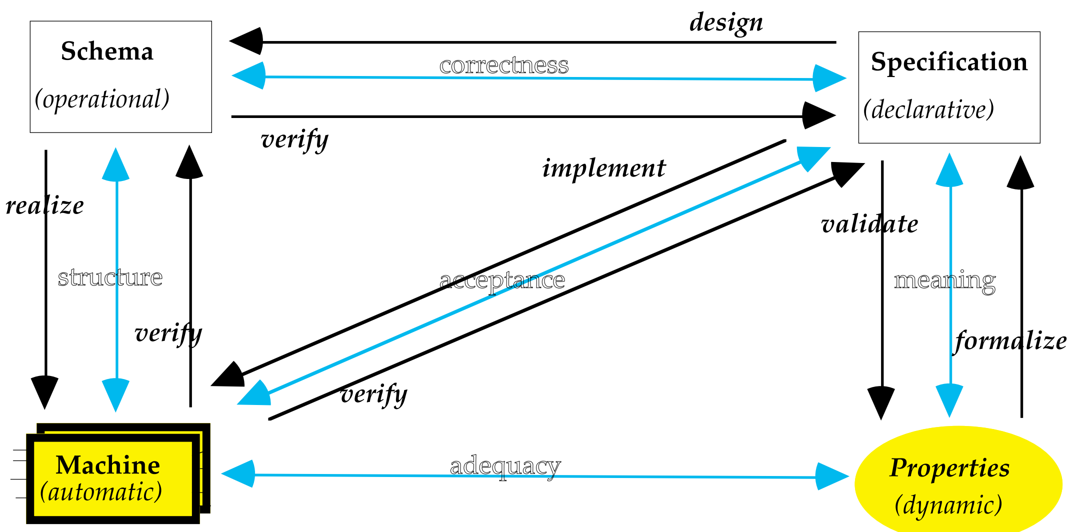
> Of course most systems are constructed without a formal correctness proof along formula (*). Nevertheless technologists should understand that what they are doing informally *could* be formalized in a systematic way and proved mathematically.

By contraposition one may find out what may cause any 'faults', viz. that the machine does not have a certain intended property. It may be that the specification doesn't state it, in which case one should repair the specification (by 'declaring the bug a feature' or otherwise). Assuming that the specification precisely states the desired properties, we call the machine to be 'faulty' (i.e. not fulfilling its specification). Then either the schema must be incorrectly designed (and not proven anyway), or the machine must be assembled in the wrong way, or else at least one of the parts itself is faulty. This leads to the classification: *specification mistake*, *design mistake, assembly mistake, faulty component*.

If a machine must be *fault tolerant,* there is no other way than to weaken the specification s_i of the error-prone component m_i until m_i fulfils s_i and then to re-establish the truth of formula (*).

3. The diagram: professional activities

The four notions introduced in the previous chapter give rise to the following diagram of which they constitute the 'cornerstones'. The five characteristic problems introduced there appear as relations between them.



In fact, the boxes refer to particular objects, i.e., instances of the respective notions. Between (nearly) every pair of objects there exist certain arrows which capture quite disjoint *professional activities* and, we believe, together cover *all* essential activities carried out by working information technologists.

The diagram is a reworked version of the well-known *software life cycle*[4], which is an Escher-like variant of the 'waterfall model' (Sommerville, 1996). It is also an idealized diagram in the sense that in practice the activities are not normally performed in strict sequence but iterated, backtracked, *etc*.

The main activity of *computer scientists*, however, is on a meta-level, as shall be explained in chapter 4.

Definition. *To formalize* is to write a formal specification which states exactly the desired properties. This activity comprises sub-activities like their actual formalization in some language and their structuring for purposes of presentation, reuse, etc.

Definition. *To validate* a specification is to establish that that specification indeed states the desired properties. To this end one must become sure about what will be the

[4]However, we see no reason to distinguish between hard- and software systems.

case when the specification is made true. There are basically three ways to become sure, viz. by

- *insight*, when the specification is so clear that we comprehend it immediately,
- *prototyping*, where the prototype is tested or measured, or
- *reasoning*, where additional properties are mathematically proven.

Definition. *To implement* is to physically make a machine which fulfils a given specification. This is either accomplished rationally, by designing and realizing a schema, or by putting parts together intuitively.

Definition. *To verify a machine against a specification* is to ensure that a machine fulfils a given specification. If a correct schema of the machine cannot be produced, verification must be performed *inductively* (or in 'black box'-style) where as many hypotheses as possible derived from the specification are verified by experiments. Information technologists call this *testing*.

Definition. *To realize* is to physically make a machine which is the realization of a given schema. This involves picking up the required parts and putting them together in the right way. In the case of pure software systems, we assume —by using the word 'pure'— a programmable machine to be used as part. As a consequence, the realization of the machine as a whole is trivial, viz. loading a machine-readable representation of the schema into it[5]. In the case of a machine in hardware one may want to let the realization be performed automatically.

Definition. *To verify a machine against a schema* is to ensure that a machine is a realization of a given schema. Note that a verification is (only) necessary if the machine is made by an untrustworthy device, in particular a human—in which case the verification consists of checking whether the right parts are assembled in the right way. When the machine is very complex or cannot be opened (e.g. without breaking down), it may be that one can only verify the machine against its specification.

Definition. *To design* is to elaborate a schema which satisfies a given specification. In our framework this means no more and no less than finding a suitable schema (s, X) such that our formula (*) becomes true. Apart from small 'toy' cases this activity is too complex to be performed without computer support. On the other hand, the activity cannot in general be fully automated. *Designing is a creative mathematical activity, which comprises finding a theorem, if necessary strengthening its assumptions until it can be proven.*

Definition. *To verify a schema against a specification* is to prove that a schema satisfies a given specification. This verification must be performed *deductively* (or in 'glass box'-style). The automatic part of the design will not need a proof as long as the used tool is trustworthy. The hand-made part may be elaborated systematically and stepwise, in which each step will need to be proven. Or the schema may be 'invented' in which case it must be proven from scratch to be correct.

[5]This may be the reason that in software engineering the term 'realization' is often used for the activity that we call 'design'.

4. The tetrachotomy: a computer scientist's principal task

The diagram as given in the previous chapter displays the professional activities of working information technologists. However, the primary activity of *computer scientists* is research which results in some form of support for the professional activities. Often, such support is presented in the form of a software package or a new language or 'paradigm'. This may hamper dissemination and acceptance of valuable items.

> Even if one cannot afford to buy an expensive software package or to switch to a new programming language one may benefit from pieces of theory on which they are founded or follow a method which they are meant to support.

It may therefore be useful to separate complex results into the following classes.

Definition. A *theory*, which is a collection of scientific laws or theorems ('true statements'), usually generated by a finite collection of deduction rules from a finite collection of axioms. Most theories are infinite, in which case one needs a proof and a decision procedure to establish whether a statement is true.

Definition. A *method*, which is a collection of notions, rules and procedures which may help to systematically find a solution for a given problem[6].

Definition. A *language* or *formalism*, which is a collection of terms (or sentences, or sequences of symbols) with well-defined syntax and semantics (which maps each well-formed term to its 'meaning' in some physical or mathematical model, usually following its syntactical structure).

Definition. A *tool*, which is a computer system by which parts of the professional activities may be automated. The fact that most tools which support information technology are themselves prime examples of products of that technology cause what one might call a 'strange loop of computer science' (Hofstadter, 1997). Of course do many technologies produce their own tools (for instance, engineering), but it seems that in information technology it happens at an overwhelming scale[7]. In particular the fact that initially software tools are the results of research by computer scientists causes an intricate intertwining of computer science and information technology[8] which is one of the primary causes of the terminological chaos referred to in the introduction.

5. Conclusion

Summarizing, we believe that every professional activity in information technology archetypically consists of

---

[6]Unfortunately, methods are often calles 'methodologies', but that term should really be reserved for the theory of *all* methods (e.g. answering the question what a method is, and what not).

[7]The operating systems of our current personal computers are developed by means of tools on earlier computers, and so on until the very fourties. Perhaps the Algol 60 compiler for the ELX1 by Dijkstra and Zonneveld is the only compiler ever completely written 'from scratch', but even then some kind of assembler/loader was undoubtedly needed to bring it in the machine. In this way the development of information technology may be seen as one gigantic *bootstrap*.

[8]Is $\mathrm{T_EX}$ the work of a programmer or a scientist?

– writing down a problem in some language (the specification),

– developing a solution (a schema) for it using some method, while writing down the partial or intermediate solutions in that same or some other language,

– proving—on the fly or afterwards—that the schema satisfies the specification, by showing that our formula when applied to that specification and schema is a theorem in an appropriate theory, while

– employing tools in doing so

and that it is the principal task of computer scientists to provide the necessary languages, methods, theories, and tools.

Scientific understanding is impossible without abstraction. Moreover, abstraction of the reality into mathematical models enables scientists to reason about the reality by mathematical means. However one may need to be very careful about the appropriate level of abstraction. In many fields of computer science one may identify a machine with the schema of its software (‘the program is the true result’), in some other fields one may identify a schema with its specification (‘the program is an executable specification’), but negligent use of such abstractions may be very confusing.

Therefore we believe that one should always try to depart from the distinction between the mathematical description of a computer program (the specification), the bit pattern representing that program inside the computer (the schema) and their physical combination whilst executing (the machine), which has certain effects in the real world (the properties). The distinction appears even on the very abstract level that our diagram represents. Not separating the three aspects is, we believe, the main cause of the Babylonian confusion which pervades computer science and which has at least the following consequences.

• Technical terms may have (slightly) different meanings in different research areas, which does not favour communication among scientists.

• Computer scientists, particularly young ones—students, may have difficulties putting results of others in context.

• It may be unnecessarily difficult to relate the results of computer science to those in other disciplines.

• Finally it may be very hard to explain to the general public or the authorities what computer science is all about and which benefits may be expected from research programmes.

We have tried to give a general framework for an answer to that question. People want real machines with certain real properties. Information technologists must design such machines in a professional way and verify that these machines indeed do what they are expected to do. Computer science provides the theories, languages, methods and tools for accomplishing those tasks. These may be associated with the elements of the diagram of chapter 3 in a systematic way. Languages have their rôle in the mathematical nodes ‘specification’ and ‘schema’ with all possible intermediate nodes. Theory deals with the mathematical modelling of the physical nodes ‘machine’ and the ‘properties’ and with the five basic relations between the nodes together with their associated problems, thereby laying the foundation for the formula of chapter 1. Methods and tools support the arrows, i.e. the professional activities, in particular where they are done by hand or automatically, respectively. In a forthcoming paper we will elaborate the research questions of computer science in much greater detail along these lines and try to develop a formal basis for them, of which a very brief outline is given in the appendix.

Systematically applying Ockham's razor, we have touched upon a number of *divide-and-conquer*-like classifications which may be of great help in explaining computer science to, for instance, students. We summarize:

physical – mathematical

meaning – correctness – complexity – structure

formalize – design – realize

act – verify

theory – method – language – tool

professional activities – scientific research

Furthermore we hope that our approach clarifies much of the relationship with other scientific disciplines. Computer science may learn a lot from the 'development cycle' in engineering disciplines as well as from the 'verification cycle' in the natural sciences. The rôle of induction vs. deduction points to an 'equal opportunity' status for experimental methods. Mathematics will of course teach us how to handle the central correctness problem and how to deal with its complexity.

Finally, we have experienced that our taxonomy may help students of information technology—the future makers of software—to distinguish essential concepts from everyday's whim and to see their own future profession in a broad perspective; it may help them understand why they may have to learn new methods, etc. during their whole professional life; it may help students of computer science to see more clearly what their theories, languages, methods and tools are good for. Scientific as well as professional education should be *methodological* and we hope that our framework is a contribution to that.

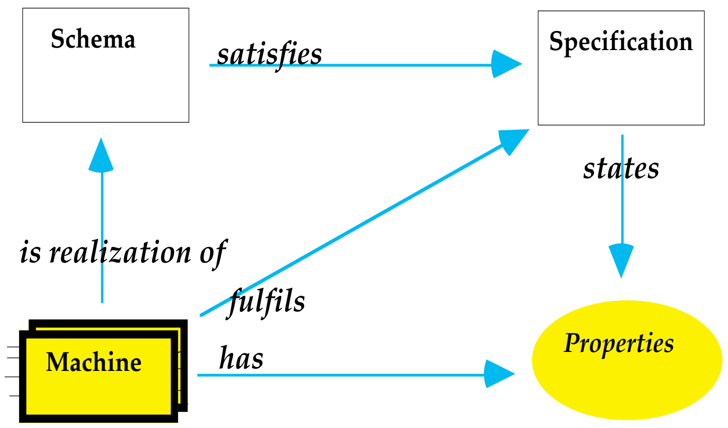
References

Hofstadter, D.R. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books, New York.

Sommerville, I. (1996). *Software Engineering.* Addison-Wesley, Reading, Massachusetts.

Bunge, M.A. (1977). *Treatise on Basic Philosophy, vol. 3: Ontology I.* Reidel Publishing Company, Dordrecht, Holland.

Appendix. Outline of a theoretical foundation



The theoretical foundation of the five relations depicted here must be based on a model of reality in terms of mathematical structures **Prop** , **Mach** , ***has***: **Mach**$\rightarrow\mathbb{B}\leftarrow$**Prop**, **Ass**$\subset(\bigcup n: \mathbb{N}.$ **Mach**$^n\rightarrow$**Mach**) and on the semantics of the specifications and structure terms, given by **Spec**, ***states***: **Spec**$\rightarrow\mathbb{B}\leftarrow$**Prop**, **Struct**, ***assembled***: **Ass** $\leftarrow$ **Struct**.

The schema language and the remaining relations are then defined as follows.

def ***fulfils*** : **Mach**$\rightarrow\mathbb{B}\leftarrow$**Spec** **with** ($M$ ***fulfils*** S) = ($\forall P$: **Prop**. S ***states*** $P \Rightarrow M$ ***has*** P)

def Schema := $\bigcup n: \mathbb{N}.$ **Spec**$^n\times$ **Struct**(n)

def ***is realization of*** : **Mach**$\rightarrow\mathbb{B}\leftarrow$(**Spec**$^n\times$ **Struct**(n))

with (M ***is realization of*** (s, X))= ($\exists m$: **Mach**^{n}. ($\forall i$: 1..n. m_i ***fulfils*** s_i) $\wedge$ M = (m ***assembled*** X))

def ***sat*** : **Schema**$\rightarrow\mathbb{B}\leftarrow$**Spec** **with** ($Y$ ***sat*** S) = ($\forall M$: **Mach**. M ***is realization of*** $Y \Rightarrow M$ ***fulfils*** S)

From these, the following can be concluded, which is the justification of this approach:

Theorem. M ***is realization of*** $Y \wedge Y$ ***sat*** S $\wedge$ S ***states*** P $\Rightarrow$ M ***has*** P

Another conclusion exhibits our formula (*) as the essence of the goal of the design process:

Theorem. ((s, X) ***sat*** S) $\Leftrightarrow$ ($\forall$i: 1..#s. m_i ***fulfils*** s_i)$\Rightarrow$ (m ***assembled*** X) ***fulfils*** S)

In order to prove that Y ***sat*** S one needs a set of sound and complete proof rules. They depend on the semantics of **Spec** and **Schema** and have, of course to be compatible with the relation ***has***.

Index of definitions