Presented at the 3rd International Conference on Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany (March/April 2000) and printed in its proceedings: *"Fundamental Approaches to Software Engineering"* (edited by Tom Maibaum), LNCS **1783**, 163–177 (2000).

What is '*Mathematicalness*' in Software Engineering?
— Towards Precision Software Engineering —

Hidetaka Kondoh

Systems Development Laboratory, Hitachi, Ltd.
1099 Ohzenji, Asao, Kawasaki 215-0013, Japan
kondoh@memeber.ams.org

1 Introduction

What kind of human activities do theories (*theoretical computer science*) and practices (*industrial software development*) on software most resemble? Hoare and He (1998) adopted the following viewpoint; the relationship between theoretical computer science and real-world software development corresponds to that of physical sciences and classical fields of engineering, *e.g.* mechanical engineering or aero-dynamical engineering. From this viewpoint, they successfully showed an approach towards a unified theory intended for industrial programming.

Here, we take a distinct viewpoint because of the difference between *underlying logics* of each scientific field, *i.e.* physical sciences and theoretical computer science. The underlying logic of physical sciences is *inductive* in principle, and this inductive nature of physical theories is inevitable since the physical world is given to us independent of ourselves. In this case, we must formulate properties of the physical world and then develop physical theories via comparisons between theoretical predictions and experimental results.

On the other hand, a computer program is just an implementation of some computable function which is defined in a *deductive* formal system. Such *deductive* underlying logic governing software is quite an important character of this field, hence it must be taken into account when developing software. We therefore take an approach different from that of Hoare and He.

Our approach is this: the relationship between theories and practices on software corresponds to that of mathematical logic and *living* mathematics (which we call '*ordinary mathematics*') done by *working* mathematicians (in the sense of the title of Mac Lane (1971)) as their daily work. We therefore claim that software engineers should develop their products just like working mathematicians develop mathematics; *i.e.*, software should be developed *really mathematically*.

Now most formal methods claim to be 'mathematical'; but *are they really mathematical?* Are they truly desirable approach to mathematical software development? Such a question, whether one thing is 'mathematical' or not, may seem very subjective and a matter of taste. We, however, claim that this question can be answered objectively as far as from the viewpoint of 'mathematicalness' as stated above, *i.e.* to define 'to be mathematical' as *cultural properties commonly kept by working mathematicians*. At this point, we only point out that such working mathematicians' culture has been successfully stable for sufficiently long time (at least from Hilbert's proposal of the axiomatism in the early of this century) and, hence, that our definition of mathematicalness is not arbitrary.

According to the above definition of mathematicalness, we reconsider the basic question, whether formal methods are mathematical or not, and will answer "NO", and explain why formal methods are not actually mathematical from the viewpoint of working mathematicians.

To answer this question, we will analyze activities of working mathematicians and will reveal several 'ideas' (actually some kind of *misconceptions* we think), which unfortunately prevent formal methods from becoming widely accepted in the field of industrial software even though such ideas are commonly considered to be useful or even strong points of formal methods. In Section 2, we will discuss the differences between 'to be mathematical' from the viewpoint of working mathematicians and 'to be logical' in the sense of mathematical logic on which formal methods are based. In Section 3, we will analyze the differences between mathematicians' proofs and formal proofs of formal methods.

After discussing those misconceptions, in Section 4, we will show the similarity between software and mathematics; *i.e.*, the correspondence between macro structures found in software development (*e.g.*, architectural patterns, design patterns, *etc.*) and macro structures found in mathematical arguments. Then we propose a novel concept of *mathematical* software engineering, which we call '*Precision Software Engineering*', on the basis of this correspondence between software and mathematics. Finally, we will point out remaining themes necessary for realizing this concept as an industrially applicable discipline so that future software engineering will become truly mathematical and produce reliable software.

2 Is '*Logicalness*' $\Longleftrightarrow$ '*Mathematicalness*'?

In this section, we analyze the difference in the viewpoints of working mathematicians and mathematical logicians, and we evaluate whether formal methods are actually mathematical or not.

Each formal method is based on some formal system such as axiomatic set theory, a system of modal or temporal logics, *etc.* These formal systems are originated from mathematical logic or theoretical computer science, which is applied mathematical logic. Hence, formal methods are *(mathematico-) logical*. It seems to us that, on the basis of this fact, most formal methods claim to be 'mathematical'.

It is, however, not necessarily true that a thing is mathematical if and only if it is logical. One direction (mathematicalness implies logicalness) is clearly true. This is because mathematics is a deductive language; *i.e.*, all the notions and the statements in mathematics are formalizable in a logical system, say axiomatic set theory.

The other direction (mathematico-logicalness implies mathematicalness) is, however, much more problematic. This implication would have been trivially true if we had taken the mathematical logicians' viewpoint of mathematics. But as we stated in §1, we adopted the working mathematicians' viewpoint as the definition of 'to be mathematical'. Hence, we must carefully analyze what 'mathematical' means in the working mathematicians' community.

There are many criticisms against mathematical logic. These are given by working mathematicians (*e.g.*, Jean Dieudonné (1982)). These criticisms are mainly due to working mathematicians' misunderstandings about and prejudices against mathematical logic. Mathematical logic is clearly an important part of our culture just like mathematics is.

We, however, think that these criticisms must be reconsidered more seriously so that we can learn some lessons from them by carefully analyzing why working mathematicians display such hostility against mathematical logic. Such criticisms show the following facts: (1) most working mathematicians do not regard mathematical logic as *the logic for ordinary mathematics*; (2) mathematics analyzed from the mathematico-logical viewpoint is *irrelevant* to ordinary mathematics which working mathematicians love and develop.

Why does such an unfortunate gap between mathematical logicians and working mathematicians occur? This gap is due to the nature of mathematical logic; *i.e.*, its complete *reductionistic* nature. For example, consider the following set of formulae:

$$\begin{gathered}\forall x.\, x^{-1} \cdot x = e, \\ \forall x.\, e \cdot x = x, \\ \forall x, y, z.\, x \cdot (y \cdot z) = (x \cdot y) \cdot z.\end{gathered}$$

These are very familiar axioms of groups and most working mathematicians consider this mathematical structure very important.

From the mathematico-logical viewpoint, however, these formulae are no more than sentences of first-order equational logic with one constant symbol and two function symbols (one is unary, the other binary). That is, mathematical logic cannot explain nor even pay any attention to the importance of these axioms in ordinary mathematics.

Just like this example of the group structure, mathematical logic reduces any mathematical structures to formulae in a formal system and also reduces proof techniques based on such structures to combinations of primitive inference rules of this formal system by discarding their *pragmatic* significance for working mathematicians. This is because the main purpose of mathematical logic is to analyze logical but *not* mathematical structures of mathematical arguments.

On the other hand, working mathematicians consider mathematical structures very useful and important, and also regard that proof techniques based on such structures indispensable in developing ordinary mathematics as their daily work (*cf. e.g.*, Mac Lane (1986)). *E.g.*, in most books on mathematics, the group structure is frequently used in developing mathematical theories.

In other words, mathematical logic, especially proof theory, reduces all the mathematical contents into *meaningless* syntax in a formal system and then analyzes such syntactic objects to obtain some information, say logical complexities of mathematical notions. We call such a nature of mathematical logic *reductionistic*. From such a viewpoint, all intuition (or *semantics* in a very vague sense) on mathematical objects held by working mathematicians are completely abandoned. Every mathematical theory is reduced to a collection of purely syntactic phrases and the liveness of such a theory on the working mathematicians' *Platonic world* is completely lost.

Criticisms to mathematical logic raised by working mathematicians can be summarized like this: *"mathematical logic neither helps our imagination in the* living *mathematical world nor gives any hints for proving theorems about our mathematical world; the logical structure shown by mathematical logic has hardly any relation to our* 'logic' *of living ordinary mathematics."*

Moreover, mathematical logicians do their daily work (*i.e.*, research on mathematical logic) not mathematico-logically but mathematically. That is, they live in their own world of imaginations.

Hence, *mathematical logicians themselves are actually working mathematicians!* They are working mathematicians in fields different from those of ordinary mathematics. But there is an essential difference between mathematical logicians and ordinary working mathematicians. Their viewpoints on ordinary mathematics are quite different. Mathematical logicians observe ordinary mathematics from the meta-level (in the sense of metamathematics) so view it *syntactically*, while working mathematicians live in the world of this ordinary mathematics: *i.e.*, ordinary mathematics is at the object-level for them; hence, they view it *semantically*.

Software formalists (we hereafter denote 'researchers on formal verification' by this term) often claim that one of strong points of formal methods is to allow software engineers to reason the correctness of software purely syntactically. Such a claim clearly shows that software formalists view software development from the meta-level. This viewpoint is quite similar to the mathematical logicians' one on ordinary mathematics but is *never* the working mathematicians' one on it nor the logicians' one on logic itself.

From these observations, we answer "NO" to the basic question whether formal methods are mathematical. Formal methods force their users to work in a very strictly defined formal syntax just like mathematical logic analyzes ordinary mathematics. Working mathematicians, on the other hand, think semantically and do their daily work in a natural language in a very rigorous manner with minimal use of formal syntax (*i.e.*, mathematical formulae) only when using such formalism is more compact and clearer to express their mathematical ideas than expressing such ideas in a natural language.

Any formal syntax has a strong tendency to force its users to work syntactically rather than semantically, even though syntactical manipulations are much more inefficient than working with semantical imaginations. In the next section, we will observe the fact that mathematicians' productivity is unbelievablly high when we compare it to that of software engineers. One of keys of their high productivity is that they are thinking semantically rather than syntactically about mathematical objects. In the next section, we will reveal the secret how mathematicians attain such high productivity.

3 Are Formal Proofs Imperative?

NOTICE: Hereafter, the term 'mathematics' will be used in a broader sense than in §2; *i.e.*, as the generic name for all the deductive sciences including mathematical logic and theoretical computer science as well as (ordinary) mathematics. The term '(working) mathematicians' therefore denotes researchers in all such fields. When we use these terms in the narrower sense as in §2, we affix an adjective 'ordinary'.

Many formal methods are supported by (semi-)automatic provers or proof-checkers. Users of such methods can verify the consistency of their specifications and the correctness of programs with respect to specifications completely formally supported by such provers/proof-checkers running on a computer. Hence, those formal methods claim that such possibility of formal verifications is one of strong points of themselves. Is this possibility really a strong point? Are such formal verifications compatible with reasonable productivity of software?

Now we estimate working mathematicians' productivity and compare it with the productivity of software engineers. It is very difficult to estimate the amount of *semantical contents* in a mathematical paper and almost impossible to compare such an amount with the amount of a program. It is, however, possible to estimate the amount of *syntactical objects* denoting such semantical contents. In estimating such syntactical amount, the next example will give some insights. Let X be a set, $\sqsubseteq$ and $\leq$ be partial orders on X, and $\equiv$ and $\simeq$ be equivalence relations induced by these partial orders, respectively. Suppose $x \sqsubseteq y \Rightarrow x \leq y$ for all $x, y \in X$. Then, it is trivial that $x \equiv y \Rightarrow x \simeq y$ for all $x, y \in X$. But if we want to prove this trivial theorem formally (in, say Gentzen's NK) we need more than 10 steps. It is well known from constructive programming that every formal proof step exactly corresponds to a step of a program.

We can therefore roughly estimate the amount of a program-like formal object denoting the contents in a mathematical paper by multiplying its number of lines by 10 or more (if proofs of that paper contain large 'gaps' which novices cannot fill in, then 1,000 or even more would be appropriate) to obtain the corresponding LOC (lines of code) as software. This means a 20-page paper (it is not so long as a mathematical paper) with 30 lines per page corresponds to at least 6,000 LOC as a program. In fact, this estimation may be too small since most mathematical papers are very technical and contain many 'gaps' which only professionals can fill in, so we should estimate the amount of contents of such a paper as at least 60,000 LOC (here, still a rather small scaling factor 100 is used). Even if a mathematician produces only one paper per year, his productivity about the mathematical contents corresponds to 5,000 LOC per month!

This would be surprising productivity if he were a software engineer. At first sight, this comparison may seem to be unfair, but it is not so unfair. Suppose there were a *formal mathematician* who do mathematics completely formally (perhaps using some formal-proof development tools). Note that his aim is not to record existing mathematical theories formally but to develop his own novel mathematical theory just like usual mathematicians do. Then, such a formal mathematician had to have his productivity as high as the above LOC value in order that he could write a paper (per year) containing equivalent mathematical contents to the above (*non-formal but rigorous*) mathematician's one. We should aware that activities of such formal mathematicians are almost the same as those of software engineers. In fact, they can be said '*programmers in mathematics*'.

The fact that the above comparison seems unfair shows an important point. That is, *software engineers have been handicapped already compared with mathematicians.* Software engineers' handicap is that their final products must be completely formal, computer programs. Most of software engineers do not think formally at the current state-of-art, but their productivity has been significantly reduced already in order to produce formal objects. In the case of mathematics, if there were formal mathematicians, they would be handicapped, too. Therefore, the productivity 5,000 LOC per month required for such formal mathematicians sounds unrealistic, and such unreality is solely due to their way of working, *i.e.* syntactical manipulation instead of semantical imagination, because mathematical (*i.e.*, semantical) contents of formal mathematicians' products are completely the same as those of ordinary mathematicians' ones.

To understand such formal mathematicians' handicaps, we should remind that in mathematical papers there are many 'gaps' which only very trained people can fill in. In fact, mathematicians attain their surprisingly high productivity mainly because they do not prove their theorems formally. They leave some 'gaps' in proofs unfilled, and such 'gaps' are actually not true gaps (*i.e.*, flaws of proofs). They are *trivial* for intended readers of such papers. Mathematicians do not want to spend their time filling in such 'trivial gaps' completely.

Formal proofs (say, of the correctness of software) do not allow the smallest 'gaps', even if such 'gaps' are actually trivial for software engineers. Proving formally decreases productivity of software engineers from that with semantical thinking to the level of syntactical thinking; *i.e.*, programming. The secret of mathematicians' surprising productivity is thus: they limit use of formal languages in describing and proving mathematical properties as little as possible and then keep their productive semantical imagination as much as possible.

Note that *no mathematical logicians, theoretical computer scientists, nor software formalists have ever written a paper in which their theorems are proved formally.* That is, any researchers on formal systems do their daily work mathematically (in the sense of §2) but NEVER formally.

If software engineers are requested to prove the correctness of their programs complete formally, then they will be doubly handicapped. They must not only produce completely formal products (programs) but also work with such products completely formally. Then software engineers' performance in provings will surely be decreased from the level of mathematicians' performance in provings to that of formal mathematicians' one.

Software engineers' mission is to produce software with reasonable reliability and performance within a given budget. For software engineers, activities such as designing software and proving its correctness are their daily work; these activities just correspond to stating theorems about mathematical objects and proving them for working mathematicians, or stating metatheorems about properties of formal systems and proving such metatheorems for software formalists. *It is quite unfair to request that only software engineers must prove the correctness of their products formally.* Though it is true that the main part of a software engineer's product is a formal object (*i.e.*, a computer program), such a formal artefact is produced in the very final step of their daily work. A correctness proof of a program is only a way to guarantee the quality of their product. For software engineers, a correctness proof is not their ends but a mean of quality assurance.

The following fact is empirically well known among working mathematicians: in a seminar, if a theorem is stated and a sketch of its proof is shown and also if most of the participants in that seminar think that it is correct, then *the statement of the theorem itself is true in most cases* even if the original proof sketch may contain serious flaws when a proof is written in detail. In verifying the correctness of software, 'a theorem' above corresponds to the correctness. For software engineers, the correctness of software is very important but achieving absolutely correct correctness proofs lies outside their principal aim.

Most formal verifications are methods that directly guarantee *the correctness of correctness proofs* and then inform us of the correctness of software. As mentioned above, this approach is outside software engineers' principal aim. To remedy this problem, the notion of 'verification' in the Cleanroom Method, Linger et al. (1979), gives a good hint. In this method, 'correctness proofs' are planned to be done rigorously (but never formally) as verification-reviews by a small design team consisting of several engineers. This approach to correctness proofs is quite similar to checking proof sketches by participants in a seminar of working mathematicians. Hence, the notion of and the approach to verification in the Cleanroom Method, in its spirit, quite resemble the daily activities of working mathematicians. This method has been successfully applied to real projects. Overall productivity is reported to be the same or improved compared with that in conventional (without any correctness proofs) developments even though correctness are proved in this method, *cf.* Gibson (1997).

The Cleanroom Method verification is limited to the scale of individual procedure and cannot be applied to larger scales. In the next section, keeping the spirit of this method (*i.e.*, focusing the correctness of software but never that of correctness proofs), we will show an approach to mathematical software development by comparing *macro structures* in software and mathematics.

4 The Software-Mathematics Structural Correspondence

In this section, we analyze the structural resemblance between software and mathematics by focusing their macro structures.

Constructive Programming is an approach to systematically derive programs from their specifications written in some version of intuitionistic (*i.e.*, constructive) logics, say a higher-order intuitionistic type theory, instead of the usual classical (*i.e.*, two-valued) logic. This approach is based on the following de Bruijn-Curry-Howard correspondence between programming and logic; *cf.* Nordström et al. (1990), Luo (1994), and Hindley (1997):

Table 1. The de Bruijn-Curry-Howard Correspondence

Programming	**Logic**
Specification	Proposition
Program	Proof

The problem of the constructive programming approach is its too *microscopic* and *reductionistic* viewpoint. There are many textbook on the same mathematical theme, but we say that some are well-written and some are poor even if later ones had no gaps in proofs and no typos. If these books are considered formally then they are equivalent, since each book gives the same main theorem and shows its proof. The difference between such well-written books and poor ones lies in the style of their presentation.

Well-written books show intuitively clear and natural definitions capturing essential notions adequately, provide many useful lemmata, and prove the main theorem in a very beautiful and reusable way. Just like this, a program at the programming-in-larges level, which is so large that it needs software engineering, is expected to have a beautiful internal structure consisting of understandable and stable-against-modification components in order to decrease efforts in its maintenance.

The constructive programming approach has another problem, the difficulty of formal proofs as we discussed in §3. Therefore, we must admit that this approach is quite difficult to be applied to industrial software development.

We, however, think that this constructive programming approach has its own interests; *i.e.*, this approach gives hints to consider an analogy between programming and mathematics/logic. In the constructive programming approach, this analogy is quite formal but stays at the *programming-in-smalls* level. If we relax this analogy to be informal and extend it to a more macro scale, the *programming-in-larges* level, then we can find a correspondence between structures found in software development and those used in mathematical activities at various levels of granularity. This correspondence is shown in Table 2. We now analyze this *Software-Mathematics Structural Correspondence* more carefully. Note that the base of our analysis is the correspondence shown in Table 1.

Table 2. The Structural Correspondence between Software and Mathematics

Software Development	**Mathematical Activity**
Basic Control Structure (Repetitive Loop, Conditional, ... *etc.*)	Elementary Proof Step (Mathematical Induction, Case Analysis, ... *etc.*)
Idiom (Useful Combination of Control Structures)	Proof Technique (Conventional Technique for Fragmental Proof)
Abstract Data Type (Collection of Operations on Common Data)	Theory on a Mathematical Notion (Collection of Lemmata on a Mathematical Notion)
Design Pattern (Specific Combination of (Possible) Classes and Specific use of their Interdependencies)	Proof Tactics (Specific Combination of Subgoals (Lemmata) and Specific use of their Interdependencies)
Architectural Pattern (Specific Combination of Specification of Components)	Theory Strategy (Collection of Basic Definitions and the Main Theorem)
Software System	Mathematical Theory (Structure formed by Definitions, Theorems & Proofs)
Domain	Mathematical Field (Basic Framework for Mathematical Thinking with Common Vocabulary)

- **Basic Constrol Structure *vs.* Elementary Proof Step**

The lowest level of the correspondence, between a *basic control structure* and an *elementary proof step*, is essentially formal in the sense of the constructive programming approach. *E.g.*, a sequential composition corresponds to the cut rule in logic, a conditional corresponds to the disjunction elimination in logic, and a terminating repetition corresponds to an induction on a well-founded ordering.

- **Idiom *vs.* Proof Technique**

An *idiom* in programming is a specific pattern of combination of basic control structures. As stated above, control structures correspond to elementary proof steps. Hence, an idiom corresponds to a specific pattern of combination of elementary proof steps in mathematics. Such a pattern is a *proof technique*; *e.g.*, Fermat's method of 'infinite descent', Reid (1988), (often used in elementary number theory) is a specific combination of a *reductio ad absurdum* and a mathematical induction.

- **Abstract Data Type *vs.* Theory on a Mathematical Notion**

An *abstract data type* (or a *class* in the object-oriented paradigm) is a collection of operations acting on a common data to be encapsulated. As we have seen in Table 1, a specification of an operation corresponds to a proposition. Hence, a specification of an abstract data type (or a class) corresponds to a (usually miniature) *theory on a specific mathematical notion* and an implementation of an abstract data type (or a class) is in correspondence with a proof of such a theory. Most typical example of such a theory on a mathematical notion is group theory (though it is not miniature). This correspondence is well-known and is a basis for the algebraic approach to abstract data types, Ehrig and Mahr (1985).

- **Design Pattern *vs.* Proof Tactics**

A *design pattern* is a specific combination of classes and specific dependencies (inheritance relations) among them. As shown above, classes correspond to theories of mathematical notions. In mathematics, a *proof tactics* is a collection of conventional knowledge on matters such as what lemmata should be stated in order to prove the desired theorem and how to use dependencies between theorems. In many cases in mathematical theories, those lemmata form a miniature theory on an auxiliary mathematical notion which is used to develop the proof of an upper-level proposition. Hence, a design pattern corresponds to a proof tactics in mathematics.

• **Software System *vs.* Mathematical Theory**

Before considering the relationship between architectural patterns and theory strategies, we analyze the next pair (*i.e.*, software systems and mathematical theories) since each of the former pair is a skeleton of corresponding one in the latter pair; hence, the discussion on the second pair is expected to be more intuitive than that on the first one.

A *software system* is developed according to its specification, has some interface with its environment, and is implemented as a collection of various components interacting one another; each component has its own specification and interfaces to other components. At the side of mathematics, by a *mathematical theory*, we mean the contents of a mathematical book or a paper. It consists of a collection of definitions and theorems (some of them are *main theorems*, for which the book or the paper is written) and their proofs. Of course, in order to show main theorems, many auxiliary definitions, lemmata and their proofs are necessary. Such organic collection of these mathematical items form a mathematical theory. This just corresponds to a non-trivial programming-in-larges software system as follows:

- the (functional) specification of a whole software system corresponds to statements of main theorems as a whole (by taking their conjunction);
- the specification of a component corresponds to the statement of a lemma which is used to prove main theorems;
- the implementation of this component corresponds to the proof of this lemma;
- a component, which is provided as a *library* and is outside of the development project of the system, corresponds to an 'external' lemma whose statement is borrowed from some reference and is used without a proof;
- (definitions of) mathematical notions for describing the statement of main theorems corresponds to (definitions of) abstract data types used to give the functional specification of the software system;
- (definitions of) auxiliary notions used to describe statements of lemmata are in correspondence with (definitions of) abstract data types for interfaces among components.

Note that a mathematical monograph contains quite a lot of materials; *e.g.*, Barendregt's "*The Lambda Calculus*" (1981) has the contents as much as *ca.* 400,000 LOC (> 600 pages × 30 lines/page × 20 (as a scaling factor); in this case, the author estimates the scaling factor to be a very small value 20, since that book hardly contains any 'gaps' in proofs), which is large enough as non-trivial programming-in-larges software.

- **Architectural Pattern *vs.* Theory Strategy**

An *architectural pattern* is a specific pattern of a collection of specifications of components, interfaces among these components and an interface with an external environment for the system which is expected to build with this architectural pattern. Each component is given a *partial* specification; *i.e.*, this specification is so weak that it cannot uniquely define the external function of the component. An architectural pattern therefore is a template for constructing software systems.

Now consider the mathematical side. A *strategy for developing a mathematical theory* is characterized as its starting point (basic definitions), its pattern of the goal (*i.e.*, the pattern of main theorem(s)), and patterns of several important auxiliary lemmata. For example, in λ-calculus or rewriting systems alike, if we want to show the Church-Rosser Theorem (*i.e.*, the congruence of reductions), then we have several choices. One of them is to show it via the Hindley-Rosen Diamond Lemma. Another is to use parallel reduction. Each of them gives a strategy for developing the reduction theory of such a calculus. In this example, the goal, the concrete statement of the Church-Rosser Theorem, depends on each reduction system; hence, the Church-Rosser Theorem is the name of a family of *similar* statements. Just like this, each of the Hindley-Rosen Lemma and the notion of parallel reduction is a family of similar statements and that of similar notions, respectively. As we have seen in this example, a theory strategy is a template to build many concrete mathematical theories; in this case, each strategy (*i.e.*, via the Hindley-Rosen Lemma or via parallel reduction) is a template to construct a concrete rewriting theory of various rewriting systems. As we have shown the correspondence between software systems and mathematical theories, we can conclude that architectural patterns and theory strategies corresponds to each other.

- **Domain *vs.* Mathematical Field**

A *domain* has its own basic vocabulary to describe problems and each term in such vocabulary carries its own specific meaning and restrictions for its use. Similarly, a *mathematical field* defines a basic framework and a common vocabulary for mathematical thinking. For example, it is often said by working mathematicians that algebraists think with the equality, $=$, while analysts use the inequality, $\leq$. This means that $=$ is a basic notion in algebra, but it is not the case in analysis. In analysis, $=$ is a compound notion shown by a pair of two $\leq$'s ($x = y$ from $x \leq y$ and $y \leq x$) by taking limits of appropriate sequences. To show another example, let's see what the field of λ-calculus is. When we read Barendregt's encyclopedic monograph (1981), it tells us that we should investigate such a calculus on four aspects: *i.e.*, conversion relations, reduction relations, equational theories, and models. These are the most basic vocabulary in λ-calculus and give the most basic thinking-framework in developing similar various calculi.

Differences between Software Engineers and Working Mathematicians

We have seen that macro structures known in software engineering well correspond to macro structures in mathematics. *There is, however, an essential dif-*

ference between software engineering and mathematics in handling such macro structures. Mathematicians are very rigorous in using their mathematical macro structures. They very carefully analyze limitations of usage of these structures: *e.g.*, under what condition a lemma can be used, what additional conditions are needed for a mathematical construction to preserve a desirable property, *etc.* Software engineers treat their macro structures such as architectural patterns, Buschmann et al. (1996), and design patterns, Gamma et al. (1995), in very intuitive and empirical manners. They have found such reusable structures but have never analyzed their rigorous properties.

A design pattern can be said a structure with several holes, each of which is expected to be filled by a class or an object. Current works on design patterns, however, do not analyze necessary properties for such holes: *e.g.*, what a class invariant is necessary for a class to fill a specific hole of the pattern; what preconditions/postconditions are expected for a specific method supported by a hole-filling class. In the case of an architectural pattern, it is a macro structure with holes for large components and interactions among them. We should therefore know rigorously about necessary properties of each component to fill each hole. We also need to know about under what conditions additional properties of these components are preserved all over the architecture. For example, what additional conditions are necessary to extend the deadlock-freeness of each component to this architecture as a whole. Without knowledge about properties of these macro structures, software engineers either cannot safely reuse them in building reliable software systems or must prove the correctness of the software from scratch every time they use macro structures.

There are several works on formal descriptions of such software macro structures. *Most of them, however, are at the level of description.* They are not at the level of analysis of their properties. Careful analysis and accumulating such rigorous properties are essential and far more important than describing structures formally, because if we knew their properties rigorously, we could correctly use those macro structures in constructing software systems with high reliability and could prove its correctness efficiently on the basis of such rigorous properties.

5 Towards Precision Software Engineering

So far, we have discussed how we can make software engineering truly mathematical. The key idea is the use of macro structures such as architectural patterns and design patters with their own rigorous theories.

In other words, software engineering must not only describe the structure itself (with, say, UML or some kind of architecture description languages) but also develop a rigorous miniature theory (*mini-theory* for short) for each software macro structure, which has holes to be filled (instantiated) by some components. Each such mini-theory must clarify rigorous properties of each structure: *e.g.*, invariants of the structure, relationships among invariants of components to fill holes of that structure, properties preserved by that structure and additional condition (if necessary) for this preservation. Such properties on a macro structure form an *ad hoc* (in the sense of mathematical logic) but useful mini-theory ('theory' exactly in the sense of mathematical theories) on this structure. These mini-theories can be used in proving correctness of software systems built up with those structures. Then software engineers can *safely* use such macro structures in building reliable software and rigorously proving its correctness *efficiently*. This situation is completely parallel to the situation in mathematics: *i.e.*, working mathematicians first empirically found useful mathematical structures such as groups, rings, and topological spaces; then, for such structures, mathematicians developed *ad hoc* but rich theories such as group theory, general topology, *etc.*; and now, they can effectively use such theories on mathematical structures in developing their own mathematical theories by proving their own theorems efficiently on the basis of such theories in order to output their *products*; namely, mathematical papers and books.

We call such an truly mathematical discipline of software engineering as *Precision Software Engineering.* This discipline must consists of *ad hoc* mini-theories on *empirically useful* software macro structures just like mathematics consists of theories on mathematical structures like groups, rings, *etc.* which are *ad hoc* from the mathematico-logical viewpoint. There are much materials, namely generic meta-theories such as many variations of programming logics and calculi for concurrency, provided by theoretical computer science (including researches on formal methods), which are applicable to build up such mini-theories for Precision Software Engineering.

The most important difference between Precision Software Engineering and formal methods is the difference in viewpoints and not that of meta-theories on which they are based. Most formal methods aim to formally *describe* software with a language (as a concretization of some generic meta-theories) and to prove the correctness of software formally. On the other hand, Precision Software Engineering stresses to rigorously *analyze* software macro structures and to construct their own *logically ad hoc* (but *logically consistent*, of course) and *practically useful* mini-theories on reusable such structures on the basis of generic meta-theories. This approach to software engineering gives a good interface between software engineering and theoretical computer science. That is, the latter provides *generic* meta-theories on the nature of computation, while the former develops *ad hoc* mini-theories on software macro structures and methodologies to work with such mini-theories founded on generic meta-theories.

Precision Software Engineering is not already-established discipline, but there are several interesting works which share the spirit of Precision Software Engineering. The most significant and already practical one is the Cleanroom Method, which captured the essential spirit of mathematical proof and applied it correctness proofs of programs, *cf.* Linger et al. (1979). Bertrand Meyer's Design by Contract is also a practical approach to program correctness on the basis of an interesting analogy that the precondition and the postcondition of a procedure can be seen as the contract of the provider and users of the procedure, and this approach covers the object-oriented paradigm, *cf.* Meyer (1997).

Another very ambitious and quite interesting works on more macro-scale structures is Dines Bjørner's Domain Theory (1997 and 1998), where Bjørner tries to develop theories on various application domains of software by describing each application domain by a formal specification language RSL to standardize the vocabulary of the domain, then analyzing properties of each domain very rigorously (in fact, formally) and finally obtaining its properties (invariants of the application domain and many useful lemmata) in the form of formal sentences in RSL.

Yet another interesting rigourous approach on macro-scale structures is Bjørner et al. (1997), which characterizes Michael Jackson's (originally not so rigourous) Problem Frames (1995) in software development; and the notion of the Problem Frame, in turn, was inspired from G. Polya's classical work(1957) on the classification of mathematical problems for solving them.

Also interesting works are on formal definitions of practically useful design/modeling languages like UML, so-called Precise Semantics of Software Modeling Techniques: *e.g.*, Broy et al. (1998). If UML have such formal foundations and properties and limitations of various design transformations on that languages are analyzed rigorously, then working software engineers can effectively work with UML in a very rigorous and safe manner.

There have been many proposals to make programming mathematical. Among others, Hoare (1986) clearly showed the correspondence between programming and mathematics. Those proposals are, however, at the programming level; *i.e.*, within the programming-in-smalls scope and actually from the mathematico-logical viewpoint in stead of working mathematicians' one. The novelty of our proposal is its scope. Our approach offers a way for more upper design processes in software development to become mathematical on the basis of the Software-Mathematics Correspondence shown in Table 2, which we have found.

As a (mainly academic) research field, Precision Software Engineering may also be called *Abstract Software Engineering* just in the sense of *abstract* algebra. In fact, the principal aim of Precision Software Engineering is to establish *abstract (mini-)theories* of empirically useful software macro structures independent from how those structures are concretely implemented (in C, Java, *etc.*). This is quite analogous how abstract algebra was born. For example, groups were originally found by Galois as very concrete objects; *i.e.*, substitutions of solutions of algebraic equations. Later, Noether and other mathematicians abstracted and purified the notion of groups and established modern abstract group theory. Other branches of abstract algebra were also emerged by abstracting originally very concrete mathematical objects (*i.e.*, various number systems like $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{C}$). Like this, what are necessary in current complicated software development is abstract theories of useful design patterns, software architectures and so on. Therefore, when both of Abstract Software Engineering as a research field and Precision Software Engineering as an engineering discipline are established, working software engineers can efficiently develop highly reliable software, and then, *software engineering can become a truly* modern *engineering field like other conventional ones, e.g. aircraft engineering.*

Acknowledgements

The author wishes to express his deepest thanks to Professor Roger Hindley, who told the author much about logicians' viewpoint and also about historical conflicts between working mathematicians and logicians. Discussions with him form the basis of Section 2. The author also much thanks to Professor Henk Barendregt, whose enthusiasm to make programs more beautiful strongly motivated the author to think about a more rational way of software development and to initiate this work. The author wishes to express his sincere gratitudes for Professor Dines Bjørner, who kindly gave the author's group a lecture on his Domain Theory, and he also gave the author several comments on a draft of this paper, which inspired and encouraged the author very much. The author also owes some of inspiration appeared in this work to Professor Tony Hoare (as well as his book) whose interests and comments on a draft of this work also encouraged the author so much. Constructive criticisms given by Mr. Hirokazu Yatsu, Professor Tetso Tamai, Professor Motoshi Saeki and Dr. Kazuhito Ohmaki much helped the author in clarifying the notion of Precision Software Engineering. Last but not least, the author wishes to thank Professor Kenroku Nogi, Mr. Shigeru Otsuki, and Mr. Hirokazu Tanabe for their encouragements and warm advices which were very helpful for the author to improve the current work.

References

Barendregt, H. P. (1981): *The Lambda Calculus*, North-Holland.
Bjørner, D. (1997): *UNU/IIST's Software Technology R&D in Africa, Asia, Eastern Europe, and Latin America: 'Five' Years — a Personal View*, UNU/IIST Report No. 90.
Bjørner, D. (1998): *Domains & Requirements, Software Architecture & Program Organization*, Full Day Seminar in IFIP '98, Budapest.
Bjørner, D., et al. (1997): Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools, in *Proceedings of 1st ICFEM*, 263–270.
Broy, M., et al. (1998): *PSMT — Workshop on Precise Semantics of Software Modeling Techniques*, Tech. Report of TUM I9803.
Buschmann, F., et al. (1996): *A System of Patterns*, John Wiley & Sons.
Dieudonné, J. (1982): in *Penser les mathématiques*, Editions du Seuil.
Ehrig, H. and B. Mahr (1985): *Fundamentals of Algebraic Specification 1*, Springer-Verlag.
Gamma, E., et al. (1995): *Design Patterns*, Addison-Wesley.
Gibson, R. (1997): *Cleanroom Software Engineering Practice* (by S. A. Baker et al.), 116–134, Idea Group Publishing.
Hoare, C. A. R. (1986): *The Mathematics of Programming*, Clarendon Press, Oxford.
Hoare, C. A. R., and He, J. (1998): *Unifying Theories of Programming*, Prentice Hall.
Jackson, M. (1995): *Software Requirements & Specification*, Addison-Wesley.
Linger, R. C., et al. (1979): *Structured Programming*, Addison-Wesley.
Luo, Z. (1994): *Computation and Reasoning*, Clarendon Press, Oxford.
Mac Lane, S. (1971): *Categories for the Working Mathematician*, Springer-Verlag.
Mac Lane, S. (1986): *Mathematics: Form and Function*, Springer-Verlag.
Meyer, B. (1997): *Object-Oriented Software Construction*, 2nd ed., Prentice Hall.
Nordström, B., et al. (1990): *Programming in Martin-Löf's Type Theory*, Clarendon Press, Oxford.
Polya, G. (1957): *How to Solve It*, 2nd ed, Princeton Univ. Press.
Reid, M. (1988): *Undergraduate Algebraic Geometry*, Cambridge Univ. Press.
Shaw, M. and Garlan, D. (1996): *Software Architecture*, Prentice Hall.