On the Role of Logic and Algebra in Software Engineering

Manfred Broy

Abstract Software engineering is a field of high relevance for many practical areas of advanced technology. It is essential also for a number of safety critical systems and technical infrastructures. Stimulated by the exponential growth of the power and speed of electronic hardware we observe an exponential growth in the functionality, the size, and complexity of software. In contrast to electronic hardware where we expect that everything is built based on carefully investigated theories with a deep scientific understanding, software creation still is to a large extend ad hoc. Nevertheless, due to the rising quality demands for software and the necessary improvement of productivity by advanced tools we see a growing need for a proper foundation and comprehensive theory of software engineering. In the following we outline the role that we see for logic and algebra in the emerging field of software engineering.

1 On Software Engineering

Today software is everywhere, although not always visible. More than 98 % of all programmable processors run in embedded systems. Software provides an essential part of the technical and organisational infrastructure of our world today. In software intensive business areas companies might get bankrupt within a few days or even within a few hours if their software systems are out of operation. A substantial number of people interact directly or indirectly with computers most of their working time. The reactions and man–machine-interfaces of the devices of today are essentially influenced by the concepts underlying software systems.

M. Broy (✉)
Fakultaet fuer Informatik Technische Universitaet Muenchen,
Boltzmannstr. 3 D-85748 Garching, Germany
e-mail: broy@in.tum.de; http://wwwbroy.informatik.tu-muenchen.de

P. Paule (ed.), *Mathematics, Computer Science and Logic - A Never Ending Story*,
DOI 10.1007/978-3-319-00966-7_2,

Building software is a complex and error-prone task. Software engineering is the discipline of the development of large, powerful software systems optimising their quality, costs, and delivery schedule.

Advances in software engineering has to be based on a broad scientific and practical view and thus comprises many aspects requires a spectrum of skills. This is one reason why it is very difficult to gain measurable progress in software engineering.

In fact, researchers often tend to concentrate only on particular aspects of software engineering. However, as long as one is just concentrating on a strictly monocausal view onto software engineering such an approach is condemned to failure and will not improve the field in a substantial way. Among the many aspects the following points are of particular importance for software engineering practise.

- Economy and costs
- People, and their skills
- Development process
- Documentation – modelling & programming languages
- Application domain
- Usability – user requirements and man machine interface
- Implementation quality – correctness
- Technical quality – reliability
- Hardware and software infrastructure – performance
- Tool support – productivity
- Maintenance

In spite of the fact that software engineering has to look at and integrate many different aspects we concentrate in the following mainly on technical issues and, in particular, on the role of logic and algebra in software engineering. We argue that from a rigorous view software is, in particular, of mathematical nature and, if we want to get our hands on the foundations of software engineering, we have to understand the foundational role of logic and algebra.

Furthermore we argue that additional investigations into logic and algebra are mandatory to adopt these disciplines in an effective way for software engineering.

2 Technical Aspects in Software Engineering

We concentrate on technical aspects in software engineering and their foundations in the following. We do not look very carefully into issues of management, economy, and also not on the organisation of the development process but rather on technical and modelling issues. We discuss the technical and modelling issues nevertheless along the lines of the organisation of the development process.

In the following we go through the phases of software development as structured by the waterfall model where the idea is that each phase is to be finished before the next one is started. However, we want to emphasize that we choose these phases

only for structuring purposes. Actually the phases can be interleaved as well, as it is suggested in so-called agile, incremental or iterative software development processes.

2.1 Analysis Domain Engineering and Requirements Engineering

The most difficult and most decisive phase of software construction is not the implementation phase where the coding is done but the decision what functions and behaviours are to be realized and to be implemented and how to choose the overall structure of the development process and the software. An important, perhaps the most important step in software development is in fact the analysis and requirements phase (see [15]). In the analysis phase a first understanding of the problem domain and of the software requirements has to be achieved. Dealing with the first aspect we also speak of *domain engineering,* referring with the second aspect we speak of *requirements engineering*.

Domain engineering aims at the formalisation of a domain based on a logical theory. It depends very much on the particular domain a software engineer is looking at how much theory is initially available. In domain engineering we have to capture, analyse, understand, and document the essential notions, structures, and rules of an application domain. In essence, this means that we have to construct a data model and define on top of it the rules of the domain. In addition, we have to provide a number of behaviour models to document the behaviour and processes in the application domains. Domain engineering shows close relationships to logics. Logics is the discipline of formalizing structures and properties of subjects of discourse – which we call application domains – and ways as well as rules to reason about them.

A lot of work has to be done in informatics to formalize domains such that finally typical structures of computer science get visible that represent the relevant aspects of the domain, towards which then a detailed analysis and requirements can be targeted. Prominent examples are biology, chemistry or genetics. To do so, a domain has to be understood through the eyes of software engineers using their specific models of informatics. This is a fascinating process since informatics uses models that are quite different from the models that classical mathematics has introduced over the last two centuries.

Classical mathematics is mainly directed towards models in terms of continuous functions, integral and differential calculus. In fact, the models of informatics are much closer to logic and algebra. Informatics is based on discrete digital models. In contrast to the quantitative models of continuous mathematics, logic and algebra provide qualitative models that are able to capture the logical nature of domains in terms of data structures and functions on top of the data structures. This is one of the essential steps in making a field of application fit for computing, making a

domain accessible for a computer science treatment. In such a work we see at many places how the domains and the domain theories of application fields change and are deeply influenced by computer science modelling techniques, which are based on logic and algebra.

The relationship between the description techniques of informatics and software engineering and mathematical logic and algebra are much closer than it may seem at a first look. The description techniques of software engineering describe models that are essential the same as the models of logical formalisms. Even more, we may understand every description in software engineering, be it a table, a diagram or a text, as a logical formula (see [5]). This allows us to transfer helpful notions from logic such as consistency, completeness, soundness, and correctness to software engineering. Especially for tool support these notions are of major importance.

2.2 Sorting Out Requirements

The goal of requirements engineering is to identify and document all the requirements of the software system under construction. The first step is to prepare the requirements capture. In the requirements capture the basic requirements of software systems are identified and made precise. This activity is crucial since, if the requirements are captured incorrectly, then the resulting system will be useless, will not function in a correct way, will not address the users needs, or will be more expensive and more complex than necessary. Therefore, to find out what the actual requirements of an application system are is an important, difficult, and often painful and error prone process.

Software systems and their behaviour are highly abstract artefacts. For many users it is very difficult to imagine the functioning of software systems and all the consequences of their operation in advance. Therefore we need techniques to capture the requirements and show the consequences of imposing all kinds of requirements. Again logic can help here.

Typically requirements as they are collected from different stakeholders tend to be contradictory and inconsistent. Logic can help to define what consistency means, to analyse inconsistencies and give hints how they may be resolved.

In requirements engineering we find an interesting interplay between formal techniques from logic and algebra, techniques from domain modelling, psychology, and general techniques for structuring problems.

Roughly we can structure the process and activities of requirements engineering into the following major tasks:

- Requirements capture, identification, decision, and agreement,
- Requirements specification and documentation.

For both tasks logic and algebra can help. In the requirements capture phase we identify and discuss the choice of requirements and finally try to agree between the many people and roles involved in the particular requirements.

In the requirements specification phase we have to make the captured requirements precise. It is important to distinguish carefully between these two steps. One step makes sure that the requirements address the users' needs, the second step make sure that the requirements are formulated in a way such that the development engineers will understand them correctly to depart from them in the right direction and to show the correctness of their design and implementation.

Specifying and documenting requirements involves the description of complex properties. This is very similar to the classical techniques of mathematical logic. However, in contrast to classical logic, the number and size of the properties is often fairly huge. Therefore techniques for structuring and supporting easy understanding and reduced complexity are essential.

2.3 Design: Software and Architecture

In design we decompose a system into subsystems. This decomposition is called architecture. The architecture of a software system is essential for its manageability and maintainability. Designing architecture of a software system means structuring a software system under development into components in an appropriate way. Over the years we have learnt many lessons how to structure software systems appropriately. Modularity is only one issue here. Modularity means making sure that the components of a systems are self contained, described by proper interfaces such that we can construct and verify the architecture by just understanding the interfaces of the components. Moreover, in turn we can construct the components by only understanding their interfaces.

Although understood that way since many years due to the early work of David Parnas the theory and practice of software architecture is still, from a practical point of view, not satisfying. Many architectural descriptions are ad hoc, are not made precise enough. Architectural description languages are mostly not flexible enough to capture the complexity of sophisticated software systems.

In fact, a proper tractable theory for specifying interfaces even in object-oriented languages is not really available so far.

In software architecture, roughly speaking, we have to define a system in terms of two issues:

- A network of components,
- Its component interfaces.

The network describes the components of an architecture and how they are connected. The component interfaces describe the observable behaviour of the components as far as it is important for the functioning of the architecture.

The quality of an architecture is, of course, quite independent of the preciseness of its description. In fact, we have many problems to solve when selecting a software architecture including architecture description and justification.

First of all we have to make sure that the architecture addresses all the needs of a systems properly. This includes functional and non-functional requirements that have to be addressed by the architecture.

The role of logic and algebra in architectural design is first of all providing a proper theory of modularity such that we can prove that the decomposition of a system into components by their component interface specifications guarantees the functioning of the overall behaviour of the system. Although from a practical point of view this might be difficult to carry out, in principle, in practical situations, it is essential to have a theory ad hand available to support that.

A consequent step is to see an architecture essentially as a logical expression. As demonstrated in [6] the composition of components can be fully understood us logical composition. Then each component behaviour is captured by a logical formula, the parallel composition of components is represented by logical "and" (conjunction), the hiding of channels corresponds to existential quantification.

2.4 Stepwise Decomposition into Modules

Having described the components of an architecture each component has finally to be realized by code. To do that we decompose components further into modules. In principle, the decomposition of a component into modules is very similar to the decomposition of a system into an architecture. We can do that in a hierarchical top down decomposition as long as components are too big to be implemented by modules directly. Finally we will hopefully arrive at a size of components that can directly be realized by modules. In an object oriented terminology modules correspond to objects and classes.

Again the specification of modules has to be done in terms of logical and algebraic concepts. In contrast to software components that are more self-contained, in general, modules show lots of context dependencies. In fact, still a lot of work has to be done to find appropriate, tractable logical and algebraic foundations for the description of module interfaces.

2.5 Coding Modules

Finally after the hierarchical system decomposition into modules is finished we can go over to the module coding. In the module coding we use the module interface specifications as the starting points for the module coding. We have merely to understand the module interfaces and we do not have to look at the whole system. This reduces the complexity and the mutual dependencies of the implementation tasks considerably.

Starting from the module specifications, the implementation can be constructed in a strictly systematic top down process along the lines of structured programming as advocated by Hoare, Dijkstra, Wirth and Dahl and many more.

2.6 Code Verification

In the course of a software development, after the overall structure of a software system is fixed and the granularity of the decomposition of the components into subcomponents and modules is finished we start the implementation activities. When modules are implemented, an important issue is to verify that they are correctly realized according to their given interface specifications. This is called verification. Verification can actually be carried out in many ways. We can do it in a very logical style by logical deduction or by model checking but we can also use more pragmatic techniques such as inspection, reviews, or testing. Even for testing logic provides a firm basis and framework.

However, what is essential to keep in mind in verification is that none of the techniques really works effectively if we do not have a precise module specification. Especially testing gets into a mess if the interfaces are not specified properly. Moreover, if we are not able to verify the coding of the modules before we start to assemble them into larger building blocks the complexity of the integration of systems would be much too high and the integration process would get out of control.

A valuable technique and concept in the specification, documentation, and verification of programs are assertions. An assertion is a logical formula that uses programming constructs as logical concepts. So programming variables are used as logical variables. Functional procedures may be used as mathematical functions. Data types are used as logical types.

Assertions are logical statements about the states computer programs manipulate when executed. Assertions are helpful in many respects and are used explicitly or implicitly a lot in practice (see [14]).

2.7 System Integration and Architecture Verification

Given properly verified building blocks starting with modules we step by step compose and integrate them into larger subcomponents and in turn put together subcomponents into large components and finally assemble them into systems. To do that, of course, we have to be sure that the modules and components work together properly according to their decomposition into parts of an architecture. If the conceptual correctness of the architecture is verified with respect to the interface specifications of the components, the correctness of each component

implementation with respect to its interface specification is enough to guarantee the correctness of the overall system.

It is essential that the correctness of modules with respect to the architecture of a system can already be justified at the level of their interfaces. Therefore what we need is a theory that allows us to justify and verify the software architecture with the help of the component interface specifications independently of the verification of the correctness of the modules and the components. This is what modularity is about.

Again here the verification can be done by logical means including techniques that are applied automatically such as model checking but also by more pragmatic ones in the development process such as reviews and inspections or testing – all of them based on rigorous interface specifications.

2.8 Usability Engineering – User Centric Engineering

The most difficult and most important issue in software engineering is not only to achieve the correctness of systems according to their specifications but also the appropriateness of the specifications according to the users' needs. The problem is that this is sometimes very difficult and time consuming. A newly introduced system changes the users' view of the world. It is difficult to find out and to understand what the users really need in advance before a system is build.

However, it is also very difficult for a user to understand what the system does as it is specified by a software engineer before a prototype is available. Logical specification does not help here a lot. The only solution to this difficult problem of communication between humans today is prototyping. Prototyping means putting systems together and getting first implementations of systems by which it is demonstrated to the user what a system does and whether the system functionality is actually useful and needed. This is, of course, a very costly and very difficult issue.

The more systematic and precise the descriptions of systems are at the beginning and the better the techniques are to generate from those descriptions first demonstrators and prototypes, the easier it is to find out early about the actual needs and interests of the user. Also here logic and algebra play a major and decisive role for documentation, foundation and tool building.

Finally, logic may even be used directly for prototyping. This is the idea of logic programming and approaches that generate programs from logical specifications.

3 Discovery of a New World: The Theory of Software Engineering

As we have outlined in the last section software engineering has to look at many mutually dependable aspects of a software system but nevertheless in nearly all the steps logic and algebra can be helpful. Most importantly logic and algebra

provide a proper basis for software engineering activities, modelling and description techniques leading to a theoretical foundation.

Actually for software engineers it is always the discovery of a new world to look at the logical and the algebraic foundations of their discipline. Doing so they discover that what they do is not just putting together difficult statements and cryptic formalism and notation which may work or not according to the technical machinery but it provides them with a proper theory in which we can make sure that a particular statement is correct or not.

3.1 Mathematical, Logical, Algebraic Islands in Software Engineering

Mathematic logic and algebra provide islands of theory and foundations in the field of software engineering (see [5]). Finally they form proper grounds for building bridges. These bridges can help engineers in understanding their basic structures when describing their models, in specifying their artefacts, in finding models for what they are doing and finally in getting tool support with a justified methodology. The islands without the bridges are not very helpful, however. Only, if the theory is adjusted to the engineers' needs and integrated into their processes, it gets useful.

3.2 Denotational Semantics

Perhaps the most remarkable achievement in the foundations of programming languages and their semantics is denotational semantics (see [12, 16, 17]). According to its principles of mathematical denotations and compositionality it provides exactly what we need in software development namely a modular description tool kit of the essentials of programming languages and their abstract models such that we can put together programming languages and their behaviour in a way that we can abstract away details as much as possible. This supports abstraction and reduces complexity.

We can apply just the same principles for nowadays system description and modelling languages. Also there the concepts, notations, and treatment of denotational semantics are most appropriate because it introduces a kind of modularity that is badly needed in software engineering.

3.3 Assertions

Another very powerful technique in software specification and analysis are logical assertions. Assertions are logical formulas that are comments in the context of statements that talk about the artefacts of software in a logical style. Assertions

show a close relationship to denotational semantics. More precisely, an assertion is a logical formula that uses programming variables as logical variables and this way allows formulating logical statements about the state of a program.

Many computer scientists quite independently starting even with Alan Turing but later in particular by Floyd, Hoare, Dijkstra, Dahl and many others introduced assertions. Assertions are a powerful and elegant technique to provide precise statements about programs. For the usefulness of assertion recent papers by Tony Hoare [14] give many examples.

3.4 Abstract Data Types

Abstract data types were a major step in the foundations of data structures. Data structures are one of the most important issues of software because software needs data models. Data models are built today onto data types. Often more involved data models are provided by entity/relationship techniques and object oriented data models. Also these models can be translated into axiomatic specifications of abstract data types. All kinds of data models can be seen as descriptions of algebraic structures.

It has taken years for the computer scientists to understand that in computer science data structures are not just to be seen and used as sets but they are always to be studied in the context of their characteristic operations and relations. These operations form, together with the data structure sets, algebras in the sense of mathematics, in particular, universal heterogeneous algebras.

The properties of these algebras can again be specified nicely by algebraic axioms. This brings in a very solid foundation of data structures that can be also used and implemented by support tools. How important such concepts are can also be seen in the recent developments of UML with its Object Constraint Language OCL.

Abstract data types are in its purest form nothing but heterogeneous typed higher order equational predicate logic.

3.5 Process Algebras

Algebraic techniques apply, however, not only for axiomatization of data structures but also for control structures and for entire programs. This was the major insight derived from process algebras as they have been suggested by Robin Milner, Tony Hoare and later by many others.

The theory of process algebras shows that algebraic techniques cannot only be used to describe simple data sets but they can also be used to describe the control structure of programming languages in terms of their algebraic properties. In fact, by this, at a very high-level, data and programming languages are and can be described by algebraic laws. This leads into an algebraic theory of data, programs

and processes that fits very well together with the denotational semantics of Scott and Strachey (see [18]) because it makes explicit the algebraic theory of the notation and its model if done in a proper and appropriate way.

3.6 Temporal Logics

Another fascinating generalisation of logic in software engineering, in particular towards the needs of reactive and concurrent software systems, is temporal logics. Temporal logic has been suggested to address the issues of systems where not only a transformation of an initial state into a final state is of interest but of systems which are reactive and interactive such that many places of interaction between the system and the environment during the lifetime of a system are of major importance. Therefore it is more appropriate to describe systems that way by traces of states or actions. Temporal logic allows us to specify properties of such tracks modelling systems and to reason about them.

Temporal logic is the logic of sequences of states or actions and provides particular logical foundations for that. Pioneering work by Fred Kröger, Armir Pnueli and Zohar Manna has paved the way for an extended theory of temporal logics, which nowadays becomes more and more practical, in particular, for the specification and verification of reactive or communicating systems in connection with the model checking of temporal properties. Lamport has turned this theory much more into an engineering approach capable with systems of impressive complexity. Model checking brings together algorithmics and issues of computation with logics.

3.7 Interface Abstraction

The interface of a software system or a component is given by the patterns of interaction between the system and its environment. Interface abstraction is perhaps one of the most useful concepts of software engineering.

Without interface ideas large software systems are incomprehensible and much too complex to manage. Interface abstraction is based on core techniques from algebra and from logics. Algebra tells us how to weaken given constructions in terms of homomorphisms and quotient structures, logic tells us how to describe the interfaces in a precise way.

3.8 Model Checking

Model checking is a consequent application of techniques from automata theory and temporal logic combined with a lot of ideas to gain efficiency. In model checking the set of reachable states of a system is enumerated to check whether all of them fulfil a given property.

In principle, people have thought for long while that it would be practically impossible to check large finite state systems by considering and visiting all their states. However, according to the growing computation power of our machines and sophisticated algorithms that make it efficient to check millions and schematically even billions of states model checking today is a serious and practical technique to prove properties about programs and software systems.

4 The Role of Digital Modelling

As already pointed out one of the remarkable achievements of informatics is digital modelling technology. Digital modelling technology is different to the analog models of classical mathematics formed out of continuous functions and the mathematics of integration and differentiation.

Analog models: The modelling techniques of the nineteenth and twentieth century were continuous functions, differential and integral mathematics; we speak of quantitative models.

Digital models: The modelling techniques of digital technology the twenty-first century are discrete models of logic and algebra; we speak of qualitative models.

The foundation of digital models is logic and algebra. However, in contrast to the classical way in which logic was treated in the field of mathematical logic as a theoretical field of study, computer sciences needs an engineering usage of logic and deduction. Still, there is a long way to go to make the methods of mathematical logic practically usable in a broad sense in the field of software engineering and to exploit all the potentials of logics there.

5 On the Nature of Software Development

Many people think that the following extreme views of logic and algebra on one side and the software development on the other side cannot be integrated:

- Software development means the construction of formal/mathematical/logical models – therefore it is a formal activity – software is a mathematical object that can be analyzed by mathematical techniques, specifiable, and verifiable.

- Software development is an art and a craft; it proceeds by esoteric lore, by stepwise improvement, by trial and error – software is an artifact, immeasurable, unreliable, and unpredictable.

We, however, believe that software engineering will only make substantial progress if we manage to integrate both extreme views.

We have to understand all the logical and algebraic properties and foundations that are related with software systems and at the same time we need to have all the pragmatic understanding of the economics and the organization and the management aspects of software. This applies, in particular, also to all the technical and system aspects of software.

6 Software Development as Modeling Tasks

From conceptual point of view software development has to be understood as a modelling task. A software system is just a digital executable model. Everything in the application domain that is relevant for a software process has to be finally captured explicitly or implicitly in terms of the digital models of informatics. The better and more appropriate and more integrated these models are the easier it will be to make use of them in the software development process.

6.1 Software Development: Modeling and Description of Various Aspects of Systems

Software development can be seen as a sequence of system modeling steps. There are many aspects to model when carrying out software development:

- Application domains, their data structures, functions, laws, and processes,
- Software requirements, based on domain model comprising data models, functions, and processes,
- Software architectures (see [9]), their structures, and principles,
- Software components, their roles, interfaces, states, and behaviors,
- Programs (see [3]), their internal structure, their runs (see [1, 2, 7, 10, 11, 13]), and their implementation.

For modelling we need a family of modelling concepts and a modelling theory. For many of these aspects we have proper and helpful models at hand by now. However, it will take a while to understand the role and the depths of these models in all their details and to find a good way to relate and to integrate them and their aspects.

6.2 Modeling in Software Engineering

Systematic development of distributed interactive software systems needs basic system models. These models have to reflect the structures of distributed software systems.

Description techniques are to provide specific views and abstractions of software systems such as:

- The data view,
- The interface view,
- The distribution view,
- The process view,
- The interaction view,
- The state transition view.

The technical development of systems concentrates on working out these views that lead step by step to an implementation. Each view corresponds to a logical model and to a logical formula. The integration of these views leads to a comprehensive logical model with intricate questions of consistency.

6.3 What a Model Is in Software Engineering

The term model is used in many meanings in software engineering. People talk about data models, process models, system models, domain models and finally about meta-models. What is a model in software engineering? An annotated graph or a diagram? A collection of class names, attributes, and method names?

In our view, a model is an abstraction. In engineering, a model is always a collection of formulas, diagrams, tables expressed in some notation with a well-worked out and well-understood mathematical theory!

Hence software engineering needs mathematical modeling theories of digital systems – algebra, logic, and model theory! Logic provides a unifying frame!

We end up with a very classical view of the role and nature of models in software engineering. A model in software engineering is basically the same as a model in logic. And what is a model in logic? Basically it is an algebra, which consist of a family of sets, relations, and functions. Therefore from a very abstract formal point of view a software system is a description of a model very much as logic uses a model for explaining the semantics of logical formulas.

7 "Formal Methods" and "Software Engineering"

The term "formal method" generally refers to the application of techniques from logic and algebra in software development. Formal methods are often considered by practioners as being inadequate, too expensive, too difficult, and "not practical".

In contrast, practical software development is often considered by formalists being ad hoc, “immature”, uncontrollable, and “not an engineering discipline”.

The following observation is remarkable: A high percentage (> 70 %) of large software projects fails or falls short. This reflects the state of the art today of software engineering. There are many reasons for that. Only one reason is the lack of theoretical foundations.

The practice in modeling in software engineering today is diagrams. In practice, today we find many diagrammatic methods for modeling and specification (SA, SADT, SSADM, SDL, OMT, UML, ROOM, ...). Diagrams are helpful as long as they provide easy to understand intuitive descriptions of systems and their aspects. However, often diagrams stimulate quite different associations by their various viewers, depending on their background (see [4]). Then the quite intuitive understanding is counterproductive and a dangerous source of misunderstandings.

Here a universal modeling language could help. A universal language with a well-defined meaning, well structured to provide a basis of understanding. The concept of universal modeling languages is obviously a great idea – but a closer look at languages such as UML shows, how ad hoc most of its concepts and “methods” are for the practical modeling tasks of today. At best, they reflect interesting ad hoc engineering insights and practices in engineering particular applications.

The consequence of these weaknesses and ad hoc approaches is remarkable: never have practical diagrammatic modeling been justified on the basis of a comprehensive mathematical foundation.

We only mention three disappointing but representative examples:

- UML and its statecharts dialect with its endless and fruitless discussions about its semantics.
- Behavior specification of interfaces of classes and components in object oriented modeling techniques in the presence of callbacks.
- Concurrency and cooperation: Most of the practical methods especially in object orientation seem to live in the good old days of sequential computation and do not properly address the needs of highly distributed, mobile, asynchronously cooperating software systems. The introduction of threads as it is done for instance in Java to deal with concurrency is ad hoc and error prone.

The examples show the state of the art. A much deeper understanding is required to overcome the weakness and come up with appropriate models with deep and comprehensive tool support.

8 From Logics to UML and Back to Logics

As said above practical engineering tools of today such as UML fall short in many respects. They do not provide the strong and precise modelling framework that we need so badly. They do not offer a basis of common understanding.

8.1 The Vision – An Academic View!

Today advanced practical software developers have understood that modelling is at the core of software engineering. This has led to developments like UML, the Unified Modelling Language, which claims to provide all the basics for modelling software systems. Unfortunately UML does not make much use of all the deep understanding that has been gained in models in informatics over the last 30 years in fields like denotational semantics, assertions logics, temporal logic, algebraic data types and many other research contributions.

As a result the UML approach, as technique to describe a formal model of distributed software systems the approach remains shallow, imprecise, full of contradictions, and fails to provide an integrated system model.

What we need is rather obvious:

- Foundations of modeling: A tractable scientific basis, understanding, and theory for modeling, specifying, and refining of programs, software and systems.
- Powerful models supporting refinement, levels of abstractions, multi-view modeling, domain modeling.
- Comprehensive description techniques based on these foundations.
- A family of justified engineering methods based on these foundations.
- A flexible development process model combining these methods.
- Comprehensive tool support including validation, consistency checks, verification and code generation by algorithms and methods justified by the underlying theories.

Finally we aim at modeling and its theory as an integral part of software construction as an engineering discipline.

9 A Logical and Algebraic Manifest of Software Engineering

Looking at what we have outlined so far we see that there is still a long way to go. A lot of the foundations that have been worked out over the last 30 years are understood quite well by now. Unfortunately still there is a surprising discrepancy between people that have all the theoretical understanding based on logic and algebra but do not understand their practical impacts and the people in practice which do not have the theoretical understanding and are used to their ad hoc methods.

What we need to improve the situation is a close cooperation between theoretical experts and practical experts. Only this way we can be sure that the essential practical issues are addressed with the right foundations.

A consequent foundation for the engineering of software intensive systems is obtained by logics. Each description in software engineering – be it text, diagrams,

graphics or tables is a logical statement and can be translated into a logical formula: this way logic and algebra provide unifying frames, which are not available for UML, so far.

Consequences and advantages of the semantic insights for methods and tools are obvious:

- Precise semantics (see [8, 9])
- View modelling and integration
- Notion of view consistency
- Basis for transformation, manipulation, and reasoning about systems

Logic and algebra engineering will form the basis of digital systems modelling. This will make logic practical and practice logical.

10 Outlook

We have gained a lot in the theory and foundations of software engineering over recent years. The transfer to engineering is on its way. But more theoretical work is needed. We need deeper insights, better methods and tools that can only be provided by a team of experts:

- Researchers working on the foundations in logics, algebra and mathematics,
- The designers of practical engineering methods and tools,
- The programmers and engineers in charge of practical solutions,
- Application experts modeling application domains.

Successful work, however, does not only require the interaction between these types of people – we also need *hybrid* people that have both a deep theoretical and practical understanding and therefore can build the bridges between theory and practice.

References

1. Alur, R., Holzmann, G.J., Peled, D.: An analyzer for message sequence charts. Software – Conc Tool **17**, 70–77 (1996)
2. Ben-Abdallah, H., Leue, S.: Syntactic analysis of message sequence chart specifications. Technical Report 96–12, Department of Electrical and Computer Engineering, University of Waterloo (1996)
3. Booch, G.: Object Oriented Design with Applications. Benjamin Cummings, Redwood (1991)
4. Broy, M.: Towards a formal foundation of the specification and description language SDL. Form Asp Comput **3**, 21–57 (1991)
5. Broy, M.: Mathematics of software engineering. Invited talk at MPC 95. In: Möller, B. (ed.) Mathematics of Program Construction, July 1995. Kloster Irsee, Lecture Notes of Computer Science, vol. 947, pp. 18–47. Springer, Berlin (1995)

6. Broy, M.: A logical basis for modular software and systems engineering. In: Rovan, B. (ed.) SOFSEM'98: Theory and Practice in Informatics. Lecture Notes in Computer Science, vol. 1521, pp. 19–35. Springer, Berlin/New York (1998)
7. Broy, M.: The essence of message sequence charts. Keynote speech. In: Proceedings of the International Symposium on Multimedia Software Engineering, IEEE Computer Society, pp. 42–47, 11–13 Dec 2000
8. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, New York (2001)
9. Broy, M., Hofmann, C., Krüger, I., Schmidt, M.: A graphical description technique for communication in software architectures. Technische Universität München, Institut fürInformatik, TUM-I9705, February 1997 URL: http://www4.informatik.tu-muenchen.de/reports/TUM-I9705, 1997. Also in: Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC′97/ICSC′97)
10. Cobben, J.M.H., Engels, A., Mauw, S., Reniers, M.A.: Formal semantics of message sequence charts. Technical Report CSR 97/19, Departement of Computing Science, Eindhoven University of Technology (1997)
11. Damm, W., Harel, D.: Breathing life into message sequence charts. Weismann Insitute Technical Report CS98-09, April 1998, revised July 1998, to appear. In: FMOODS′99, IFIP TC6/WG6.1 Third International Conference on, Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, 15–18 Feb 1999
12. Dana, S.: Scott: Logic and programming languages. CACM **20**(9), 634–641 (1977)
13. Graubmann, P., Rudolph, E., Grabowski, J.: Towards a Petri Net based semantics definition for message sequence charts. In: Faergemand, O., Sarma, A. (eds.) Proceedings of the 6th SDL Forum, SDL′93: Using Objects, (1993)
14. Hoare, C.A.R.: Assertions in programming: From scientific theory to engineering practice. In: Soft-Ware 2002: Computing in an Imperfect World, First Conference, Soft-Ware 2002, Belfast. Lecture Notes in Computer Science, vol. 2311, pp. 350–351. Springer, Berlin/Heidelberg (2002)
15. Holzmann, G. J., Peled, D. A., Redberg, M. H.: Design tools for requirements engineering. Bell Labs Tech J Spec Issue Softw **2**(1), 86–95 (Spring 1997)
16. Scott, D.S.: Outline of a mathematical theory of computation. In: Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems, pp. 169–176. Princeton University, Princeton (1970)
17. Scott, D.: Data types as lattices. SIAM J **5**(3), 522–586 (1976)
18. Scott, D., Strachey, Ch.: Towards a mathematical semantics for computer languages. In: Proceedings, 21st Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, pp. 19–46. Also Programming Research Group Technical Monograph PRG–6, Oxford (1971)

http://www.springer.com/978-3-319-00966-7

Mathematics, Computer Science and Logic - A Never Ending Story
The Bruno Buchberger Festschrift
Paule, P. (Ed.)
2013, VII, 113 p. 35 illus.,
ISBN: 978-3-319-00966-7