Embedding Mathematical Techniques into System Engineering

Maritta Heisel, Stefan Jähnichen, Martin Simons, Matthias Weber

Technische Universität Berlin* and GMD Forschungsstelle FIRST†

Abstract

Some of the reasons why formal methods have not been widely accepted in practice are analyzed. This analysis provides the basis for a more modest approach to embedding mathematical and formal techniques into the system design process: identifying places in traditional design methodologies where formal reasoning can be convincingly introduced. The approach is outlined in general and illustrated by giving overviews of three different research activities.

Keywords: Formal methods; dynamic simulation; traditional design methods; object-oriented design.

1 Formal methods' chronic crisis

The quest for abstraction and structuring concepts has been one driving force in the design of programming languages and design methods. It is generally accepted, that they increase productivity, reduce the amount of errors, and support maintenance and reuse. The list of advantages could be continued. The recent success of object-oriented concepts is merely one proof for this. However, even though conceptual abstraction and structuring has also been a major driving force of formal methods, they have not been embraced by industry to any comparable extent; rather, the state of formal methods could be characterized (or dramatized if you prefer) as that of chronic crisis, at least with respect to industrial relevance.

Industry today produces complex software systems that do work properly. Safety critical systems are installed. However, already now, disastrous failures occur (such as the Therac-25 accidents [LT93]), and yet the complexity of systems that will be constructed in the future is destined to grow, and the areas where safety-critical embedded systems are installed will undoubtedly grow too. Thus, the problems to control complexity that are involved in the construction of such systems will increase too. That the construction of reliable systems is still far from being a mature engineering discipline is being aptly documented in a recent Scientific American article on "Software's chronic crisis" [Gib94]. Despite the lack of widespread application there is continuing

*Forschungsgruppe Softwaretechnik (FR5-6), Franklinstr. 28/29, D-10587 Berlin, Germany. e-mail: {heisel,jaehn,simons,we}@cs.tu-berlin.de

†Rudower Chaussee 5, D-12489 Berlin, Germany.

and even increasing interest in formal methods within the scientific and the industrial research community. The reason seems to be that, in principle, formal methods promise to be a cornerstone for software engineering to become a mature engineering discipline.

Formal methods have been seriously applied during the past years in various industrial and academic pilot projects as, for instance, reported in [CGR93]. However, the breakthrough has not been achieved. Many companies involved in such projects are scaling down their use of formal methods to a level that is in accordance with their current industrial relevance. For instance, they only have small teams of highly trained research staff working on selected critical aspects of systems.

What are the reasons for the failure of formal methods to achieve mainstream acceptance? From our own experience [HWW94] and from our analysis of experience reports [CGR93, HK91, SOF94, for instance] we believe that one major reason is that presently formal methods come with too broad a goal. Very often, they aim at a complete and superior methodological framework for the development of correct systems without compromises. They often presuppose idealized circumstances, and they usually have been developed in academic environments where such circumstances can be guaranteed. Also, such a monolithic approach does not leave much room for it to coexist and to interact with other methodologies that are in standard use within an industrial development context. Still, research on such methods is necessary and has provided us with many useful techniques and results, but it will most probably not lead to methods that will be quickly accepted in practice.

We believe that a more modest approach to the integration of formal techniques into the system design process will lead to a more immediate application of such techniques in the system design process. Starting out from existing and accepted conventional design methods which are amenable to the integration of mathematical techniques, one should investigate at which points and places during the design process mathematical techniques can be smoothly and reasonably integrated. Resorting to formal techniques at these points and places should be convincing to an experienced engineer. Once experiments and case-studies have provided evidence that the formal elements introduced are accepted, one can start to investigate further possible anchor points for mathematical techniques. Then, one can base this investigation on the experience gained during the first phase and on the grown formal literacy of the design team. Hence, in principle, by iterating this process, one obtains a method that has more and more formal elements. Note carefully, that we do not attempt to introduce conventional techniques to a formal method but rather the other way around.

Relating back to our opening remarks, we can now present and discuss some implications of our position towards formal methods:

Formal methods' chronic crisis will continue as long as. . .

. . . **we are unable to embed mathematical techniques into conventional system design.** As for abstraction concepts that found their way into programming languages and design methods, we believe that a pragmatic and liberal embedding of formal techniques into conventional methods bears great potential. In those areas where the use of mathematics helps to make underlying abstractions tangible, precise, and subject to mathematical reasoning, their use should be advocated. Convincing evidence in the form of examples and case-studies should be provided. In other areas, where insisting on pure formality makes the process cumbersome, it should not be enforced. As an ex-

ample, take the use of the mathematical notation Z to specify systems with complex data relationships: it is our experience that the result of many discussions about some aspect of the functionality of such systems could be neatly captured on a blackboard filled with Z schemas. We think, that it is precisely this kind of experience, which has led to the relatively large popularity of the Z notation. On the other hand, Z does not seem to be quite as appropriate to specify complex control relationships, and one would rather prefer a notation such a Statecharts for this activity.

. . . **we understand them as a complete technology.** Formal methods are certainly not the exclusive and superior method for the development of safety-critical systems. It is highly questionable, judging on the basis of our own and on reported experience, whether the current monolithical viewpoint of formal methods as a complete technology will lead to reasonable and accepted practical methods. We think that the complexity of future embedded systems will require continuing experimentation with a variety of new techniques to tackle complexity and enhance safety, including the use of formal methods. It is now generally accepted, that the use of formal methods does not *guarantee* that the designed system performs as expected, even though this myth keeps popping up every once in a while. Furthermore, formal methods as such do not offer much help in many other important aspects of engineering safety-critical systems. As an example, take the problem to design a suitable overall architecture of a safety-critical system, which as much as possible localizes and simplifies the truly safety-critical functionality, following the guideline "keep it simple". A simple and transparent architecture can as much contribute to the safety of a system as can the use of a verification tool for selected critical pieces. As another example, take the issue of designing an adequate amount of fault tolerance into a system so as to make it behave predictably under various catastrophic circumstances.

. . . **we understand them as a mere safety technology.** Formal methods are industrially relevant for the engineering of any kind of complex system, not merely safety-critical ones. With a more liberal approach to what one demands of "formal" methods, they have the potential to also improve software engineering in general. As an example take software and system testing which is certainly relevant in any complex system. Usually, testing involves the activities of test-case definition, test data generation, and test evaluation. However, if some abstract test cases can be specified in a suitable formal notation, then test-case evaluation, which in practice often is a labour-intensive activity, especially in case of regression testing, could be partially automated.

. . . **we strive to build monolithical support environments for them.** Tool support is mandatory for formal methods to become accepted in practice. From what we have said so far, it does not make much sense to attempt to build a monolithic support environment for a single method. Instead, loosely coupled tools should be constructed. For them to cooperate smoothly, a common software architecture should be designed. Tools that are needed are primarily mathematical assistants that capture knowledge of notations, formalisms, and mathematical structures that are relevant for system design much in the tradition of, e.g. Mathematica.

In the remainder of this paper we will try to further explain and illustrate our position by describing the way we are investigating formal methods in the context of three ongoing

projects at the Technical University of Berlin: the development of a simulation environment for energy-transducing systems, the formalization of traditional design processes, and the embedding of mathematical techniques into object-oriented system engineering.

2 Elements of formal methods for dynamic simulation

The aim of an ongoing interdisciplinary research project is to design and to implement an object-oriented simulation environment for dynamical systems. It is used to simulate complex energy-transducing systems (e.g. a city with energy producers and consumers) and to answer questions about performance related, economical, or ecological aspects of such systems. The environment comprises an object-oriented simulation language, together with its compiler, run-time system and a family of mathematical solvers, a graphical user-interface, and a library of model components from which more complex systems can be constructed. This environment is being developed in close cooperation with various engineering departments using techniques of modern software engineering, like compiler construction tools, interface builders, and configuration management tools.

It has not been intended to use formal methods for the construction of this environment. We rather propose to introduce elements of formal methods to the development of dynamic simulations. This process is a special case of program development and in practice it shows many of the same deficiencies. On the other hand, the basis from which one starts to develop a simulation is, at least mathematically, well defined.

A dynamical system is modeled in principle by a set of differential and algebraic equations that express the relationships between the observable quantities of the system. In order to simulate the system, its mathematical model is usually directly translated into code expressed in a simulation language. The numerical computations are either directly implemented with the numerical operations of the language or indirectly by making use of mathematical solvers that come with the runtime system of the language. Various simulations are then carried out. By comparing the results with expected, estimated or measured values, the model that has been the basis of the simulation is validated.

The mathematical model thus serves as a specification for the simulation code. This is fine and sufficient, as long as the system is not very complex. However, in the context of large systems — like the above mentioned energy-transducing systems — that are structured into subsystems and modeled by thousands of equations, the step from the mathematical model directly to the simulation code is too large and error-prone. For instance, many local computations or local optimizations are needed to get around certain numerical constraints. What is needed is a *computational model* of the system that takes into account computational aspects of the system, which are ignored by the mathematics but which are important once the system is to be simulated by a machine.

Our approach to obtain such a computational model is to design an object-oriented specification language for hybrid systems. A hybrid system contains discrete and continuous components (such as a switch and a heater). In this language the hierarchical structure of a system can be expressed, state variables can be named and typed, while the relationships between these quantities are still expressed in the language of math-

ematics. To be more precise, the specification language we are designing can be seen as an extension of Object-Z [CDD+90]. A specification consists of a set of classes of three different types: discrete, continuous and hybrid classes. The discrete classes correspond to the classes of Object-Z. The state of a continuous class is described by continuous variables which are functions of time. The state invariant is expressed using Z-predicates and differential and algebraic equations.

A computational model expressed in this specification language is much closer to the mathematical model than to the simulation program expressed in a simulation language. On the other hand, the structure of the system and the interdependence of the subsystems is made explicit. Side conditions, that have to hold for a subcomponent to work properly can be formally expressed. Hence, one of the immediate uses of such a specification is that it can serve as a documentation in a library of reusable components.

Once a precise semantics for the specification language has been worked out, it will also serve as the basis for further investigations. First of all, to analyze what is necessary to automatically generate a simulation program out of such a specification will involve locating where numerical constraints have to become explicit. Next, it is worthwhile to study how the notion of refinement in modeling theory is reflected in the computational model and how it relates to the notion of refinement from formal software development. This is important in the context of adaptive simulation. Take for instance a building as an energy consumer. It can be modeled either as a single unit by specifying how much energy it consumes as a function of time. In order to have more precise results, the model can be refined to take into account that the building is made up of rooms with walls and windows, which are modeled in turn and which are composed to form the building. Finally, the parallelization of simulations on the level of models is of great interest if one wants to simulate large systems. Parallelization of simulation in general is an important research area because simulation in general and the simulation of large systems is very computation intensive. We want to investigate ways to make the independence of subsystems of a large system explicit, allowing the subsystems to be simulated in parallel. We believe that this can be much better expressed at the level of the computational model than at the level of the simulation program. Furthermore, at this level, the tradeoff in precision by simulating in parallel should be easier to analyze.

What we have described in the previous paragraph are very tentative research goals. But the motivation behind these goals should have become clear: We tried to isolate areas where formal techniques can be profitably applied. By doing this, we hope on the one hand to make as much use of the results of formal methods and on the other hand to experiment with, to adapt, and to extend on these results in a restricted field of system design.

3 Formalizing elements of traditional methods

The project we describe now has the aim to make the formalization of elements of traditional software development methods possible. The motivation for such a formalization is the aim to positively guarantee certain semantic properties of the product. Examples of such properties for programs are correctness or complexity. For specifications, one might wish to show that one function is the inverse of another, or that the system – if implemented correctly – cannot enter certain undesirable states.

This approach avoids to impose a prescribed and unflexible style of working on the users of formal methods. Formal methods are adapted to traditional software development processes, instead of vice versa.

Our contribution mainly consists of two parts: its basis is the notion of *strategy* as a knowledge representation mechanism [Hei94] which makes it possible to formally describe software development activities. To make strategies practically applicable, a generic system architecture has been designed [HSZ94] that provides a straightforward implementation technique for systems supporting strategy-based software development.

Formalizing processes as strategies. Strategies describe possible steps during a development. Examples are how to decompose a system design to guarantee a particular property, how to conduct a data refinement, or how to implement a particular class of algorithms. This kind of knowledge can be found in text books on software engineering.

Technically, the purpose of a strategy is to find a suitable solution to some software development problem, e.g. to set up a formal specification faithfully reflecting some given requirements, or to develop a program that meets a given specification.

A strategy works by problem reduction. For a given problem, it determines a number of subproblems. From their solutions the strategy produces a solution to the initial problem. Finally, it tests if that solution is acceptable according to some notion of acceptability. The solutions to subproblems are naturally obtained by strategy applications as well. In general, the subproblems of a strategy are not independent of each other and of the solutions to other subproblems. This restricts the order in which the various subproblems can be set up and solved.

A strategy describes how exactly the subproblems are constructed, how the final solution is assembled, and how to check whether this solution is acceptable or not.

The generic system architecture. The definition of strategies is parameterized by the notions of problem, solution, and acceptability. Therefore, it is possible to design a generic system architecture to support strategy-based development processes. The two most important components of the architecture are the *strategy base* and the *development tree*.

A development consists of a loop of strategy applications. The intermediate states of the development are represented by the development tree. Its nodes contain a problem and its solution (once it has been found), and references to its children and to siblings it depends on. Each new strategy application causes the development tree to be extended, if the strategy reduces the problem to some subproblems. Otherwise, the problem is solved immediately, and the solution is recorded in the respective node. When all subproblems of a problem have been solved, its solution can be assembled from the solutions to the subproblems. The development is finished when all problems have been solved. Representing the state of development as a data structure makes it possible to obtain an overview of the development at any time.

The available strategies are stored in the strategy base. It consists of modules, each of which implements a single strategy. Strategy modules are defined in a uniform way. Due to this fact, new strategies can be incorporated in a routine way. As a result, the system becomes what one might call *locally customizable*. This means that an instance of the architecture can easily be enhanced to serve its purpose better than before.

Moreover, the genericity of the system architecture guarantees for its *global customizability*. Changing the definition of problems, solutions, and acceptability means to obtain a support system for a new activity in software development. This makes the architecture a good instrument to support different software engineering activities with different degrees of formality or rigor.

This system architecture has been implemented in a prototype program synthesis system, called IOSS (Integrated Open Synthesis System) [HSZ94].

We have stressed that a non-monolithic application of formal methods is essential for their acceptance in practice. The approach described above follows this philosophy. Strategies and their application can be embedded into traditional software processes. These are performed nearly as before, except that some parts are now formalized and machine-supported. The formalized parts will require a certain degree of expertise, but this is the inevitable price of being able to guarantee semantic properties of the software product.

Without machine support, strategy-based software development would be a hopeless enterprise. But not any system will do: the *kind* of tool support is crucial. In a first (and now discarded) version of IOSS, strategies were monolithic functions. As a result, incomplete parts of the development were only contained in the run-time stack of the system. This made it impossible to get an overview of the development; incomplete developments could not be stored. These and other restrictions made it very hard to develop nontrivial programs.

With the new architecture, not only do different methods within one phase of the software development process become compatible. It is also conceivable to support different phases with different instantiations of the architecture. Since these behave similarly, it is easier for developers to work with different instantiations of the architecture than with entirely different tools.

4 Mathematical elements in object-oriented design

In this project we try to identify areas where mathematical techniques could be embedded into object-oriented design. There are essentially three reasons for concentrating on object-oriented design: it incorporates the current state of the art with respect to many well-known problems of complex system design, it is of growing practical relevance, and we think that mathematical techniques can be embedded particularly well into certain parts of the object-oriented paradigm. We will try to sketch some of these areas in the following discussions.

The main issue of object-oriented analysis and design is to identify adequate conceptual abstractions for a particular application and then to design a class architecture around them. From a mathematically-abstract viewpoint, each class interface induces a theory in which the abstract behavior of the class, i.e., state space, state invariant, and the behavior of its objects, is specified and properties about this behavior are derived. Another important part of this theory are mathematical descriptions of the behavioral contracts of the class operations, i.e., the precondition that clients of this operation have to ensure when expecting the operation to work properly, and the postcondition that the operation guarantees to be true after its execution. We think that the formal specification of key parts of this theory during design can be of significant value for the

clarification of complex behavior. However, we only advocate for a partial and liberal embedding of such specifications into class interfaces, in particular, we do not want to enforce the mathematical modeling of low-level aspects such as storage allocation, exception handling, or process synchronization.

In practice, object-oriented design is often described using various diagrammatic notations or textual fragments of some object-oriented programming language. The class architecture is usually represented in class diagrams [Boo94]. These diagrams describe various kinds of information, e.g. inheritance relationships, constraints on the state space, naming of classes and operations, visibility relationships, parameters of a class, cardinality constraints, and other more general associations between classes. Roughly speaking, the interface specification of a class induces this information and can thus be seen as a model of the class diagram. An interesting problem, however, is how to deal with relationships between classes, such as cardinality constraints between attributes of classes. Specifications of class interfaces localize the modeling of this information in special attributes, or in states and invariants of particular classes. Ideally however, information of this kind should be expressed with mathematical elements embedded into class diagrams.

The reactive behavior of classes is often specified in state transition diagrams, such as Statecharts. It remains an interesting topic to study the relationship between Statecharts and a mathematical notation such as Z, especially with respect to stepwise decomposition and the proof of safety properties. Again, the approach would be to embed elements of mathematical specifications into Statecharts in order to derive selected safety properties rather than the other way around.

During analysis and design, it is a common technique to identify and describe a number of expected working scenarios of the system. Such scenarios are often described in so-called *object diagram* (or the closely-related *configuration diagrams* or *interaction diagrams*). Similar to class diagrams, object diagrams illustrate the collective behavior of several classes rather than a particular individual class. Especially during analysis, they are meant to give rigorous but not fully detailed behavioral specification, so as to indicate a certain strategy or pattern in which various objects interact to cooperatively achieve some goal. In some cases, object diagrams overlap with the mathematical specification of postconditions of operations, i.e., sometimes the postcondition is a mathematical description of the collective effects of other operations. In other cases, however, object diagrams rather correspond to implementations of abstract postconditions. Thus, we think that object diagrams could benefit from the incorporation of mathematical elements.

Data refinement is the stepwise process of refining abstract descriptions of classes to more concrete ones, in the sense that the structures and operations in concrete classes are closer to an efficient implementation in an object-oriented programming language. Object-oriented development seems to deal with data refinement in two heterogeneous forms: either within inheritance trees, in case of true subtyping, or within architectural levels of abstract machines, each expressed by a cluster of classes. We think that in both of them the embedding of mathematical elements, such as the mathematical definition of a refinement relation, could very much clarify system design.

In summary, we have tried to point out how mathematical elements could be helpful when embedded into structures of object-oriented design, in particular class interfaces, class diagrams, Statecharts, object diagrams, and data refinement relationships. We are

aware that the above remarks are rather general, but still we think that they can form part of a basis for developing a practically interesting approach. We prefer to develop such an approach within a mixed academic/industrial setting, concentrating on practical case studies.

5 Conclusion

In the preceding sections, we have presented ongoing research projects of our group. Even though they are quite heterogenous in their topics and methods they share the common philosophy that was outlined in the first section. Interaction between the three projects is possible and is actively pursued. We are convinced that it is not conceivable to find a wide-spanning method designed to replace traditional processes as a whole. We instead try to identify different areas where formal methods can be helpful and develop approaches to support these with appropriate means. Finally, we would like to stress that research on formal methods has to keep up with the ongoing research in software engineering. We should not only try to support established techniques with formal methods. These could be out of date in a few years. Instead, research on formal methods should concentrate on novel and promising new techniques in software engineering.

References

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, second edition, 1994.

[CDD+90] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. Elsevier, 1990.

[CGR93] D. Craigan, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 1993.

[Gib94] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, pages 72–81, September 1994.

[Hei94] M. Heisel. A formal notion of strategy for software development. Technical Report 94–28, TU Berlin, 1994.

[HK91] I. Houston and S. King. CICS Project Report: Experiences and Results from the Use of Z in IBM. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods*, volume 551 of *LNCS*, pages 588–596. Springer-Verlag, 1991.

[HSZ94] M. Heisel, T. Santen, and D. Zimmermann. A system architecture for strategy-based software development. Submitted for publication, 1994.

[HWW94] M. Heisel and D. Weber-Wulff. Korrekte Software: Nur eine Illusion? *Informatik – Forschung und Entwicklung*, 9(4), October 1994.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 25(7):18–41, 1993.

[SOF94] Special issue on safety-critical systems. *IEEE Software*, January 1994.