Measuring the Effect Of Formalization

KETIL STØLEN, PETER MOHN*
OECD Halden Reactor Project, Institute for Energy Technology
Halden, Norway
*)the second author is on leave from the Swiss Federal Nuclear Safety Inspectorate, Villigen, Switzerland

Abstract

We present an ongoing research activity concerned with measuring the effect of an increased level of formalization in software development. We summarize the experiences from a first experimental development. Based on these experiences, we discuss a number of technical issues; in particular, problems connected to metrics based on fault reports. First of all, what is a fault? Secondly, how should the fault counting be integrated in the development process? Thirdly, any reasonable definition of fault depends on a notion of satisfaction. Hence, we must address the question: What does it mean for a specification or an implementation to satisfy a requirement imposed by a more high-level specification?

1. Introduction

The OECD Halden Reactor Project (HRP) is an international cooperative effort involving 20 countries and more than 100 nuclear organizations. The research of the HRP is specialized towards improved safety in the design and operation of nuclear power plants. The use of formal descriptions in software development, validation and verification is an active research direction at the HRP. Earlier this research was specialized towards small safety critical systems with very high reliability requirements. More recently, the scope has been extended to cover also other kinds of software; in particular, distributed systems for plant control and supervision. HRP member organizations have identified the need for this kind of research. They have also identified the need to know more about the practical consequences and effects of an increased level of formalization in software development. This paper describes our attempts to tackle problems connected to the latter issue.

Arguments in favour of an increased level of formalization in software development can easily be formulated. For example:

- Formalization of requirements raises questions whose answers may eliminate weaknesses and inconsistencies.
- Formal requirements are easier to analyse with respect to consistency, completeness and unintended effects than informal ones.
- Formal requirements reduce the number of design faults because the developers obtain a better understanding of what the requirements actually mean.
- Formalization allows faults to be identified early in the development process.
- Formalization is a prerequisite for mechanized validation in the form of animation, exhaustive exploration and mathematical reasoning.
- Formalization allows precise strategies for decomposition and design.

These arguments all sound reasonable, but are they valid in practise? And if so, are there other less desirable effects of formalization? For example, does a higher level of formalization increase software development and maintenance costs? These questions are not easily answered. We are not aware of convincing experimental evidence with respect to the effects of formalization. In the literature there is not much to find on this subject. Two notable exceptions are [1], [2]. They present results from small scale experiments where conventional developments are related to developments based on formal development methods like VDM [3] and B [4]. Both papers report on evidence in favour of increased formalization. This evidence is, however, weak. As pointed out by [1]: “Conclusions drawn from this experiment should be moderated by the small size of the development and the correspondingly small number of faults detected. The development team was also small and staffed by self-selected individuals who, being keen to make a success of the experiment, were perhaps better motivated than average. It would not be wise to extrapolate these results to larger projects.”

In 1997 the HRP initiated a research activity which, in addition to several other tasks, tries to address issues related to the effects of formalization. This paper sums-up some preliminary experiences from this research activity; in particular, based on what we learnt from an experimental development, it identifies and discusses problems connected to metrics based on fault reports:

- In order to report on faults we need a clear definition of what a fault is. This definition must be such that it does not force the alterations resulting from the kind of trial/failure experiments that is a desirable part of any software development to be counted as faults. Moreover, this definition should not depend on a software process that does not mirror how software is developed in practise.
- Any definition of fault is highly dependent on some notion of satisfaction: what does it mean that a specification or an implementation satisfies some requirement imposed by a more abstract specification? This notion of satisfaction must be sufficiently liberal to allow software to be developed in a natural manner; it must be clearly defined, and it must be expressed in such a way that it can be understood and used by ordinary systems engineers.

The remainder of this paper is divided into six sections: Section 2 gives some background on formal techniques (FTs) and computer aided systems engineering (CASE-) tools; Section 3 describes our area of specialization --- the kind of systems, specification techniques and tools we are interested in; Section 4 outlines the set-up of an experimental system development; Section 5 summarises the conclusions from this development; Section 6 discusses practical problems related to fault reporting and the comparison of different kinds of software developments; Section 7 draws some conclusions and describes future plans.

2. Formal Techniques and CASE-Tools

There is already a multitude of formal languages and notations available. Most FTs are tuned towards particular system domains or specialized application areas. They can be classified into three main categories:

- Semi-Formal Description Techniques (SFDTs).
  They are called semi-formal because the grammar and meaning of specifications expressed with the help of these techniques are not always fully defined. Typical examples of SFDTs are the Unified Modelling Language (UML) [5], the Object Modelling Technique (OMT) [6], and Object-Oriented Software Engineering (OOSE) [7].

- Formal Description Techniques (FDTs).
  The FDTs differ from the SFDTs in that their specifications have a well-defined grammar and a meaning captured in some well-understood mathematical structure. Typical examples of FDTs are the Specification and Description Language (SDL) [8], Statecharts [9], the Language of Temporal Ordering Specification (LOTOS) [10], and Message Sequence Charts (MSC) [11].
- Formal Development Methods (FDMs).
  The FDMs differ from the FDTs in their support for the logical deduction of implementations from specifications. Any FDM contains a FDT for specification purposes. Typical examples of FDMs are the Vienna Development Method (VDM) [3], Unity [12], the Temporal Logic of Actions (TLA) [13], and B [4].

It is widely recognized that the successful use of FTs in software development is not possible without effective tool support during the whole development cycle. Tools that underpin software development throughout the development process are known as CASE-tools. CASE-tools based on SFDTs and FDTs are already commercially important within several fields. Well-known examples of such CASE-tools are Cinderella, ObjectGEODE, ObjectTime, Rational Rose, Rhapsody, SDT and Statemate. Commercial CASE-tools for FDMs are also becoming more common these days. Atelier B, the B-Toolkit and the VDM Toolbox are examples of such. CASE-tools based on FTs typically offer:

- Editors specialized towards the formal specification languages on which the respective CASE-tools are based; most CASE-tools employ several specification languages --- for example, one language for the requirements capture and another language for the design.
- Syntax and type-checkers.
- Automatic generation of executable prototypes from specifications.
- User-friendly, often graphical simulators (animators) allowing bugs and inconsistencies in requirements to be discovered and mended early.
- Automatic generation of test-scenarios from requirements.
- Facilities for exhaustive exploration (also known as model checking) allowing the automatic detection of undesirable features like for instance deadlock, livelock etc.; some tools for exhaustive exploration also support verification of design with respect to requirements.
- Automatic generation of proof obligations. Such tools normally also offer facilities for the automatic and/or interactive verification of proof-obligations.
- Complete code-generation of design towards commercially important programming languages like C and C++.

3. Area of Interest

As mentioned in the introduction, the research activity on which this paper builds is concerned with several tasks. One objective is to develop strategies to measure or estimate the effect of formalization, and to try out these strategies in real system developments at the HRP. Other objectives include giving recommendations with respect to:

- the training of personell in the use of formal approaches;

- the choice and use of FTs and CASE-tools;
- the integration of FTs in a conventional software process.

This paper is concerned only with the first objective.

Industrially interesting FTs are always tuned towards particular system domains or specialized application areas. Therefore, it seemed natural to specialize our research activity to one particular application area, one particular kind of FTs, and one particular kind of CASE-tools. We decided to concentrate on:

- Systems in which interaction and communication are essential features. The components may be distributed in space, but this is not a requirement: Logical concurrency is sufficient. The systems will typically be real-time and based on object-oriented technology.
- FDTs supplemented by SFDTs when this falls natural. Formal verification of design steps in the style of FDMs will not be considered. The requirements capture will be based on sequence charts (as, for example, in MSC or UML), and the design will be based on communicating state machines (as, for example, in SDL or Statecharts). We will also use class-diagrams (as, for example, in OMT or UML).
- State-of-the-art commercial CASE-tools supporting the whole development cycle. The CASE-tools should offer editors for the chosen FTs, facilities for verification and validation based on exhaustive exploration, and complete code-generation from design to commercial platforms.

We refer to [14] for a more detailed motivation for the specialization of our research activity. [15] compares eleven specification languages, including the ones mentioned above, in a more general setting. [16] evaluates leading tools for interactive verification and exhaustive exploration.

4. Set-up of Experiment

Since we had little experience with empirical experimentation connected to software developments, we decided to try out the metrics and techniques for the collection of experimental data employed in [1] in a real system development at the HRP. [1] is concerned with FDMs like VDM and B. However, the metrics and data collection techniques of [1] carry over to our area of interest, straightforwardly. The metrics and data collection techniques of [1] are summarized below:

- Four metrics are used to compare a formal development process with a conventional one:

  Number of faults per thousand lines of code found during unit and integration tests.
  Number of faults per thousand lines of code found during validation test.
  Number of faults per thousand lines of code found during customer use.
  Person months of effort per thousand lines of code produced.

- Faults are registered throughout the development process to compare the relative effectiveness of the various stages of the formal development process. For each fault, it is recorded at which stage/activity the fault was introduced and at which stage/activity it was discovered.
- The notion of fault is defined as follows: A fault is found when a change is required to a design decision made at an earlier development stage. A design made and corrected within the same stage is not considered as a fault.

The system development in which these metrics and data collection techniques of [1] were tried out, was connected to another HRP research activity [17] aiming at the development of an analytic design methodology supported by a computerized tool, the so-called FAME tool. The task of the FAME tool is to help determine the optimal allocation of tasks between man and machine in the control of advanced technical processes. Our initial system development was to design a communication manager for this tool --- referred to as the FAME Communication Manager (FCM) in the sequel.

Five persons were involved in the FCM development:

- Two research scientists who worked out the informal system requirements and thereafter were available for questions and discussions concerning the formalisation, design and implementation of these requirements.
- Two systems engineers responsible for the actual development.
- One FT expert who gave advice on the use and integration of FTs, and supervised the collection of experimental data.

With the exception of the latter, none of the participants had earlier experience in the use of FTs. Moreover, the FT expert had used FTs only in system developments of academic size. None of the participants had background from the use of commercial CASE-tools. The following FTs were employed:

- The use-case diagrams of UML, the sequence charts of MSC --- including hierarchical ones, and OMT class-diagrams to capture the abstract requirements.
- SDL with C embedded to describe the design.

The tool support was SDT (from the Swedish company Telelogic). We employed the SDT facilities for editing, animation, exhaustive exploration and code-generation. The FCM development was based on a waterfall process. This was in accordance with [1] and also consistent with the software quality assurance manual at the HRP [18] (which is based on the international standard ISO 9000-3).

5. Preliminary Results from Experiment

As already mentioned, one objective with the FCM development was to test out the metrics and data collection techniques of [1] in a real system development. Thereby we hoped to gain a better understanding of their suitability and usefulness in a practical context. The system development was successfully completed and resulted in about 10 000 lines of C-code (adjusted number correcting for automatic code generation). The experimental results (together with the experiences and recommendations connected to the other objectives of the research activity) will be published in a technical report [19] currently in preparation. The main conclusions with respect to the experimentation is summed up below:

- The experimental data gained from the FCM development is not very valuable as such. There are several reasons for this:

  The FCM is a prototype to be used in a scientific experiment. It was therefore not tested out and will not be maintained in the same way as a commercial product.

  A very large percentage of the person hours was invested in learning to use the FTs and, in particular, SDT.

  There was not a clear separation between the scientists responsible for the experiment and the systems engineers who carried out the software development.

Experimental data collected from only one, relatively small software development is of course insufficient.

- We identified several problems connected to metrics, fault counting and notions for satisfaction. The most important of these are discussed below.

6. Discussion

To investigate the effects of formalization by experimental means is a very challenging task. This we already knew at the start up. The FCM development made this even clearer; in particular, it highlighted a number of problem areas. Some very important ones are discussed below.

6.1 *Measuring the right thing*

To compare formal and conventional development processes, great care must be taken to ensure that we measure the effects of formalization, and not the effects of something else. One central question is: To what degree should the activities in the formal development process be mirrored in the conventional one, and vice versa? The answer depends, of course, on the exact purpose of the experiment. This purpose must be clearly defined, and both the formal and conventional developments must be configured with respect to this purpose. If we are interested only in the effects of formalization, it seems the experiments should be organized in such a way that the formal and conventional developments differ with respect to the formalization and the immediate effects of the formalization only.

For example, in the FCM development, the formal and informal requirements specification were validated against each other based on a review where all involved parties were present and forced to decide whether they accepted the specified requirements as correct or not. In those cases where a mistake was identified, the involved parties were required to agree on how it should be corrected. This proved very efficient, and considerably improved both the informal and the formal requirements specification. Such a review would of course be helpful also in a conventional development, although there would then be only an informal requirements specification on which the review could be based. In order to compare a formal development process with such a review at the requirements level with a conventional one, it seems reasonable that also the conventional process has such an activity.

6.2 *Integration in the software process*

To define a fault is not as easy as one might think. Anyone with at least some experience from software engineering knows that both specification and programming involves a lot of experimentation based on trials and failures. Clearly, we need a definition of fault that does not force the systems engineers to report on the large number of alterations resulting from this kind of activities --- activities that are desirable and play an important role in any practical software development. The fault definition of [1], from now on referred to as the definition of fault, seems well-suited in this respect. Since a change to a design decision made and corrected within the same development stage does not count as a fault, the creative experimentation mentioned above is not constrained.

The FCM development was based on a waterfall process. Initially, we intended to complete each stage before the next was started up, since this is consistent with the definition of a fault. In the FCM development the requirements stage was completed before the other stages were initiated. However, for practical reasons, the design and implementation stages overlapped in time. Does this indicate that the definition of fault has to be

modified? We do not think so. The problem was not really the definition of fault, but rather our very strict implementation of the waterfall process. The overlap of the development stages occurred because the development of clearly separated sub-components progressed at different speeds. In our next experimental system development we will try to adapt our routines for the collection of faults to an iterative, more flexible development process in the tradition of [20], [7].

6.3 *Satisfaction*

The dependency on the development process is, however, not the only problematic aspect connected to fault reporting. We may view the activities within one development cycle as a sequence of steps from one specification to the next where the final implementation is seen as just another specification written in a format suitable for efficient mechanized execution. In order to decide whether we have found a fault or not, we need to know what it means for a specification to satisfy a requirement or design decision imposed by a more abstract specification. In other words, we need a clear understanding of the term “satisfy” --- or, more scientifically, a well-defined notion of satisfaction.

The FCM development made it perfectly clear that in the context of MSC, OMT, SDL and UML there is no generally accepted notion of satisfaction. In fact, we are not aware of any design methodology based on this kind of FTs that defines the notion of satisfaction in a, for our purposes, fully satisfactory manner.

In the context of step-wise software development the term “refinement” is often used instead of “satisfaction”. A specification A can be refined into a specification B if the specification B satisfies A. Within the FDM community there is a large literature on the formalization of refinement principles. Pioneering papers were published in the early 70ies [21], [22], [23]. These papers were mainly concerned with the development of sequential non-interactive software. Recent proposals concerned with the formalization of refinement are directed towards more complex systems --- in particular, systems based on interaction and concurrency.

FDMs for interactive and concurrent systems like TLA [24] and Focus [25] distinguish between three notions of refinement of increasing generality, namely property refinement, interface refinement and conditional refinement. Property refinement allows a specification to be replaced by another specification that imposes additional functional and non-functional properties. Property refinement supports step-wise requirements engineering and incremental system development in the sense that new requirements and properties can be added to the specification in the order they are captured and formalized. Property refinement can be used to reduce the number of behaviours allowed by a specification. Property refinement does not allow behaviours to be added; nor does it allow modifications to the external interface of a specification. Hence, property refinement characterizes what it means to reduce under-specification.

If software developments were based on property refinement alone, the inability to change the external interfaces would basically enforce the same level of abstraction throughout the whole development. As a result, the developments would often be unnecessarily complex and inflexible. To avoid this, we may use the notion of interface refinement.

Interface refinement is a generalization of property refinement allowing a specification to be replaced by one that has a different syntactic interface under the condition that all the runs of the refined specification can be translated into runs of the original one. Interface refinement supports the replacement of abstract data types by more implementation dependent ones, it allows the granularity of interaction to be modified, and it supports changes to the communication structure in the sense that, for example, one channel can

be represented by several channels, or vice versa. Interface refinement also captures what it means to adapt the interface of an already completed system to allow reuse in another environment or software development.

In the final phases of a system development, many implementation dependent constraints and restrictions must be considered. This may require the introduction of additional environment assumptions. Unfortunately, property and interface refinement do not support this. Instead, we may use the notion of conditional refinement, which can be understood as property or interface refinement with respect to additional conditions or assumptions about the environment in which the specified component is supposed to run. Conditional refinement supports the transition from system specifications based on unbounded resources (unbounded buffers, memories etc.) to system specifications based on bounded resources. It allows purely asynchronous communication to be replaced by handshake or time-synchronous communication, and it supports the introduction of exception handling.

In TLA specifications are logical formulas written in linear time temporal logic. If A is the more abstract specification and B is the more concrete specification then the three notions of refinement described above are all captured by the following formula:

$$C \wedge B \Rightarrow A$$

Hence, in order to verify a step of refinement it is enough to show that the abstract specification A is a logical implication of the logical conjunction of the concrete specification B and some additional formula C. In the case of property refinement, C is logically equivalent to true and therefore not needed; in the case of interface refinement C characterizes the relationship between the concrete and abstract interfaces; in the case of conditional refinement, C also formalizes the additional environment assumptions. Hence, if we think of A as the abstract requirement specification and B as the concrete implementation, then C describes their relationship and gives together with A the required system documentation.

In Focus this notion of refinement can be described graphically as in Figure 1. The downwards mapping D translates the abstract input to the concrete input; the upwards mapping U translates the concrete output to the abstract output; the condition C imposes additional environment assumptions; the arrow from the output of A to C indicates that the assumption about future inputs may depend on what A has produced as output so far.

The close relationship between Focus and SDL is well-documented in the literature [26], [27]; in fact, Focus can be seen as a formal foundation on which a SDL based methodology can be built. For example, the definition of refinement given above can easily be translated into a SDL setting (as proposed by [28]). Assume A and B are SDL specifications, then B is a refinement of A if we can specify C, D and U in SDL such that for any abstract input history generated by C, the abstract output history generated by the block consisting of the piped composition of the SDL specifications D, B, U is a possible output history of A.

The interesting question at this point is of course: Can we base our definition of fault on this notion of refinement? This is not quite clear. Firstly, this definition is certainly not sufficient on its own. The reason is that it considers only the external black-box behaviour --- in other words, requirements on the externally observable behaviour. Hence, so-called glass-box requirements, namely requirements on the internal design that have no effect on the external behaviour, are not captured. This may have nontrivial consequences.

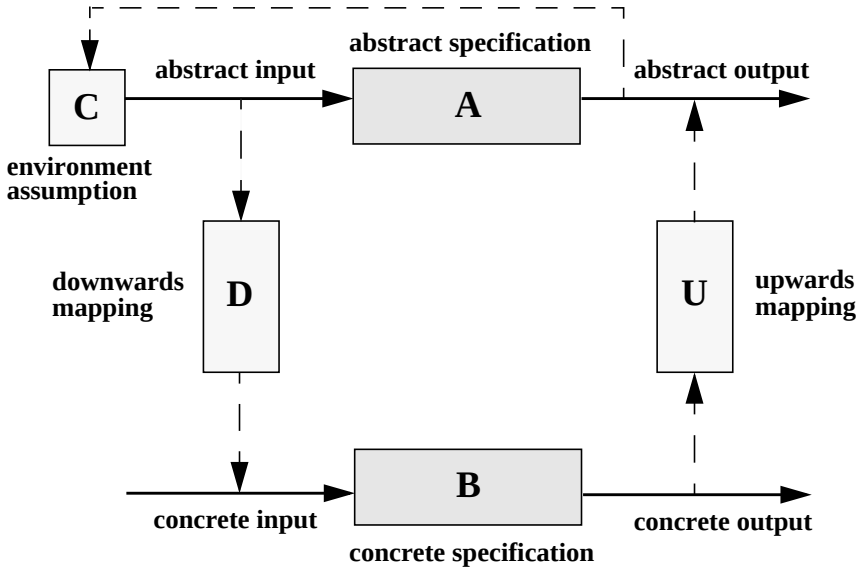

Figure 1: Graphical representation of refinement

For example, in the FCM development we recorded, on the one hand, a very low number of design faults with respect to the external black-box behaviour of the requirements specification. Hence, if we define a notion of satisfaction that considers only the external black-box behaviour, this could be interpreted as experimental evidence for the claim that formalization reduces the number of design faults (there are of course a wide range of alternative explanations). On the other hand, the organization and relationship of classes and objects in our design specification was not at all in accordance with the requirements imposed by the class-diagrams of the requirements specification. Hence, if we had used a notion of satisfaction requiring the glass-box constraints imposed by these class-diagrams to be mirrored at the design level, we would have recorded a very high number of design faults, and our experiment could be interpreted as experimental evidence for the invalidity of the claim that formalization reduces the number of design faults (again there are of course alternative explanations).

Secondly, although we believe that this definition, with some minor generalizations, is well-suited to describe what it means for an SDL specification to refine the black-box behaviour of another SDL specification, we are less convinced that this definition captures the relationship between two MSC specifications, or between a MSC specification and a SDL specification. The Focus definition works well for SDL because a SDL specification, as a specification in TLA or Focus, describes the set of all allowed behaviours. Under the assumption that C, D and U have been specified correctly, we are satisfied with the SDL specification B from a black-box point of view, if its set of behaviours is a subset of the behaviours allowed by the SDL specification A with respect to C, D and U. The SDL specification A may, however, allow additional behaviours. An MSC specification, on the other hand, describes a set of behaviours (example runs) the system is required to support --- at least this seems to be consistent with how the MSC language is used, traditionally.

The problem areas identified above are important. They are of course not the only ones. For example, the issue of defining and selecting the right metrics on size, effort, and the various quality characteristics has not been mentioned. How to motivate the systems

engineers to provide the required fault reports etc. in a consistent manner, is another slightly different, but nevertheless very important, issue.

7. Conclusions

The lack of experimental evidence for scientific claims seems to be a general problem within computer science. [29] argues: “There are plenty of computer science theories that haven’t been tested. For instance, functional programming, object-oriented programming, and formal methods are all thought to improve programmer productivity, program quality or both. It is surprising that none of these obviously important claims have ever been tested systematically, even they are 30 years old and a lot of effort has gone into developing programming languages and formal techniques.” That the level of experimental evidence is much lower in computer science than in the more classical sciences has been confirmed by several studies. In a random sample of all the papers the ACM published in 1993, [30] found that 40 percent of the papers with claims that needed empirical support had none at all. In software related journals, this fraction was 50 percent. The same study also analysed a non-computer science journal, Optical Engineering, and found that the fraction of papers lacking quantitative evaluation was merely 15 percent. [31] found similar results.

To set up experiments and collect experimental data is not difficult. To do this in a way such that interesting and scientifically valid conclusions can be drawn is, however, challenging. This was highlighted by the FCM development. [31] classifies experimental approaches into three main categories:

- Observational method --- collects relevant data as a software development develops.
- Historical method --- collects data from developments that have already been completed using existing data.
- Controlled method --- provides for multiple instances of an observation in order to provide for statistical validity of results.

The method employed in the experiment on which this paper reports belongs to the first category. This does not necessarily mean that future activities within our research activity will be restricted to observational methods.

One important objective of our research activity is to come up with a strategy allowing at least some aspects connected to the effects of formalization to be measured. Hence, what we expect to achieve are better techniques and strategies for this kind of experiments.We will try out these techniques and strategies in software developments at the HRP. The empirical data collected from these HRP experiments will, however, hardly be sufficient as experimental evidence for very general claims. There are several reasons for this:

- The HRP is a research project and not a software house. System developments at the HRP normally result in prototypes that are used for scientific purposes only. The resulting prototypes are not tested out and maintained in the same way as the products of commercial tool vendors. Hence, the data collected from our experiments will not give a realistic picture (except, of course, for HRP like research projects).
- System developments at the HRP are usually rather small, and there are not very many of them.
- There should be a clear separation between the scientists who set up and run the experiments, and the system engineers who carry out the developments. This is not easily achievable at the HRP.

Hence, in order to provide significant experimental evidence with respect to the effects of formalization in an industrial context, we need to find commercial partners willing to provide the required amounts of experimental data.

Acknowledgements
Andreas Bye, Tor Steinar Brendeford and Håkon Sandmark participated in the development of the FCM. We are grateful for their positive and open-minded attitude. Børge Haugset, Tore Willy Karlsen and Helena Olausson aided us in the evaluation of CASE-tools. Øystein Haugen, Terje Sivertsen and Wenhui Zhang read an earlier version of this paper and provided valuable feedback.

References

[1] BICARREGUI, J., DICK, J., WOODS, E. Quantitative analysis of an application of formal methods. In Proc. FME96, LNCS 1051, pages 60-73, 1996.

[2] DRAPER, J., TREHARNE, H., BOYCE, T., ORMSBY, B. Evaluating the B-method on an avionics example. In Proc. DAISA96, pages 89-97, European Space Agency, 1996.

[3] JONES, C.B. Systematic software development using VDM, second edition. Prentice Hall, 1990.

[4] ABRIAL, J.R. The B book: assigning programs to meaning. Cambridge University Press, 1996.

[5] UML proposal to the object management group, version 1.1, September 1, 1997.

[6] RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., LORENSEN, W. Object-oriented modelling and design.Prentice Hall, 1991.

[7] JACOBSON, I., CHRISTERSON, M., JONSSON, P., OEVERGARD, G. Object-oriented software engineering --- a use case driven approach, Addison-Wesley, 1992.

[8] Recommendation Z.100 - CCITT specification and description language (SDL). ITU, 1993.

[9] HAREL, D. STATECHARTS: a visual formalism for complex systems. Science of Computer Programming 8:231-274, 1987.

[10] LOTOS- A formal description technique based on the temporal ordering of observational behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

[11] Recommendation Z.120 - Message Sequence Chart (MSC). ITU, 1996.

[12] CHANDY, K.M., MISRA, J. Parallel program design, a foundation. Addison-Wesley, 1988.

[13] LAMPORT, L. The temporal logic of actions. ACM TOPLAS, 16:872-923,1994.

[14] STØLEN, K., KARLSEN, T.W., MOHN, P., SANDMARK, H. Using CASE-tools based on formal methods in real-life system developments of distributed systems. HWR-522, OECD Halden Reactor Project, 1998.

[15] STØLEN, K. Formal specification of open distributed systems --- overview and evaluation of existing methods. HWR-523, OECD Halden Reactor Project, 1998.

[16] ZHANG, W. Verification techniques and tools for formal software development. HWR-526, OECD Halden Reactor Project, 1998.

[17] BYE, A., BRENDEFORD, T.S., HOLLNAGEL, E., HOFFMANN, M., MOHN, P. Human-machine function allocation by functional modelling - FAME - a framework for systems design. HWR-513, OECD Halden Reactor Project, 1998.

[18] Software Quality Assurance Manual, Version 1.0 - 1/6/95. IFE, 1995.

[19] MOHN, P., SANDMARK, H., STØLEN, K. Experiences from the development of the FAME communication manager using the CASE-tool SDT. To appear as HWR, OECD Halden Reactor Project, 1999.

[20] BOEHM, B.W. A spiral model of software development and enhancement. IEEE Computer, 21:61-72, 1988.

[21] MILNER, R. An algebraic definition of simulation between programs. In Proc. 2nd International joint conference on artificial intelligence, 1971.

[22] HOARE, C.A.R. Proof of correctness of data representations. Acta Informatica, 1:271-282, 1972.

[23] JONES, C.B. Formal development of correct algorithms: an example based on Earley's recogniser. In Proc. ACM conferences on proving assertions about programs, SIGPLAN Notices, 7:150-169, 1972.

[24] ABADI, M., LAMPORT, L. Conjoining specifications. ACM TOPLAS, 17:507-533, 1995.

[25] BROY, M., STØLEN, K. Focus on system development. Book manuscript, May 1998.

[26] BROY, M. Towards a formal foundation of the specification and description language SDL. Formal Aspects of Computing, 3:21-57, 1991.

[27] HOLZ, E., STØLEN, K. An attempt to embed a restricted version of SDL as a target language in Focus. In Proc. Forte94, pages 324-339, Chapman & Hall, 1994.

[28] HAUGEN, Ø. Practitioners' verification of SDL systems. Dr. Scient thesis, University of Oslo, April, 1997.

[29] TICHY, W.F. Should computer scientists experiment more? IEEE Computer 31:32-40, 1998.

[30] TICHY, W.F., LUKOWICS, P., PRECHELT, L., HEINZ, E. A. Experimental evaluation in computer science: a quantitative study. Journal of Systems and Software 28:9-18, 1995.

[31] ZELKOWITZ, M.V., WALLACE, D.R. Experimental models for validating technology. IEEE Computer 31: 23-31, 1998.