Measuring the Effect of Formalization

*Ketil Stølen, Peter Mohn**
OECD Halden Reactor Project, Institute for Energy Technology
Postbox 173, N-1751, Halden, Norway
Email: {ketils,peterm}@hrp.no
**)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. Based on the experiences from a first experimental development, we discuss a number of technical issues; in particular, problems connected to metrics based on error reports. First of all, what is an error? Secondly, how should the error counting be integrated in the development process? Thirdly, any reasonable definition of error 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? Our answers are so far only partly satisfactory.

Keywords
error counting, experiment, formalization, metrics, refinement, satisfaction, software process

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 reduces the number of design errors because the developers obtain a better understanding of what the requirements actually mean.
- Formalization allows errors to be identified early in the development process --- in the following we refer to this argument as the “bugs discovered early” hypotheses.
- 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 costs? These questions are not easily answered. The HRP has recently initiated a research activity with the objective to address this kind of issues.

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 error reports:

- In order to report on errors we need a clear definition of what an error 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 errors. Moreover, this definition should not depend upon a software process that does not mirror how software is developed in practise.
- Any definition of error is highly dependent on some notion of satisfaction: what does it mean that a specification or 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 system engineers.

The paper is divided into six main sections: Section 2 describes our area of specialization --- the kind of systems, specification techniques and tools we are interested in; Section 3 summarizes experiences from a first system development; Section 4 discusses practical problems related to error reporting and the comparison of different kinds of software developments; Section 5 draws some conclusions and describes future plans; in particular, what we expect to achieve, and what we hope to achieve; Section 6 provides a list of references.

2. AREA OF INTEREST

There are many different kinds of formal techniques (FTs) intended to support software development. Since there are also many different kinds of systems, this is hardly surprising. That there is no FT that is well-suited for the description of all kinds of software systems is also quite obvious; after all, the required generality of such a universal FT would make it impossible to overview and use in a practical context. Hence, practical FTs are always tuned towards particular application areas. For the very same reason, it seemed natural to specialize our research activity to one particular application area, one particular kind of FTs, and one particular kind of tools.

We concentrate on systems in which interaction and communication is essential; the components may be distributed in space, but this is not a requirement: we also consider systems with only logical concurrency. The systems will typically be real-time and based on object-oriented technology.

We distinguish between three disjoint classes of FTs: semi-formal description techniques (SFDTs), formal description techniques (FDTs) and formal development methods (FDMs).

- The SFDTs are semi-formal in the sense that the syntax and semantics of specifications expressed in SFDTs are not fully defined. Typical examples of SFDTs are diagrams for object-oriented modelling. Almost any commercial software/hardware developer employs SFDTs in one form or another.
- The FDTs differ from the SFDTs in that the specifications expressed in FDTs have a well-defined syntax and, in addition, a semantics captured by well-understood mathematical notations. Within the fields of telecommunications and embedded systems FDTs already play an important commercial role.
- The FDMs differ from the FDTs in their support for the logical deduction of implementations from specifications. A FDM contains a FDT for specification purposes; this FDT is normally quite mathematical to allow conventional mathematical and logical theories to be exploited in the step wise deduction of implementations. FDMs have not yet received much industrial attention; moreover, it seems that if they are employed in an industrial context, they are mainly used as FDTs.

We concentrate our research activities on FDTs; we will use SFDTs as a supplement 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 [1] or UML [2]); the design will be based on communicating state machines (as, for example, in SDL [3] or Statecharts [4]); we will also use class-diagrams (as, for example, in OMT [5] or UML).

The usefulness of FDTs in practical system development is highly dependent on adequate tool support. We will use state-of-the-art CASE-tools supporting the whole development cycle. The CASE-tools should offer editors for the chosen FTs, facilities for verification and validation, and complete code-generation from design to commercial platforms.

We refer to [6] for a more detailed motivation for the specialization of our research activity. [7] compares eleven specification languages, including the ones mentioned above, in a more general setting.

3. EXPERIMENTAL DEVELOPMENT

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 [8], [9]. They report on small scale experiments where conventional developments are related to developments based on FDMs. We briefly discuss their results in Section 5. Since we had little experience with this kind of experiments, we decided to start our investigations by testing out the metrics and techniques for the collection of experimental data proposed by [8] in a real system development at the HRP. [8] is concerned with FDMs like VDM [10] and B [11]. However, the proposals of [8] with respect to metrics and data collection carry over to our application area, straightforwardly. These proposals of [8] are summed up below:

- To compare a formal development process with a conventional one, four metrics are proposed:

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

- To compare the relative effectiveness of the various stages of the formal development process, and thereby, for example, try to validate the "bugs discovered early" hypotheses, errors were registered throughout the development process; for each error, it was recorded at which stage/activity the error was introduced and at which stage/activity it was discovered.
- The notion of error is defined as follows: an error 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 an error.
- The developments were based on a waterfall process.

Our initial system development to test of these proposals of [8] was carried out within the frame of another HRP research activity [12] aiming at the development of an analytic design methodology supported by a computerized tool, the so-called FAME tool, with the objective to determine the optimal task allocation between man/machine in the control of advanced technical processes. Our task was to design a communication manager for this tool --- referred to as the FCM in the following.

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, neither of the participants had earlier experience in the use of FTs.

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 embedded C to describe the design.

The tool support was SDT (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 [8] and also consistent with the software quality assurance manual at the HRP [13].

As already mentioned, the objective with the FCM development was to test out the proposals of [8] with respect to metrics and data collection in a real system development. Thereby we hoped to gain a better understanding of their suitability and usefulness in a practical context. In this respect, the FCM development was a success. As explained in Section 4, we identified several problem areas.

The experimental data gained from the FCM development is not very valuable as such. First of all, we developed a prototype that is 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. Secondly, a very large percentage of the person hours was invested in learning to use the FTs and, in particular, SDT. We will therefore not report on the experimental data here. However, the system development was successfully completed and resulted in about 10 000 lines of C-code (adjusted number correcting for automatic code generation).

4. DISCUSSION

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

4.1 Measuring the right thing

In order 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. For example, in the FCM development, the formal and informal requirements specification were validated against each other based on a review were all involved parties were present and forced to decide whether they accepted the requirements specifications 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. Hence, in order to compare a formal development process with such a review at the requirements level with a conventional one with respect to the effects of formalization, it seems reasonable that also the conventional process has such an activity.

Of course, such a review is only one of many different system development activities. The 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 on the exact purpose of the experiment. This purpose has to be clearly defined, and both the formal and conventional developments have to be configured with respect to this purpose. If we are only interested in the effects of formalization, it seems the experiments should be organized in such a way that the formal and conventional developments differ only with respect to the formalization and the immediate effects of the formalization.

4.2 Integration in the software process

To define an error 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 error 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 error definition of [8], from now on referred to as the definition of error, 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 an error, 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 was basically required by the definition of error. In the FCM development the requirements stage was completed before any of the other stages were started up. However, for practical reasons, the design and implementation stages overlapped in time. Does this mean that the definition of error has to be modified? We do not think so. In our opinion, the problem was not really the definition of error, 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 errors to an iterative, component based development process in the tradition of [14], [15].

4.3 Satisfaction

The dependency on the development process is, however, not the only problematic aspect in connection with error reporting. In the following we think of the final implementation as just another specification written in a format suitable for efficient mechanized execution. In that case, we may view the activities within one development cycle as a sequence of steps from one specification to the next. In order to decide whether we have found an error 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 satisfactory manner.

In the context of step wise software development it is more common to talk about refinement 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 different notions of refinement. Pioneering papers were published already in the early 70ties [16], [17], [18]. These papers were only 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 [19] and Focus [20] 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 that allows 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 hand-shake 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 nice thing about this definition is that it can easily be translated into a SDL setting. 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 error 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. Hence, glass-box requirements on how A is to be organized internally are not captured.

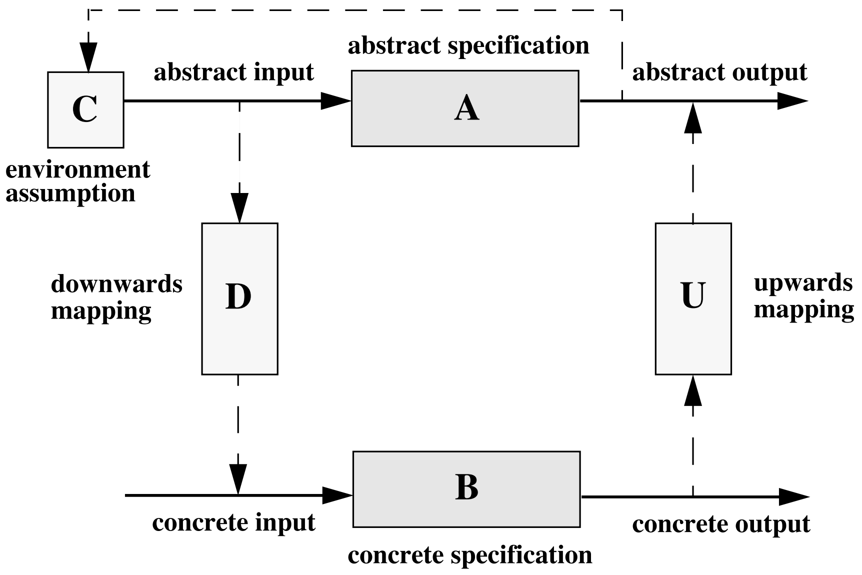

Figure 1: Graphical representation of refinement

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 has not been mentioned. How to motivate the systems engineers to provide the required error reports etc. in a consistent manner, is another slightly different, but nevertheless very important, issue?

5. CONCLUSIONS

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, in many contexts very challenging. This was highlighted by the FCM development. The possibilities for mistakes and undesirable effects are many. For example, in the FCM development we recorded, on the one hand, a very low number of design errors 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 errors (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 errors, and our experiment could be interpreted as experimental evidence for the invalidity of the claim that formalization reduces the number of design errors (again there are of course alternative explanations). We just mention this to stress that this kind of experiments must be carefully set up and monitored --- and moreover, one should not be too ambitious.

We found the proposals of [8], [9] both helpful and inspiring. Both papers present results from small scale experiments and report on evidence in favour of increased formalization. However, as pointed out in [8]: “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.”

As we see it, the task of our research activity can only be to come up with a strategy that allows 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 certainly test 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 interesting claims. There are two main reasons for that. Firstly, as already mentioned: 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. Our system developments are usually rather small, there are not very many, and 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). Secondly, there should be a clear separation between the scientists that set up and run the experiments, and the system engineers that carry out the developments. This is not easily achievable at the HRP. This sums up what we expect to achieve.

What we hope to achieve is to provide significant experimental evidence with respect to the effects of formalization, but this requires that we find commercial partners that are willing to try out our proposals and provide sufficient amounts of experimental data.

ACKNOWLEDGMENTS

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.

6. REFERENCES

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

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

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

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

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. Object-oriented modelling and design. Prentice Hall, 1991.

[6] K. Stølen, T. W. Karlsen, P. Mohn, Håkon Sandmark. Using CASE-tools based on formal methods in real-life system developments of distributed systems. HWR-522, OECD Halden Reactor Project, 1998.

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

[8] J. Bicarregui, J. Dick, and E. Woods. Quantitative analysis of an application of formal methods. In Proc. FME96, LNCS 1051, pages 60-73, 1996.

[9] J. Draper, H. Treharne, T. Boyce, B. Ormsby. Evaluating the B-method on an avionics example. In Proc. DAISA96, pages 89-97, European Space Agency, 1996.

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

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

[12] A. Bye, T. S. Brendeford, E. Hollnagel, M. Hoffmann, and P. Mohn. Human-machine function allocation by functional modelling - FAME - a framework for systems design. HWR-513, OECD Halden Reactor Project, 1998.

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

[14] B. W. Boehm. A spiral model of software development and enhancement. IEEE Computer, pages 61-72, May 1988.

[15] I. Jacobson, M. Christerson, P. Jonsson, G. Oevergaard. Object-oriented software engineering --- a use case driven approach, Addison-Wesley, 1992.

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

[17] C. A. R. Hoare. Proof of correctness of data representations. Acta Informatica, 1:271:282, 1972.

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

[19] M. Abadi, L. Lamport. Conjoining specifications. ACM TOPLAS, 17:507-533, 1995.

[20] M. Broy, K. Stølen. Focus on system development. Book manuscript, May 1998.