Experiences from Using MSC, UML and SDL in the Development of the FAME Communication Manager[1]

Ketil Stølen
SINTEF Telecom and Informatics, Oslo, Norway
Peter Mohn
Deimos AG, Zurich, Switzerland

abstract

This paper presents experiences from using formal description techniques and the CASE-tool SDT[2] in the development of a communication manager. UML use-case and class diagrams together with MSCs were used for requirements capture. The design specification was written in SDL and C. The final implementation in C was to a large extent generated automatically from the design specification and two additional SDL specifications relating the abstract interface of the design to the concrete interface of the implementation. The paper describes the overall development process and the various activities within each development stage. For each stage the lessons learnt are summarized and discussed.

1. Introduction

The objective with this paper is to present experiences, results and conclusions from employing FDTs (Formal Description Techniques) and state of the art CASE tool technology in the development of the FAME Communication Manager (FCM). This development was carried out in full at the Institute for Energy Technology (IFE) in Halden, Norway. The FCM development was started up in September 1997 and completed in July 1998. Both the requirements and the design specifications were formalized, and the implementation in C was to a large extent generated automatically from the formal design specification. The full version of this paper is available as a technical report [SMST99]; more background and results from technology evaluations are provided by [SKMS98], [Stø98].
The remainder of the paper is divided into six sections. Section 2 describes the system to be developed, the background of the participants, and the development process. Sections 3, 4 and 5 present the activities within the requirements, design and implementation stages and summarize experiences. Section 6 provides statistics based on data collected during the FCM development. Section 7 discusses and evaluates our experiences, and presents the main conclusions.

2. Outline of Development

This section gives an overview of the FCM development; in particular, we describe the main architecture of the FAME Tool, the background of the personnel that took part in the FCM development, and the development process.

2.1 System to be Developed

The research activity Function Allocation Methods (FAME) aims at the development of an analytic design methodology to find the optimal task allocation between man and machine in

---

[1] The research on which this paper reports was carried out in full at the Institute for Energy Technology (IFE) in Halden, Norway. The first author was at that time employed at IFE. The second author was on leave from the Swiss Federal Nuclear Safety Inspectorate, Villingen, Switzerland.
[2] The system development on which this paper reports used mainly the SDT3.2 release and only partly SDT3.3. All comments concerning SDT facilities in this paper refer to these two SDT releases only.

control of advanced technical processes. [BBHHM98] describes the background and motivation for FAME. Scenario simulations are seen as a central part of the FAME approach; hence, a tool for integrated operator/process-simulations is needed. As illustrated by Figure 1, the FCM component is a subcomponent of the FAME Tool. The FAME Tool consists of four main components in addition to the FCM:

- Operator Simulator: Simulates the behaviour of operators controlling a physical process.
- Automatics Simulator: Simulates the behaviour of an idealised "automatic" operator.
- Process Simulator: Simulates the behaviour of a physical process.
- Man Machine Interface (MMI): Used to initialise and configure simulations.

The purpose of the FCM is to set up connections and to manage the exchange of signals between the Automatics Simulator, the Operator Simulator and the Process Simulator. Signals sent from one of these components to the FCM are supplemented with additional information and forwarded to the two other components according to rules specified in scenarios defined by a user. The user may set up scenarios and control the execution of the FAME Tool via the MMI.

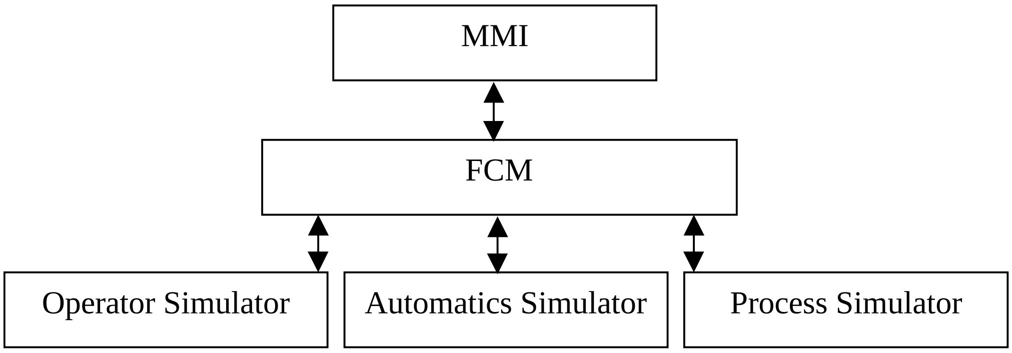

Figure 1 Architecture of the FAME Tool

Note that the MMI, the Operator Simulator and the Automatics Simulator were developed independently of the FCM. The Process Simulator was already available at the start-up of the FCM development. The FCM development activities on which this paper report were completed in July 1998. The integration of the FCM and the Process Simulator had at that point been completed. The integration of the FCM with the three other components still remained.

2.2 Background of the Personnel Taking Part

Five persons were involved in the FCM development:

- Customers: Two research scientists responsible for the design and development of the FAME Tool. Their tasks in the FCM development were to work out the informal requirements and thereafter be available for questions and discussions concerning their formalisation, design and implementation. Both research scientists were experienced programmers and system developers. They are in the following referred to as the customers.
- Developers: Two systems engineers responsible for the actual development. They were supposed to formalise the informal requirements and thereafter develop the FCM using the selected techniques, CASE-tool and software process. They are in the following referred to as the developers. The two developers had no earlier experience with formal techniques and CASE-tools.

- FDT Expert: A research scientist, referred to as the FDT expert, responsible for the research activities within the FCM development in general, and for supervising the use of FDTs and tools in particular.

2.3 Development Process

The software process of the FCM development was to a large extent determined by two factors. First of all, since the development was carried out at IFE, the recommendations of the IFE Software Quality Assurance Manual [SQA1.0] were followed. It distinguishes between 10 categories of software developments of which 5 are believed to satisfy the requirements in [ISO9000-3]. The FCM development was classified as a category 8 development. Secondly, since we had decided to base our development on the CASE-tool SDT, we tried to integrate the methodology on which SDT builds, namely the SDL-Oriented Object Modelling Technique (SOMT) [SDT3.2]. Luckily, the SQA-manual and SOMT combined well.

The FCM development was divided into three main stages, namely the requirements, design and implementation stages. The first stage was completed before the two other were started up. The design and implementation stages were partly overlapping in time. Each stage involved several activities. The various activities within each stage are described below:

- Stage 1 (Requirements):
  There were two main deliverables:
  Informal Requirements Specification (IRS): The IRS is written in English and structured in accordance with the recommendations of the SQA-Manual [SQA1.0] for a category 8 development. The IRS describes the desired black-box behaviour of the FCM; it imposes, in addition, a number of design and implementation related constraints. The IRS was prepared by the customers and checked for inconsistencies and mistakes by the developers. It was formally accepted by the developers.
  Formal Requirements Specification (FRS): The FRS aims at describing the abstract black-box behaviour of the FCM. Thus, the design and implementation related requirements of the IRS are not formalised in the FRS. The FRS was prepared by the developers and checked for inconsistencies and mistakes by the customers. It was formally accepted by the customers.
  The validation of the FRS was based on a formal walk-through.
- Stage 2 (Design):
  There was one main deliverable:
  Formal Design Specification (FDS): It describes an FCM Kernel with interfaces towards the MMI, the Operator Simulator, the Automatics Simulator, and two components called the File Operations Component and the Signal Wrapper. The File Operations Component is concerned with the concrete file operations; the Signal Wrapper relates abstract signals addressed to the Process Simulator to their concrete representation within the Process Simulator. Hence, the FCM was decomposed into three components, namely the FCM Kernel, the File Operations Component and the Signal Wrapper. The File Operations Component and the Signal Wrapper were specified and designed during the implementation stage. They relate the abstract design represented by the FCM Kernel to one particular concrete realisation. We may map the FCM Kernel to other concrete realisations by redefining the File Operations Component and the Signal Wrapper. The FDS was formally accepted by the Customers.
  The validation of the FDS was based on various tools for validation and demonstrations for the customers.
- Stage 3 (Implementation):
  There were three main deliverables:

Specification of the File Operations Component: It describes the concrete file operations. Hence, it maps the abstract file operation signals of the FCM Kernel to their concrete realisations, and the other way around.

Specification of the Signal Wrapper: It maps abstract signals of the FCM Kernel addressed to the Process Simulator to their concrete representations in the Process Simulator, and the other way around.

Integrated Implementation (II): The II is coded in C and consists of the FCM Kernel, the File Operations Component and the Signal Wrapper. The code for these three components was automatically generated from their respective specifications. The communication between the three components was implemented directly in C. The II was connected to the Process Simulator accordingly.

The validation of the II was based on integration testing.

3. Requirements (Stage 1)

The FRS consists of UML use-case and class diagrams in addition to sequence charts expressed in MSC. The IRS is a document of 24 pages written in English. The document was produced by the customers. The document consists of text illustrated by figures showing the overall structure of the FAME Tool. The document is divided into ten sections including an introduction to the research activity FAME, a list of definitions and terms, and sections describing the target system environment, the system requirements, requirements to the system environment and requirements to the development project.

For the FCM development the section on the system requirements was the most relevant. It explains the purpose of the FCM, lists a number of FCM functions and describes in detail the interface to the other components of FAME. In addition, a possible user interface for building scenarios and the controlling the FAME tool is sketched.

The specification describes more the internals of the FCM than its external behaviour. Most of the functionality described is concerned with how the FCM behaves internally. The developers missed description of certain external aspects of the FCM. For example, concrete descriptions of scenarios involving all the components of the FAME tool.

3.1 Formal Requirements Specification

Class diagrams and MSCs were drawn using specialized editors provided by SDT. Use-case diagrams were drawn directly in Framemaker. The transformation of informal requirements into formal requirements expressed by class-diagrams, use-case diagrams and MSCs was a challenging task. The IRS was by no means complete when the formalization was started up - only a first draft was available. The FRS and IRS thereafter developed in parallel - although the IRS was most of the time a little ahead of the FRS.

After having identified the use-cases the developers described their dynamic behaviour with the help of MSCs. The IRS is very detailed and contains many requirements only concerned with the internals of the FCM. In particular, it puts much more emphasis on the internal than the external behaviour. In the FRS the intention was to describe the external behaviour of the FCM - in other words, the behaviour of the FCM as seen from the outside. As a consequence of this, many informal requirements describing the internal behaviour of the FCM were not part of the FRS.

The IRS contains a very low-level protocol for the communication between the FCM and the Process Simulator. This protocol was not described in detail in the FRS. Instead it was based on a more abstract communication protocol that could also be used for other process simulators.

Certain parts of the MMI were specified with MSCs although the MMI was not supposed to be a part of the FCM. The objective was to obtain a description of the user's interaction with the FCM through the MMI.

3.2 Validation

The IRS and the FRS were of course compared throughout the requirements stage. Moreover, the diagrams of the FRS were presented for the customers who could then identify misunderstandings and mistakes. The requirements stage was completed by a formal walk-through. The formal walk-through lasted for about four hours. The walk-through was attended by the developers, the customers and the FDT expert. It was organised as follows: all the diagrams in the FRS were presented by the developers aided by an overhead projector; the correctness of each diagram was discussed and a protocol identifying the required modifications to each diagram was written. The FRS was formally accepted by the customers first after the diagrams had been corrected in accordance with this protocol.

3.3 Discussion

The requirements stage of the FCM development was a positive experience. The formalization process obviously forced the developers to have a critical look at the IRS. Thereby they identified a large number of features and aspects that were not clear, that were missing or that were wrong. The fact that the IRS and the FRS to some extent were written in parallel allowed the developers to give feedback to the customers with respect to the completion of the IRS.

The developers found the MSCs easy to understand, and moreover, that they capture the message exchanges in an elegant manner. As in the case of the class diagrams and the use-case diagrams they were also well-understood by the customers. To select the right level of abstraction was a major challenge. With MSC96 the new concept high-level MSC (HMSC) was introduced. HMSC extends MSC with constructs that should give a better overview over MSC documents. The experience made in the FCM development was that the developers found HMSCs quite useful. HMSCs made it possible to formalize properties that could otherwise not have been formalized. Additionally, an overall description of the whole system starting with one top-level diagram could be given. On the other hand, the customers had more problems understanding HMSCs than MSCs.

4. Design (Stage 2)

The FDS consists of SDL diagrams - to some degree augmented with C-code. Some MSCs of the FRS were translated into new MSCs reflecting the particular features of the design. This was necessary to allow automatic verification of the SDL design.

4.1 Tools Used to Support the Design

During the design stage the FCM developers made use of three SDT facilities, namely an SDL editor, tools for graphical simulation of SDL design, and facilities for model-checking. As explained in more detail below, by model-checking we mean validation of SDL specifications based on exhaustive exploration.

4.2 Relationship to the Formal Requirements Specification

When transforming one system description into another system description written at another level of abstraction, as we normally do when we move from requirements to design, it is necessary to make a decision with respect to the relationship between these two specifications. In the FCM development we agreed that the FDS should mirror the external communication behaviour captured by the FRS. On the other hand, we decided to interpret the class diagrams of the FRS more as an aid to understand the meaning of the requirements than as a requirement that should be mirrored by the class structure of the FDS.

4.3 Overall Design of the FCM Kernel

As already mentioned in Section 2, the FCM Kernel specifies the main functionality of the FCM - but at a rather abstract level of interaction. It provides interfaces to the Operator Simulator, the Automatics Simulator, the MMI, the File Operations Component and to the Signal Wrapper. The Signal Wrapper translates the abstract signals of the FCM Kernel to the concrete signals of the Process Simulator. The main task of the FCM Kernel is to distribute incoming signals to the components according to user defined scenarios and to log the runs of scenarios.

4.4 Discussion

Only the external communication of the FCM Kernel was specified with MSCs. Hence, the communication inside the system, between the blocks and the processes within the blocks, was not captured by MSCs. There were several reasons for this:

- Before the design was initiated, the need to specify the interaction between and within the blocks was partially not seen as important, and was partially also seen as extra work.
- There was uncertainty about the exact functionality of the blocks. Throughout the design stage the FCM developers were also unsure whether the block design was good enough and whether it would change. Changes in the block structure would have influenced the MSCs.

On the other hand, there were also several good arguments for using MSCs for this purpose:

- An MSC specification makes it easier to understand the block behaviour.
- An MSC specification can be used as a requirement specification for other designers.
- The block design can be verified against the MSC specification.
- The designers have to think more about what the block is supposed to do before they start implementing the block.

After having completed the FCM design the developers concluded that for some blocks it really would have been useful to start the development of the block by describing it with MSCs.

The FCM developers found SDL relatively easy to learn. Only a small number of graphical symbols was required to describe the static structure of systems. As soon as the FCM developers gained an overview of those they were themselves able to understand and structure non-trivial SDL specifications. When it came to the dynamic aspects, the FCM developers found SDL more challenging - in particular, they were often surprised by the overall effect when processes were composed in parallel. Although SDL describes systems in an exact and unambiguous manner, the FCM developers experienced the need for informal comments and descriptions outlining the main design ideas - ideas that in their opinion cannot easily be extracted from a pure SDL description.

In SDL a state machine of a process type is described in a diagram for the process type and in diagrams of the supertypes. The fact that the description of the state machine is placed in many diagrams makes it difficult to understand the complete state machine; both for the designers themselves and for reviewers. In fact, in the FCM development the FCM developers recognized that their understanding of the process decreased when it was described in several diagrams. What the FCM developers had appreciated is an approach where the complete state machine of a process is shown in one diagram. To make clear which states and transitions are defined in super types, these should appear differently; for example with different kinds of lines. As such complete state diagrams in certain cases can become very large, we expect from tool providers that they allow the user to define whether he wants to see the whole state diagram or only the parts that are specific for the subprocess. We also expect easy navigation from symbols in a process diagram to the corresponding symbols in the diagram of the supertypes.

The FCM Kernel contains a large number of channels - both between blocks inside the FCM Kernel and between the FCM Kernel and its environment. The reason lies in the IRS that prescribes five channels to the Process Simulator and two channels to the Operator and Automatics Simulator. In retrospect, at the abstraction level of the FCM Kernel, such a large number of channels is not really necessary. One channel for each component would have been enough and would have made the design easier and more understandable. Sometimes blocks were connected by two channels of the same direction - for example, when two different processes in one block communicate with two different processes in another block. On the block level the two communication links could have been combined and would have made the design more readable. Due to the generous use of channels some blocks in the FCM Kernel ended up with 15 gates. Afterwards the FCM developers agreed that the amount of gates made the block appear more complex than it actually was.

Often MSCs from the FRS were needed for validation purposes. They then first had to be refined to take into account more design related details. For example, the instance line representing the FCM had to be replaced by several instance lines for the sub-components of the FCM Kernel - or, alternatively, signals in an abstract MSC were refined by sequence of signals in a more concrete MSC. From SDT the developers expected some mechanisms supporting navigation between diagrams related in this manner. The FCM developers were not able to find such a facility in SDT.

Simulating the SDL design under construction proved very useful. The possibility to follow the executions graphically was an important help to understand the system behaviour, and to find faults. Furthermore, it gave the developers early feedback. The FCM developers liked the way simulations could be controlled through the simulation user interface of SDT. From a list showing all possible signals, a signal could be selected and sent to a process. If the process in question was not ready to receive the signal a message appeared in the simulation log, and the signal was also marked in the generated MSC. Most of the faults made during the design of the FCM were connected to situations where a signal was sent to a process that could not receive it. Thanks to the indications on those signals that could not be received in the generated MSCs, these faults could be identified and corrected quickly. In the design of the FCM Kernel simulations were not used from the beginning because the FCM developers felt that their design first had to reach a certain state of maturity. Thereafter, however, simulation became an integral part of the design process. In retrospect the FCM developers think they waited too long before they started simulating. In fact, they believe that simulation activities should be integrated in the design process from the very beginning. Especially during the early stages, when many things are unclear or misunderstood, it is important to simulate the various design sketches.

The FCM developers recommend simulation for personnel that is inexperienced with SDL. The FCM developers believe simulations helps inexperienced personnel to understand the main characteristics of SDL and the advantages of using SDL in system development. The FCM developers report that the simulations gave them more confidence in their design concept; they also believe the facilities for simulation resulted in a faster development and improved quality of the resulting system.

An SDT facility often used by the FCM developers was the recording of scenario runs in text files. The text files could be used for re-runs of the scenario or to easily create new scenarios in a text editor. Most of the simulation scenarios for the FCM Kernel were directly created in text files. Editing the text files is certainly the fastest way to generate simulation scenarios but requires that the engineer learns the scripting language needed for this purpose. The language is not very difficult to learn and is well-documented in the SDT documentation. The MSC generated during a simulation can become very large and drawing them requires a lot of resources. To reduce this problem SDT allows the user to configure the MSC drawing under simulation.

The use of model-checking techniques (manual navigation not included) was not very successful in the FCM development. The FCM developers believe that some of the problems could have been avoided if model-checking aspects had been taken into consideration in the FCM design. A design that includes assumptions about the environment is for example easier to validate than one that does not include such assumptions. Note that the FCM developers report that there were configuration facilities for model-checking in SDT that they did not investigate in detail. However, we do not think this was decisive. We believe there were two main reasons why the FCM developers not really managed to exploit the SDT model-checking facilities. Firstly, the FCM developers did not have the required competence, training and experience to really exploit the technology offered. Secondly, the technology offered has, in our opinion, not really reached the level of maturity and user-friendliness to be exploited by ordinary engineers. We believe model-checking can function in real system developments already today, but then the validation must be carried out by specialised personnel.

5. Implementation (Stage 3)

SDL was used to specify two implementation oriented components. The implementation language was C. The FCM developers used the SDL editor and the simulator of SDT to write and validate SDL specifications. The SDT code-generator was used to automatically generate C-code from the SDL. The integration testing was not supported by any tool.

5.1 Relationship to the Design Specification

The FCM Kernel is designed at a relatively high level of abstraction. It abstracts from the concrete signal representation of the Process Simulator and other environment entities. Moreover, it does not specify the concrete file operations. To obtain a description of the FCM taking the concrete implementation dependent data representations into account, we basically had two options:

- To write a completely new FCM description based on the concrete data representation.
- To extend or augment the FCM Kernel with new components mapping the abstract design to the concrete data representation.

As already explained in Section 2, we decided in favour of the second alternative. For this purpose, two new SDL specifications were written, namely a specification of a Signal Wrapper mapping the abstract signal representation of the FCM Kernel to the concrete signal representation of the Process Simulator, and a specification of a File Operations Component describing the detailed file operations. This had several advantages:

- The SDL specification of the FCM Kernel could be reused.

- Since a major part of the FCM implementation, namely the FCM Kernel had already been validated the need for validation was reduced.
- We obtained a nice, layered description clearly separating the design issues from the implementation issues.
- The possibilities for maintenance and porting were improved since modifications to the implementation platform would normally only require the specifications of the Signal Wrapper and the Files Operations Component to be altered.

As already mentioned, the Signal Wrapper and the File Operations Component were specified in SDL. The experiences from this activity were similar to those for the FCM Kernel and will therefore not be further elaborated here.

5.2 Integration Technology

For the three SDL specifications code was generated automatically. In the following we refer to the automatically generated code for an SDL specification as an SDT application. Additional C code had to be written manually to connect the three SDT applications with each other and with the Process Simulator.

Two technologies to connect the SDL applications were considered, namely communication through the SDT integration mechanism and communication through UNIX sockets. The latter alternative was chosen. The following reasons were decisive for the FCM developers:

- Independence from the SDT software.
- Integration in a heterogeneous environment.
- The SDT documentation describes in detail how two SDT applications, or an SDT application and another application can be combined using sockets.

Concerning the second point, it was already decided that the Operator Simulator would be implemented in the programming language Java which has very good support for sockets.

5.3 Discussion

The development of three separate SDL specifications (SDL systems) instead of just one made simulation and testing more complicated. How two SDL systems work together could not be simulated graphically. To test whether two SDL systems interact correctly, an environment interface for both systems had to be built. When an integration problem occurred the cause could either be in one of the systems or in their environment interfaces.

The integration based on UNIX sockets was time consuming. One reason may have been the inexperience of the FCM developers. They needed some time getting used to converting the types correctly. Moreover, for each signal some handwork in C programming was required, and the testing was tedious.

6. Empirical Results and their Interpretation

One objective of the system development on which this paper reports was to obtain a better understanding of the practical effects of an increased level of formalization. The FCM development was helpful in this respect. For example, as explained in Section 3, the formalization of the informal requirements identified weaknesses, mistakes and inconsistencies. This indicates that an increased level of formalization in the early stages of developments might result in earlier identification and correction of faults. Moreover, as explained in Section 4, the FCM developers felt that the SDL formalization in combination with the SDT facilities for simulation allowed them to try out and test design ideas more systematically than in conventional developments. One might expect this to result in improved system quality. The FCM developers also gained the impression that FCM-like developments could result in improved achievement (given, of course, that the developers are experienced in the methods, specification techniques and tools used). Hence, after having been through the FCM development, the FCM developers were more confident with respect to the potential of formal specification techniques. This is, however, not the same as scientific evidence. In this section we discuss to what degree the data from the FCM development underpins the impressions of the FCM developers.

6.1 Effort

The FCM development was started up in September 1997 and completed in July 1998. Table 1 summarises the invested effort.

	Training	Stage 1	Stage 2	Stage 3	All 3 stages
Customer A	0	12.1	1.3	0.6	14.0
Customer B	0	12.8	0.6	0	13.4
Both Customers	0	24.9	1.9	0.6	27.4
Developer A	54.5	22.6	40.4	19.8	82.8
Developer B	7.3	15.6	22.1	35.4	73.1
Both Developers	61.8	38.2	62.5	55.2	155.9
All Four	61.8	63.1	64.4	55.8	183.3

Table 1Effort in Man Days

For each customer and each developer the table presents the man days (8 hours per day) invested in training and the three main stages of the FCM development. The effort of the FDT expert is not recorded which is reasonable since he was not really involved in the FCM development.The column for training contains the number of man days invested in learning and working with formal techniques and CASE-tools before the FCM development was initiated. Hence, the customers did not receive any training at all. Developer A was trained in several respects. First, he received some basic training in the actual specification techniques. Then he was involved in the evaluation of the CASE-tool Rhapsody. The person intended to play the role of developer B received similar training (he was involved in the evaluation of the CASE-tool Rational Rose). This person was, however, replaced by another software engineer shortly before the initiation of the FCM development. The training of developer B consisted in that he specified a simple production cell using the selected specification techniques supervised by the FDT expert.

6.2 Lines of Code

In order to measure achievement we need an estimate of what has been produced. A standard technique is to measure the size of the resulting code. In a system development based on automatic code-generation this is problematic. Automatic code-generation will typically result in much more code than if the same system had been implemented manually. Instead, we therefore estimated the size of the specification which was the input to the code-generator, and added this to the size of the manual code.

The FCM development resulted in three graphical SDL specifications - for the FCM Kernel, the File Operations Component and the Signal Wrapper, respectively, plus some manual C-code needed to connect the automatically generated code for the three SDL specifications. Within industry lines of code (LOC) is a standard, but controversial, measure of what has been produced. Table 2 presents the lines of SDL-PR, the lines of automatically generated C-code, the fraction between generated C-code and SDL-PR, the lines of cleaned C-code, and the fraction between cleaned C-code and SDL-PR.

	SDL-PR	C	C/SDL-PR	C clean	C clean/SDL-PR
FCM Kernel	5 590	42 110	7.5	30 813	5.5
File Operations	419	6 352	15.2	5 062	12.1
Signal Wrapper	1 197	6 982	5.8	5 074	4.2
Total	7 206	55 444	7.7	40 949	5.7

Table 2 Lines of Code

The SDL-PR was generated from the graphical SDL-GR by SDT. In our SDL-PR estimates comments and blank lines have been removed together with the utilities package and the "ctype" file (supplied as C-code by SDT). The definition of cleaned C-code is that everything between /* and */ has been removed.

The FCM developers found it difficult to estimate the size of the manual C code since the manual coding to a large extent consisted in updating libraries that came with the SDT installation. However, the FCM developers concluded that the size of the manual code would be close to 1000 LOC. Based on this we estimate the size of the FCM to 8200 LOC before code-generation (7200 LOC SDL-PR + 1000 LOC C).

There are of course alternatives to LOC. For example, complexity metrics based on syntactic elements [Hal77] or the control flowgraph of the program [McC76]. In spite of their wide use, it has not been demonstrated that these metrics are better than lines counts in predicting effort, reliability or maintainability. Albrecht's function points [Alb79], based on the complexity of functionality usually computed from a detailed system specification, have been criticised for being complex, difficult to compute and too subjective. See [AG83], [Das92], [FWI95] for detailed discussions of code-size estimates.

6.3 Achievement

Table 3 presents the man days/KLOC ratio for the FCM development. KLOC stands for kilo lines of code (i.e. 1000 LOC). This ratio is a standard metric of achievement in industry. Because of uncertainty connected to LOC estimates, it is, however, not seen as very reliable.

LOC	Man Days	Man Days/KLOC
8200	183	22.5

Table 3 Achievement Captured by Man Days/KLOC

A large percentage of the man days invested in the FCM development (perhaps as much as 50%) was required to learn the technology in question. There are of course many publications giving hard numbers from real projects on effort and KLOC [WB85], [Gra85], [BBB+85], [BDW96]. However, due to the uncertainty related to the estimates from the FCM development they will not be discussed in detail here.

[ZW98] distinguishes between three kinds of methods for empirical experimentation:

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

The FCM development was based on an observational method. The use of a controlled method is of course what we would have preferred. However, since the personnel involved in the FCM development was forced to learn a completely new technology, it was not practical to also impose the rigorous routines required by a controlled method.

6.4 Faults

It is often argued that the use of formal techniques leads to a reduction in the number of faults and that faults are discovered more early. In an attempt to validate this kind of claims each fault found during the FCM development was carefully described and logged. Inspired by [BDW96], we recorded:

- the number of faults introduced in the various documents (specifications and implementations) produced during the FCM development;
- the number of faults discovered during each development stage.

This required, of course, a clear definition of what a fault is. To provide such a definition is not as easy as one might expect. Anyone with at least some programming experience knows that programming involves a lot of corrections, modifications and experimentation based on trials and failures. Clearly, what is needed is a definition of fault that does not force the system 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. We decided to follow [BDW96] and defined a fault as follows:

> A fault is found when a change is required to a decision specified at an earlier development stage. A decision made and corrected within the same stage is not considered as a fault.

Since a change to a decision made and corrected within the same development stage does not count as a fault, the creative experimentation mentioned above is not constrained. The definition of fault obviously requires that the different development stages and their various activities are carried out in some kind of sequence.

The SQA-Manual [SQA1.0] already had a form for fault reporting, namely QA-F-107. To support the fault reporting described above, QA-F-107 was modified to provide four new fields facilitating respectively:

- reporting on the stage in which the fault was made;
- reporting on the stage in which the fault was discovered;
- reporting on document identification and issue number for the document in which the fault was found;
- classification of the reported fault.

Altogether 22 faults were registered during the FCM development. Details with respect to when they were made and discovered are summarized by Figure 2.

‘

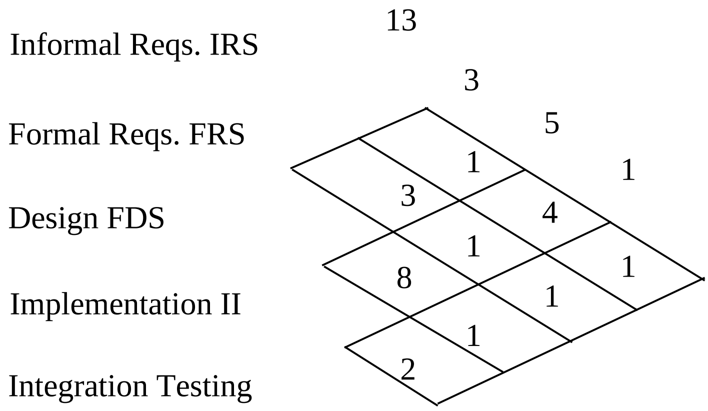

Figure 2Fault Grid

The grid records the number of faults found during the various stages with respect to the document in which the faults were introduced. It should be understood as follows. With respect to where the fault was introduced we distinguish between the IRS, the FRS, the FDS and the II. With respect to where the fault was discovered we distinguish between the design stage, the implementation stage (including the specification and validation of the Signal Wrapper and the File Operations Component), and the integration testing. Since the FAME Tool has not yet been taken in use we have no fault reports connected to the actual use of the FCM.

The four diagonals from top-left to bottom-right show the total number of faults introduced in the IRS, FRS, FDS and II, respectively. The three diagonals from top-right to bottom-left show the total number of faults found during the design, implementation and integration testing. Hence, for example, of the 13 faults found during the implementation, 8 were introduced in the FDS, 1 was introduced in the FRS and 4 were already present in the IRS. Moreover, of the 6 faults introduced in the IRS, 1 was found during design, 4 during the implementation and 1 during integration testing.

Note that there is no number recording the faults found during the formalization of the IRS. The reason is quite simply that the IRS and FRS to a large extent were written in parallel. Hence, when we distinguish between the formal and the informal requirements we do not really distinguish them as activities, but as two different documents developed in parallel giving two different views of the system.

The numbers above the grid show the number of temporal activities required to find a fault. For example, 13 faults were found after one activity, 3 after two etc. Hence, more than 50% of the faults were found during the activity immediately after the activity in which they were introduced. This corresponds very well with the result from [BDW96].

Of course the FCM development is just a single experiment and the number of faults found is low. It is not possible to draw significant conclusions. Fault reporting connected to the FCM development highlighted a number of problems related to fault counting; in the following we discuss these problems in more detail.

The FCM development was based on a waterfall process. Initially, we intended to complete each stage before the next was started up. 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. The overlap of these two stages occurred because the development of clearly separated subcomponents progressed at different speeds. Hence, this did not really cause problems for the fault reporting. If we had used a more iterative development process with rapid cycles as recommended by recent development methods like the Unified Process, and which the FCM developers clearly saw the need for, a more sophisticated definition of fault would have been required.

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 formalisms that defines the notion of satisfaction in a, for our purposes, fully satisfactory manner.

7. Conclusions

The FCM development was carried out within the research activity FAME. As explained in Section 2, FAME attempts to develop a tool for man/machine task-allocation experiments - the so-called FAME Tool. The researchers in charge of FAME were the customers in the FCM development. The FCM development was certainly of a non-trivial size (more than 8000 lines of code before code-generation). Both the requirements and the design specifications were formalized. The requirements were formalized in UML use-case diagrams, UML class diagrams and MSCs. The design specification was formalized in SDL. In addition, we also wrote several SDL specifications during the implementation stage characterising the mapping of the design to the concrete data and signal representation of the implementation. Hence, the FCM development was of industrial size and made heavy use of formal techniques. The system development was also real-life in the sense that it was organized as a typical system development at IFE. The FCM development proved that the selected specification techniques and CASE-tool are useful and practical for the development of distributed systems of nontrivial size.

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. The FCM development employed what [ZW98] calls an observational method. The use of a controlled method would have been scientifically more satisfying. However, since the personnel involved in the development was forced to learn a completely new technology, it was not practical to also impose the rigorous routines required by a controlled method.

The ease with which formal techniques can be integrated in a conventional software process is to some degree dependent on the attitude of the management of the development project. If the management is negative to the whole exercise this attitude will easily also contaminate the minds of the software engineers in which case problems are inevitable. We believe that the positive attitude of the management of the FCM development (the research scientists in charge of FAME) was very important for its successful completion.

It is generally accepted that adequate support by tools of commercial quality is important for successful use of formal techniques in an industrial context. The FCM-development was supported by the CASE-tool SDT all the way from the formalization of the requirements to the automatic generation of C-code. In particular, we used the SDT editors for class-diagrams, MSCs and SDL. We made in addition heavy use of the SDT facilities for simulation and the code-generator for C. These components were all helpful. In particular, the facilities for simulation, where the execution of the SDL system can be followed in the SDL diagrams and in dynamically generated MSCs, proved very useful. For the FCM developers, who had almost no training in SDL before the FCM development was started up, the facilities for simulation were also important for understanding the exact meaning of the various SDL constructs. Hence, SDT is certainly a useful tool. However, SDT is also a very sophisticated tool and much training is required in order to fully master it. The developers also tried to exploit the SDT facilities for model checking. This did not work out well. One reason could be that the developers did not have the required understanding and training in this kind of advanced technology. Validating a system using model checkers is a challenging task requiring experienced personnel that is well-educated in the actual technology and knows how to exploit the specialized heuristics of the CASE tool.

The FCM developers were ordinary software engineers. Both had recently completed their education. Neither had received training in formal techniques during their education, and neither had earlier participated in system developments involving formal techniques or CASE-tools. Hence, the FCM development was for them a big challenge. In the months before the FCM development was initiated, one of the developers was involved in the evaluation of CASE-tools and acquired a basic understanding that way. The other person intended to take part in the FCM development received similar training, but had to be replaced just a few weeks before the FCM development was started up. This meant that the second FCM developer was almost without training when the FCM project was initiated. Hence, a large percentage of the man hours invested in the FCM development (perhaps as much as 50%) was required to learn the technology in question. To really understand the essentials of the software process, a developer must have gone through the whole development cycle. It is therefore important that inexperienced developers can start with smaller projects. Preferably they should be assisted by an experienced software engineer with competence in the specification techniques, tools and development process in question (such a software engineer was not available in the FCM development). If such a person is not available in-house, it seems worthwhile to first send the developers to a course. However, to really profit from a course they should at least work through some of the tutorials on the technology in question before attending the course.

During the FCM development use-case diagrams, class diagrams and the simple MSC92 diagrams proved very helpful as a medium for communication between the customers and the developers and between the two developers. In particular, a formal walk-through attended by both the developers and the customers, where all the diagrams in the formal requirements specification were discussed one by one was very successful. The walk-through lasted for almost five hours and many problems and weaknesses identified in the two requirements specifications were protocolled together with a description of how they should be mended. Although the customers had no background from using formal techniques, they were able to participate in detailed discussions on their design.

The FCM development was based on use-case diagrams, class diagrams, MSCs and SDL diagrams. These specification techniques are all graphical, and this is an important reason for their popularity. The personnel involved in the FCM development found the facilities for graphical presentation important. It was important during the walk-through mentioned above because the diagrams were so simple that they could immediately be understood by the customers. It was also important during the simulation of the SDL design. The possibilities for following the execution step-by-step in graphical SDL and MSC diagrams simplified debugging.

Acknowledgements

Andreas Bye, Tor Steinar Brendeford and Håkon Sandmark participated in the FCM development. We are grateful for their positive and open-minded attitude.

References

[Alb79] A.J. Albrecht. Measuring application development productivity. In Proc. IBM applications development joint SHARE/GUIDE symposium, pages 83-92, 1979.

[AG83] A. J. Albrecht, J.E. Gaffney. Software function, sources lines of code, and development effort prediction: a software science validation. IEEE Transactions on Software Engineering, 9:639-648, 1983.

[BBB+85] M. Barnes, P.G. Bishop, B. Bjarland, G. Dahl, D.Esp, P. Humphreys, J. Lahti, S. Yoshimura, A. Ball, O. Hatlevold. PODS (the Project On Diverse Software). HWR-323, OECD Halden Reactor Project, 1985.

[BBHHM98] A. Bye, T. S. Brendeford, E. Hollnagel, M. Hoffmann, P. Mohn. Human-machine function allocation by functional modelling - FAME - a framework for systems design. HWR-513, OECD Halden Reactor Project, 1998.

[BDW96] J. Bicarregui, J. Dick, E. Woods. Quantitative analysis of an application of formal methods. In Proc. FME96, LNCS 1051, pages 60-73, 1996.

[Das92] M.K. Daskalontas. A practical view of software measurement and implementation experiences within Motorola. IEEE Transactions on Software Engineering, 18:998-1010, 1992.

[FWI95] N. Fenton, R. Whitty, Y. Iizuka, eds. Software quality assurance and measurement, a worldwide perspective. Thomson Computer Press, 1995.

[Gra85] R.B. Grady. Practical software metrics for project management and process improvement. Hewlett-Packard Professional Books, Prentice-Hall, 1992.

[Hal77] M. Halstead. Elements of software science, North-Holland, 1977.

[ISO9000-3] Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software. International Organisation for Standardisation, 1991.

[McC76] T. McCabe. A software complexity measure. IEEE Transactions on Software Engineering 2:308-320, 1976.

[Stø98] K. Stølen. Formal specification of open distributed systems - overview and evaluation of existing methods. HWR-523, OECD Halden Reactor Project, 1998.

[SKMS98] K. Stølen, T.W. Karlsen, P. Mohn, H. Sandmark. Using CASE-tools based on formal methods in real-life system development of distributed systems. HWR-522, OECD Halden Reactor Project, 1999.

[SMST99] K. Stølen, P. Mohn, H. Sandmark, H. Thunem. Experiences from the development of the FAME communication manager using the CASE-tool SDT

[SDT3.2] Telelogic SDT 3.2 Methodology Guidelines. Part 1: The SOMT Method. Telelogic AB, 1997.

[SQA1.0] Software Quality Assurance Manual, Version 1.0 - 1/6/95. IFE, 1995.

[WB85] D.M. Weiss, V.R. Basili. Evaluating software development by analysis of changes: some data from the Software Engineering Laboratory. IEEE Transactions on Software Engineering, 11: 157-168, 1985.

[ZW98] M. V. Zelkowitz, D. R. Wallace. Experimental models for validating technology. IEEE Computer 31: 23-31, 1998.