Applying SOFL to Develop a Univeristy Information System*

Shaoying Liu†, Masaomi Shibata‡, Ryuichi Sato*
†Faculty of Information Sciences
Hiroshima City University, Japan
Email: shaoying@cs.hiroshima-cu.ac.jp
‡Network Solution Division
Haielekonkowa Ltd., Japan
*System Development Division
Active Brains & Trust Corporation, Japan

Abstract

How to effectively apply formal methods under schedule constraints to develop real systems is an important issue to address. In this paper we report our recent experience in the application of SOFL (Structured Object-based Formal Language) to developing a *University Information System*. The intention of this study is to investigate whether and how formal methods can benefit development of *non-safety critical systems* with time constraints. We have found that if used appropriately, formal methods can help to save time, to improve the accuracy of discussions and communications, to facilitate transformations from specifications to design, and then to programs. On the other hand, we have also found that formal specifications in the style of pre and postconditions can be difficult to write; formal specifications without reviews may involve mistakes; and satisfactory formal specifications that serve as the reliable documents for development and maintenance may be completed only throughout the entire development process.

1 Introduction

As real systems developed in software industry are usually required to deliver within a strict deadline due to severe competitions in the market, an interesting question is how application of formal methods can improve, if possible, the quality of the developed software within the planned schedule. As we all know, this question is not easy to answer or answer accurately, because in addition to the schedule constraint, budget, expertise, and management issues also contribute to the result of software projects.

In this paper we do not attempt to offer an effective answer to this question either, but focus on reporting our recent experience in applying the formal method SOFL (Structured Object-based Formal Language) to developing a *University Information System*. SOFL has been recently developed to promote the use of formal methods in an engineering manner [1, 2, 3, 4, 5]. It is both a formal specification language and a rigorous method for software development. As one of the major investigations in our project FMSE[1] , we attempt to gain an insight into the impact of formal methods, in particular SOFL, on the process of software development within the fixed schedule through the work presented in this paper.

We took the strict life-cycle process to develop the university information system using SOFL within eleven months. We started with proposals and analysis of the user requirements, and then proceeded to their formal specification. The informal requirements specification includes around 66 items of functional requirements, and the formal specification contains about 1192 lines of expressions. Design was carried out based on the formal requirements specification. The design specification contains around 42 condition data flow diagrams (CDFDs), graphical part of the specification, and about 2,610 lines of formal definitions.

Although SOFL implementation language can be used for the implementation of the system, we did not use it due to the lack of a complier and time for implementation. Instead, we used C++ to implement the system based on the design specification. The program that

*This work is supported in part by the Ministry of Education of Japan under Grant-in-Aid for Scientific Research on Priority Areas (A) (No.10139236),Grant-in-Aid for Scientific Research (B) (No. 11694173), and Grant-in-Aid for Scientific Research (C) (No. 11680368)

[1] FMSE stands for Formal Methods for Software Evolution. It is part of the priority areas research project *Software Evolution* in Japan funded by the Ministry of Education of Japan.

contains about 2,684 lines of C++ statements is finally tested against the design specification. The detailed specification, design, and implementation of the university information system are described in our latest technical report [6].

Through this work we have found that if used appropriately, formal methods can help to save time, to improve the quality of discussions and communications, to facilitate transformations from specifications to design, and then to programs. On the other hand, we have also found that formal specifications in the style of pre and postconditions can be difficult to write; formal specifications without reviews may involve mistakes; and satisfactory formal specifications that serve as the reliable documents for development and maintenance may be completed only throughout the entire development process. In other words, achieving a complete, reliable formal requirements and design specification before implementation seems to be extremely difficult (if not impossible).

The reminder of this paper is organized as follows. Section 2 describes the background of this application. Section 3 elaborates how we took the life-cycle process to develop the system from informal requirements specification through design, implementation to testing. Section 4 discusses our experience and lessons to learn. Finally, in section 5 we give conclusions and outline future research.

2 Background of the Application

Hiroshima City University is a new institution consisting of three faculties: *International studies*, *Information Sciences*, and *Arts*. Each faculty has several departments. As the university is funded by Hiroshima City, it is expected to serve for the local society, industry and people by teaching and research.

In order to provide timely updated information, it is desirable to develop a University Information System to offer the online information electronically. By using this system one can find comprehensive information about the lectures and the research projects being conducted at the university at present time.

This project involves the three authors, who played different roles. The first author served as the end user of the system to propose overall requirements; as a project manager to offer instructions on the development of the system; and as a developer to review the requirements and design specifications. The second and third authors wrote informal requirements and formal specifications; carried out design; and implemented the system using C++ on IBM PC. As this project is basically a fourth grade students' research project for a B.Sc degree, it has to be terminated within eleven months required by the graduation schedule. Our initial target was to guarantee the success of this project in the sense of implementing a satisfactory prototype. In order to ensure the quality and success of this project, we decided to adopt SOFL for the development. In addition, we also thought that by using SOFL we could obtain a good documentation that can serve as a solid foundation for future evolution of this system.

3 The Development Process

We took the life-cycle process to develop this system. We started with the *informal requirements analysis*, and went through the phases of *formal specification*, *design*, *implementation* and *testing*. The document produced in each phase was reviewed before proceeding to the subsequent development phases.

In this section we focus on the description of how we performed the activities in each phase, while in next section we summarize our findings and experiences gained from this project.

3.1 Informal Requirements Analysis

On the basis of studying the two documents *Collection of Course Abstracts* and *Introduction of Researchers* available at Hiroshima City University, we proposed the overall requirements for the information system. Basically, the system is expected to offer two major functionalities. One is to supply information on lectures offered by the university and their related activities, such as examinations and additional lectures. The other is to provide information on research activities of all faculty members.

However, detailed requirements were impossible to obtain directly from the end user, because the end user would be the general public, including students, local residents, and visitors. For this reason, the first author served as the end user to take charge of requirements. After studying the initial requirements, the developers (namely the second and third authors) refined them into more detailed items of requirements which were divided into two parts; one is about teaching and the other is about research. The requirements specification was written in Japanese, and constructed in a top-down manner where some *big* requirement were divided into *smaller* items of requirements, as illustrated in table 1 (we have translated them into English for this paper).

For example, to retrieve and display information about lectures on inquiry involves the function to obtain the

Table 1: Part of the informal requirements specification

1 Retrieve and display information about lectures on inquiry.
1.1 Obtain the following pieces of information on inquiry with the input *lecture title*: *lecture title*, *lecturer*, *class-object* (department, grade), *day and time*, *classroom*, and *students* who are studying this lecture.
1.2 Obtain information on cancellation of lectures.
1.2.1 Given a lecture title, if this lecture is canceled, then display *lecture title*, *classes*, *cancellation date and time*, and *classroom*.
1.2.2 ...
...
2 Retrieve and display information about research on inquiry.
2.1 Obtain the following pieces of information on inquiry with the input *faculty member name*: *research areas*, *publications*, *projects* and *their members*, *sponsors*.
2.2 ...
...

related lecture information and/or the function to obtain the information on cancellation of lectures, and so on.

In order to produce this final informal specification, we held four meetings for validation (discussion, clarification, and error-checking). The latest version of the specification at any time was always modified by the two developers after a meeting and reviewed by the first author before next meeting.

3.2 Formal Specification

Although we achieved a consensus over the final informal specification, there was no guarantee that we understood all the terms (e.g., lecturer, date and time, classroom) and all the functional requirements (e.g., Obtain information on cancellation of lectures) accurately and consistently. In particular, we did not pay attention to the issue of how each term should be defined, and what inputs and outputs are expected when each of the functional requirements is implemented.

We formalized this informal specification using the SOFL specification language. The way of the formalization was carefully chosen based on the experience and lessons learned from previous projects [2, 4, 11]. We divided the entire informal specification into nine parts, each part containing requirements for the same *big* function (e.g., lectures, examinations, overall-introduction, and research). We then constructed nine *specification modules* to define the requirements, each module having its own state and being associated with one big function. As a whole, we did not take the top-down approach to construct the entire formal specification by decomposition in condition data flow diagrams, because we had obtained a farely clear hierarchy of informal specification whose precision consists in that of the bottom level requirements. We only formalized the bottom level requirements (e.g., the requirement 1.1) as condition processes in each *specification module*, because we believed that integrating them into a hierarchy of condition data flow diagram would involve design decisions and should be a job for design. The entire formal specification is documented in our latest technical report [6].

It is worth mentioning that the formal specification was constructed by following the outline of the informal specification, but modifications were done to extend or delete some functions as necessary. Since we had to keep the planned schedule, the functional specifications of many condition processes remain at an abstract level and may not reflect the requirements sufficiently. However, this does not necessarily mean that we did not understand the requirements. In fact, through discussions on the written specification, we understood the requirements to a very sufficient extent, although some of them were not formalized in the specification. For those requirements which we did not understand well and had no time to write their formal specifications in detail, we left them to be completed in the phase of design. However, to minimize the cost and risk of the project, we did ensure that all the *critical* and *important* functional requirements be formalized in the specification before design.

3.3 Design

We followed the SOFL methodology, as described in [2], to carry out an object-based design. A condition process in the specification is transformed into a condition process in the design. All the related condition processes defined in the same module in the specification

are integrated into a condition data flow diagram. Such a diagram is treated as a definition of a higher level condition process. As some of the condition processes involved in the diagram had a complex functionality, they were decomposed into condition data flow diagrams at lower levels. During the design, the condition processes defined in the specification and used in the design may have been modified to accommodate the new requirements. All the newly introduced condition processes (e.g., those occurring in the decomposition of a condition process specified in the specification) are also specified in the corresponding specification module in the design (note that we use SOFL for both requirements specification and design). We call this approach *middle-top-down* approach.

When designing condition data flow diagrams, we may declare *classes* in the associated specification modules to deal with specific tasks if necessary, For example, a global data store serving as a database that may be accessed or updated by condition processes at different levels of the hierarchical CDFDs is declared as a variable of a class, whereas a local data store that is only used within one CDFD is usually declared as a variable of a built-in data type (e.g., **seq**, **set**, **map**). Some classes are specilized to subclasses. The style of the entire design architecture is similar to that of a C++ program, except that condition processes are decomposed into condition data flow diagrams, rather than procedural programs. Condition processes are like *functions* in C++, which can communicate with other condition processes and/or call methods of objects concerned. As an example, we show the top level CDFD of the design for this system in Figure 1.

3.4 Implementation and Testing

We implemented the system in C++ under UNIX system by following the design specification. We carried out the implementation by transforming data definitions and operation specifications in the design. We transformed a composite data type to a C++ *class* declaration; a set, a sequence, a map etc. to a linked list or a file with appropriate element type, respectively. We transformed each condition process to a C++ *function* that may change states (i.e., external variables); transformed each function to a C++ *function* that does not change states; and transformed each class to a C++ *class*. As data stores are the state variables in the design specification and may be changed by condition processes, we transformed each data store to a class and provide necessary methods in the class. This transformation allowed the accesses and/or changes of the contents of data stores (objects in C++) through their methods. Some transformations may involve extensions in representation or functionality for implementation purpose.

In order to ensure that the system meet the user functional requirements, we carried out *unit testing*, *integration testing*, and *system testing*. The first two aim to detect faults concerning the implementation, whereas the system testing aims to identify faults that lead to the violation of the required functionalities. For unit and integration testing, we generated test cases based on the syntax and types of the input variables and the structure of the associated program fragments, whereas for system testing we generated test cases based on their corresponding formal specifications (if available in the design or requirements specification).

4 Experience and Lessons

4.1 Experience

Our experience shows that if used appropriately, formal methods, in particular SOFL, can help to save time, to assist communications and discussions, and to facilitate transformations.

4.1.1 Formal Methods can Help to Save Time

Although the overall time period of the project was decided as eleven months in advance, the specific time for each phase of the development was not fixed. On the basis of our experiences in several previous projects and our workload for other jobs at the university, we proposed a initial time table for the entire development process at the beginning of the project, as shown in Table 2. Since the second and third authors knew little about SOFL at the beginning, although they had studied a course on VDM and data flow diagrams before, the first author, the designer of SOFL, first gave five seminars on SOFL to them, each taking about three hours. Then we applied SOFL to model a small *library* system. Through this example we all gained more knowledge about the SOFL specification language.

However, as the result of this project, we found that by using SOFL we actually spent less time than we expected and shortened the planned schedule for implementing the system, as shown in Table 3.

The time spent for each development phase was roughly recorded based on the real situation of the project, which also includes the time spent for communications, modifications, and reviews.

It is our belief that this positive result was obtained primarily due to three reasons. The first is that we took *total football approach* to developing the system. "Total football" is a way to play English football (or soccer) where all the players move back and forth to function as both defense and forwarder. By applying

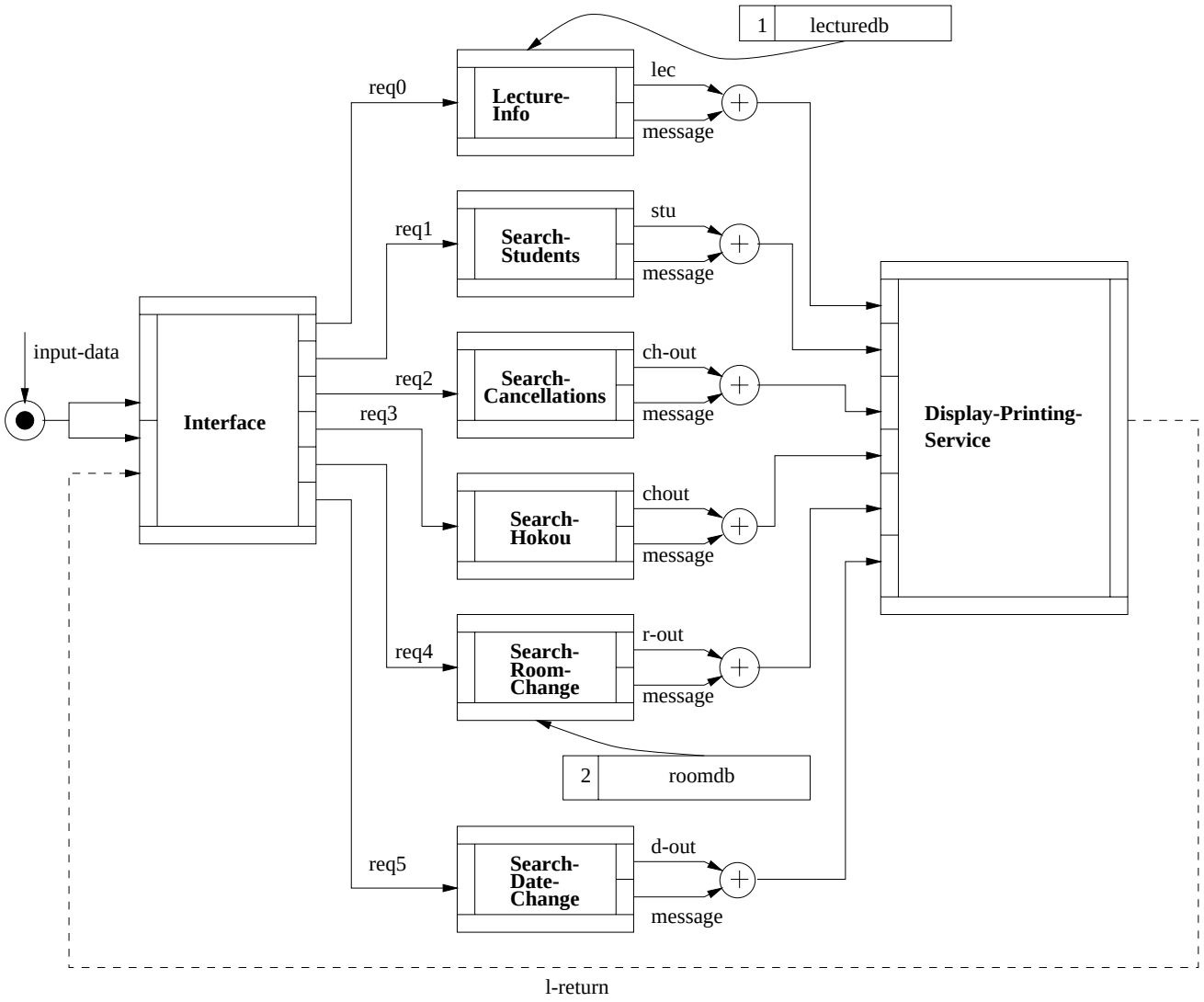

Figure 1: The top level CDFD of design

Table 2: The initially planned schedule

Phase	**Time (days)**
Training	30
Informal specification	31
Formal specification	61
Design	61
Implementation	90
Testing	30
Writing reports	30

Table 3: The real schedule

Phase	Time (days)
Training	30
Informal specification	15
Formal specification	30
Design	30
Implementation	80
Testing	20
Writing reports	70
Holiday	40

this approach to using formal methods for software development, we let the same developers write formal requirements specification, carry out design, write C++ code, and conduct some unit and integration testing. In particular, we emphasize that the formal specification and design (both abstract and detailed design) must be done by the same developers. This approach is very effective in practice because writing formal specifications is a learning process, through which the developers can understand deeply the problem and requirements. Also, they understand how the related data are modeled using what types. Thus, when carrying out design, they can quickly construct the architecture of the entire system, and reuse or revise the data types and operations specified in the requirements specification. If design were given to another developer who is not familiar with the specification, he or she would have to spent much more time to understand the formal specification before carrying out the design. This is not an easy job, especially when the specification is complex.

Another reason for saving time is due to the special feature of SOFL notation. We found that condition data flow diagrams (CDFDs) in SOFL provide us with very effective aids in three aspects. They allowed us to express our idea of requirements or design in a comprehensible manner; they serve as a guideline for formalizing their components (e.g., condition processes, data flows, data stores); and they also provide a good traceability for modifications of the specification and design, and for searching components during implementation.

The third reason is that we followed the principle of "understanding is enough" to construct the specification and design. We did not attempt to offer a complete formal definition for every condition process occurring in the specification or design. Rather we used the formal method as a tool to gain better understanding of the user requirements and problem. For this reason, we kept the formal description to a rather abstract level, and left the specification incomplete in the sense of recording all the user requirements in detail (e.g., some functions used in condition process specifications are undefined; both pre and postconditions of condition process may be **true**). However, we did not try to make any compromise if the functionality of a condition process is unclear to us. We also made much efforts formalizing the most important and critical requirements.

4.1.2 Formal Specifications can Help Communications and Discussions

Our experience shows that SOFL has helped us significantly in communications and discussions between the developers for two reasons. One is that SOFL employs the graphical notation called condition data flow diagrams to describe the high level architecture of the specification and VDM-SL-like formal notation to specify its components. When reading the specification, we can easily follow the guideline offered by the diagrams to understand how the system is expected to work at a high level, and can easily proceed to read the detailed formal definitions of the components in the corresponding specification modules.

Another reason is that type definitions and formal specifications of condition processes in specification modules offer a precise document for us to understand the precise ideas of the specifiers. This allowed us to accurately communicate and discuss over the contents of the specification.

4.1.3 Formal Specifications can Facilitate Transformations

Another advantage of using SOFL is that the transformations from the formal requirements specification to the design specification, and then to the program were rather easy due to several factors. One is that the precisely defined data types in the formal requirements specification lay the ground for further evolution in the design, and those in design for choosing concrete data structures in the program. Another factor is that condition processes specified in the requirements spec-

ification could be reused or refined in the design, and those in the design served as a basis for implementation in the program. Without those precisely defined data types and condition processes, it would be much more difficult to implement the program due to the lack of a reliable foundation.

Furthermore, we believe that this effectiveness is also due to the fact that the same developers carried out the transformations. As we mentioned previously, the same developers could take advantages of what they had learned in the previous phase during the transformations. Not only did these advantages allow the developers to save time, but they also helped to achieve the quality of the transformations.

4.2 Lessons

In addition to our experience described previously, we have also learned lessons about using formal methods.

4.2.1 Implicit Formal Specifications can be Difficult to Write

We found that writing an implicit formal specification for a condition process, particularly for a high level and complex condition process, can be difficult. There are two causes for this difficulty. One cause is that the developers were often confused by the difference between logical expressions in specifications and executable statements in programs. They had strong tendency to write specifications in procedural style, which may be influenced by their previous programming experience in C and C++.

Another important cause is the lack of the *abstraction* skills that are essential for writing implicit formal specifications. Obtaining and applying those skills seem to be difficult for the inexperienced developers. Due to this reason, we sometimes spent quite a long time for an attempt of producing accurate abstract expressions in the specification. Because of the time constraint, we actually left many of condition process specifications to an abstract level, some of which had only signatures but no pre and postconditions.

4.2.2 Formal Specifications Need Reviews

Many faults in the formal specifications were detected by reviews. Those faults can be classfied into three kinds. One kind is the inconsistency between specification modules (i.e., **s-module**) and their associated CDFDs. The second kind is syntactic mistakes. This was caused by the lack of experience of using SOFL. Another kind of faults is concerned with the accuracy and completeness of specifications. As the developers did not understand the problem domain and requirements accurately in the beginning, many of the formal expressions in the condition process specifications were not correct and satisfactory. After repeatedly reviews and discussions, the specifications were modified and improved.

4.2.3 Formal Specifications May not be Completed Before Implementation

Formal specifications are expected to serve two functions. One is to help the development of the system, and the other is to help maintenance of the system after it is delivered for operation.

We found that it was almost impossible to write a complete formal specification before implementation. By completeness we mean that the specification record all the required functions. In some cases, as the complete definition of a desired function must involve detailed expressions that may need to be represented in an algorithmic fashion, we had to leave them undefined, and expected to resolve this problem during coding time. Also, specifications were often modified and/or improved, we could not keep on reviewing all the changed versions. Thus, the gap between the original specifications and the final program was enlarged. As the result, the original specifications may not completely and accurately represent the functions of the final program. For the purpose of maintenance, the original specifications are almost useless.

5 Conclusions and Future Research

5.1 Conclusions

We have presented an application of formal method SOFL to developing a University Information System, and discussed our experience and lessons learned through this project.

Through this project we have found that by using SOFL we have shortened the the development time planned initially; stimulated discussions and communications between the developers; facilitated transformations from specifications to design and then to programs; and curbed faults in the program. On the other hand, we have also found that the formal specifications in the style of pre and postconditions of condition processes in s-modules can be difficult to write. The difficulty seems to come from the abstraction skills, which are required in writing formal specifications and are quite different from programming skills. Furthermore, formal specifications need to be reviewed, in order to

detect mistakes before proceeding to the next development phase. Formal specifications usually cannot be completed before implementation, for the sake of time and understanding, and need to be further improved after implementation, in order to serve as a reliable document for testing and maintenance.

5.2 Future Research

An interesting future work is to evolve this University Information System, in order to offer better and more comprehensive services. Another interesting issue to investigate is whether using formal methods can help to offer better quality of the same system within the same schedule than using informal methods (e.g., SSADM [14] and C++). As the nature of this study is complicated, it will remains as our long term research target.

6 Acknowledgements

We would like to express our gratitude to Professor Mitsuru Ohba, Yasuomi Sato, Akio Nakata, and graduate students Tetsuo Fukuzaki and Naoko Wakiya for their various helps during this project. We would also like to thank the Ministry of Education of Japan for their financial supports under Grant-in-Aid for Scientific Research on Priority Areas (A) (No.10139236), Grant-in-Aid for Scientific Research (B) (No. 11694173), and Grant-in-Aid for Scientific Research (C) (No. 11680368).

References

[1] Shaoying Liu and Yong Sun. Structured Methodology + Object-Oriented Methodology + Formal Methods: Methodology of SOFL. In *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems*, pages 137–144, Ft. Landerdale, Florida, U.S.A., November 6 - 10, 1995. IEEE Computer Society Press.

[2] Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1):337–344, January 1998. Special Issue on Formal Methods.

[3] Chris Ho-Stuart and Shaoying Liu. A Formal Operational Semantics for SOFL. In *Proceedings of the 1997 Asia-Pacific Software Engineering Conference*, pages 52–61, Hong Kong, December 1997. IEEE Computer Society Press.

[4] Shaoying Liu, Masashi Asuka, Kiyotoshi Komaya, and Yasuaki Nakamura. An Approach to Specifying and Verifying Safety-Critical Systems with Practical Formal Method SOFL. In *Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'98)*, pages 100–114, Monterey, California, USA, August 10-14 1998. IEEE Computer Society Press.

[5] A Jeff Offutt and Shaoying Liu. Generating Test Data from SOFL Specifications. *Journal of Systems and Software*, (to appear).

[6] Shaoying Liu, Masaomi Shibata, and Ryuichi Sato. The Specification, Design, and Implementation of A University Information System. Technical Report HCU-IS-99-005, Hiroshima City University, Hiroshima, Japan, 1999.

[7] Shaoying Liu. *A Structured and Formal Requirements Analysis Method based on Data Flow Analysis and Rapid Prototyping*. PhD thesis, University of Manchester, U.K., August 1992.

[8] Edward Yourdon. *Modern Structured Analysis*. Prentice Hall International, Inc., 1989.

[9] W. Brauer, G. Rozenberg, and A. Salomaa, editors. *Petri Nets - An Introduction*. Springer-Verlag, Berlin Heidelberg, 1985.

[10] John Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[11] Shaoying Liu and John A McDermid. A Formal Specification of Fault Trees for SAM. *Transactions of Information Processing Society of Japan*, 38(10):2014–2030, October 1997.

[12] Anthony Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software*, 13(2):66–76, March 1996.

[13] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International(UK) Ltd., 1990.

[14] David Budgen. *Software Design*. Addison-Wesley Publishing Company, Inc., 1994.