An Industrial Application of an Integrated UML and SDL Modeling Technique

Robert B. France[1], Maha Boughdadi[2], Robert Busser[2]

[1]Computer Science Department, Colorado State University, Fort Collins, Colorodo, USA

france@cs.colostate.edu

[2]Motorola, Florida, USA

fmb030@email.mot.com

ABSTRACT

The use of rigorous software development techniques provides opportunities for the early detection of defects in software requirements and design. Early detection and correction of defects can, in turn, reduce the time spent on implementing defective software models. A number of rigorous techniques that target specific development phases exist. Industrial use of these techniques requires the development of appropriate integration strategies that result in cohesive sets of techniques that effectively cover the software development process. In this paper we present some of the experiences we gained in developing and applying an integrated requirements and design modeling approach.

1. Introduction

Though *formal specification techniques* (FSTs) [Saie96] have existed for over two decades, their uptake in industry has been very slow. Early FSTs were mostly textual, utilized mathematical notations, and often required a mathematical maturity beyond the experiences of most software developers. Furthermore, creating and understanding formal models using textual FSTs that make direct use of mathematical notations can be difficult because of the large gap between the mathematical concepts and the real-world concepts being modeled. In some cases, the effort required to bridge the gap can result in less than satisfactory formal models, which in turn can lead to the introduction of defects that are not easily uncovered. These problems can diminish the use of formal models as a means of communicating requirements and designs and can negate the benefits of applying FSTs. The lack of tools to support the complex tasks of building and analyzing formal models can also make the application of FSTs impractical in some cases.

The above mentioned and other problems with the application of early FSTs has resulted in work on the development of industrial-strength FSTs [Lars96, WIFT95]. A key realization is that a practical development modeling and analysis approach requires (1) a judicious mixture of formal and informal techniques [Fran97], and (2) a set of integrated tools supporting the construction, analyses, and transformation of software models, and the linking of software models across development phases (traceability).

In this paper, a development environment that consists of an integrated set of techniques that tolerate informality and an integrated set of tools that support the application of the techniques is called a *Rigorous Development Environment* (RDE). This is to be contrasted with *Formal Development Environments* (FDEs) that do not tolerate any form of informality. FDEs become practical when the development processes can be codified in an efficient and effective manner. This is more likely to occur in application domains that are very mature, that is, there exists a wealth of experiences that can be usefully transformed to formal models of development. Unfortunately, there are very few application domains that have this characteristic. An effective RDE can be a reflection of an application domain's level of maturity: it reflects what is formalizable and what is not yet efficiently formalizable.

In this paper we present some of the experiences gained in an industrial software development project that utilized techniques from an RDE that we are currently developing. The RDE is centered around object-oriented (OO) requirements specification techniques and the *Specification and Description Language* (SDL) [Ells97]. The OO techniques are based on the *Unified Modeling Language* (UML) Class Diagrams and StateCharts [UML97]. The RDE that is being developed and deployed uses the UML-based techniques for requirements modeling and the SDL for design. It supports the translation of UML requirements models to SDL models, and the supporting commercial toolset supports traceability across the requirements and design models.

In section 2 we give an overview of the technology transfer (TT) project that provided the context for our work on the product development project. In section 3 we give an overview of the product development project and discuss our experiences with the use of the integrated

UML/SDL techniques. In section 4 we conclude with a discussion of the benefits of using rigorous techniques. This paper assumes that the reader is familiar with the SDL and UML notations.

2. Facilitating FST Technology Transfer

In 1997 an exploratory project on the use of FSTs in Motorola's Paging Division was initiated on the industrial side by one of the co-authors who was then the manager of a strategic group within a Motorola division. His group was responsible for identifying new technologies that could significantly enhance the productivity of development engineers in the division. His good relationship with key product engineers in the domain was key to getting the TT project off the ground. Two university researchers (a graduate student and a professor) and the Motorola manager made up the core of the TT project team.

The primary objective of the ongoing TT project is to develop an RDE that enhances the best experiences currently available within the division's software development environment. The intent is to introduce FSTs as supplements to proven development techniques that are currently in use within the division. This, we feel, allows for a gradual evolution of the software development environment to a more rigorous development environment. In the development projects that we worked on as part of the TT project, much attention was paid to determining precisely the role of the FSTs in the entire software development lifecycle and defining the interfaces between currently used techniques and the proposed FSTs.

In summary, the TT project has the following objectives:

1. To develop and incorporate an RDE based on proven technologies into the division's software development environment.
2. Instill an appreciation for rigorous techniques among the division's software engineers and provide the means by which the engineers can acquire the necessary skills to effectively use the RDE.

At the time the TT project was initiated the product engineers followed a well-defined process that produced mostly textual requirements and design documents. Procedures and supporting tools for tracking faults, managing versions of documents, and for carrying out formal reviews were utilized in product development projects. We presented the use of FSTs as a way of enhancing the current process through the ability of FSTs to produce rigorously analyzable requirements and design models. We argued that analysis of such models can enhance error detection at the requirements and design phases of development.

Discussions with the engineers and examination of requirements and design documents revealed that variants of state machine models were being used to model significant parts of behavior. The engineers tended to communicate their understanding of behavior in terms of states and events that moved the system from one state to another, but the notions of state and event were informally understood by some engineers. Engineers also pointed out the need to model interactions between software components. For this reason, we focused on identifying proven, industrial-strength FSTs based on communicating components, in which components can be specified in terms of state transition systems.

The need for practical analysis techniques for state-machine models led us to consider model-checking techniques. These techniques require no mathematical skills on the part of the engineer. On the other hand, effective use requires that the engineers be able to determine the state exploration algorithms that are appropriate to their models, and understand the limitations of the algorithms. Another advantage of using state machine models is that validation through model animation is possible. State-machine models can be associated with an operational semantics that can be used as the basis for mechanisms that execute the models. Developers then have the ability to 'execute' their design models and observe the behavior

The *Specification and Description Language* (SDL), has been chosen to be aplyed to a product development project. The decision to use SDL in the TT project was driven by three significant factors:

1. SDL seemed to utilize modeling concepts that the product engineers could relate to. SDL models could also be analyzed to detect errors arising from faulty interactions. These types of errors are often difficult to detect in less formal models and can be difficult to correct if identified at the code level. The graphical form of the models also appealed to the engineers.
2. Commercial tools that provided significant support for multi-person development of SDL models, and that provided an integrated set of tools that covered requirements, design, test case and code generation were available.
3. The SDL notation and tools were being used and extended in other Motorola product domains. This made available to the TT project an experience and research base from which lessons learned and exemplar models could be obtained.

In discussions with the product engineers, risks and problems associated with the use of SDL were identified.

1. The SDL notion of states and transitions may not be compatible with some of the existing notions of states

and transitions held by product engineers. For example, in some notions of state there could be actions taking place in a state. In SDL a state is a point in the behavior of the system in which it is waiting for an event to occur. All actions in an SDL model take place in transitions. Some engineers could have problems expressing their understanding of behavior in SDL terms.

2. Some of the engineers tended to view product behavior in terms of code-level concepts. Such engineers could find it difficult to abstract to a level where SDL modeling yields useful insights.
3. SDL does not support the modeling of time. This means that it cannot be used to support timing analysis. It was necessary to make clear to the engineers that SDL should be used only to analyze interactions. SDL, however, does provide limited support for modeling time dependent interactions through the use of timers.
4. Transitions are effectively instantaneous in SDL. This means that interrupt-driven behavior cannot be modeled in a straightforward manner.

Training and timely guidance during product development were viewed as the primary means for minimizing the risk of misusing the notation and for making clear the limitations of the modeling technique.

The scope of the pilot product development project was determined by the development needs of the engineers. The product engineers had already developed a requirements document and they recognized the need for rigor during design, thus the design phase became the focal point of the pilot TT project. In the project, the SDL was used primarily at the design phase to gain insights into how a reusable framework could be integrated into the product and to document the design. The experiences from this project are described in a paper [ISSRE98].

Following this project it was decided to try an integrated requirements and design modeling approach on another, smaller, project. This project, called the RCA project, utilized UML modeling techniques at the requirements level. The UML models were then transformed to SDL models from which design proceeded. The experiences we gained on the RCA project is presented in this paper.

3. Experiences from The RCA Project

In this section we outline our experiences with applying the loosely integrated UML and SDL notations to the RCA project.

3.1 The RCA Project Process

The RCA project was concerned with reengineering an existing implementation of a paging feature to incorporate new functionality. A major goal of the reengineering project was to develop precise requirements and design models of the RCA implementation that would ease the task of evolving the RCA software.

SDL provided the concepts needed to model the RCA functionality: The application had stimulus-response characteristics, and behavior was determined by application states. A requirements process that involved the use of *Message Sequence Charts* (MSCs) and UML *StateCharts* was developed and used on the project. These notations were chosen because they support state-based and interaction-based modeling at the requirements (problem-oriented) level. Furthermore, a tool that generated initial SDL models from UML StateCharts was available.

The process used to develop the requirements and design models for the RCA project is outlined below:

1. *Requirements Modeling*
   (i) *Exploratory phase*: Modeling of interactions between application and its environment
   (ii) *Detailing phase*: Modeling of externally observable behavior of the application
2. *Design Modeling*
   (i) *Conversion phase*: Transformation of requirements models to first-cut design models
   (ii) *Elaboration phase*: Modification and refinement of first-cut design models

The *Exploratory* phase of the requirements modeling activity is concerned with determining the scope of the application, identifying required functionality, and determining the relationship the application has with its environment. The use of formal techniques at this phase can be counter-productive for the following reasons:

1. There is often insufficient information to develop precise statements of behavior.
2. The wide gap between formal models and “real-world” concepts can hinder understanding. Engineers are less likely to gain insights that can lead to an initial understanding of the application by examining formal models.

It is important that notations that facilitate initial understanding of an application be applied in this phase. For this reason we used MSCs, an informal and intuitive modeling notation, in this phase.

Once an initial understanding is developed, a more precise model can be developed. The *Detailing* phase is concerned with developing precise specifications of required behavior. We used StateCharts and documented the requirements as precisely as possible in English during this phase of the RCA project.

The *Conversion* phase of the design modeling activity is concerned with generating an initial design model from the requirements models. This phase helps to initiate traceability links between the requirements and design models. In the RCA project the transformation of the StateCharts to first-cut SDL models was done by hand using rules that were extensions of the basic rules provided by the SDL tool that we used. The tool provided support for linking elements of the requirements models to elements of the SDL model.

The first-cut SDL produced in the *Conversion* phase is not likely to yield an optimal design. This should not be surprising since the requirements models are intended to capture problem-oriented concepts (i.e., the resulting structure is not dictated by solution-oriented concerns such as performance, fault-tolerance, and reliability), while the design models are intended to capture solution-oriented (and programming-language independent) concerns. The *Elaboration* phase is concerned with modifying the initial design structure so that it satisfies the functional and non-functional requirements of the application.

3.2 Applying the Modeling Techniques

The RCA project was carried out by the RCA engineers, with the TT members acting as consultants. to fine-tune the process to the engineers' needs when required.

During the Exploratory phase the initial attempts at building MSCs helped the engineers to scope the application domain appropriately. The approach taken to MSC modeling was to model the application as an object (the system) and the other entities that it interacted with as external objects (the environment). In the end the engineers came out of this process with a better understanding of the application based on a well-defined scope, and the TT members were able to understand the application that was under development. At the end of this phase all concerned came away with a common understanding of the application being modeled, and the understanding was well-documented in MSCs that were organized into high-level MSCs (HMSCs). A HMSC depicts the relationships between MSCs.

After the MSCs were developed an attempt was made to develop a UML Class Diagram (CD) for the application. It became evident quickly that the RCA was essentially a behavioral entity and that the CD did not provide any useful insights. For this reason our attention turned to developing a UML StateChart for the RCA application. The MSCs showed the interactions that took place between the RCA application and its environment, but said nothing about the effect of the interactions on the RCA application (in terms of externally observable behavior). The StateChart modeling technique was used to provide this information. An initial StateChart was developed from the MSCs. We used the following correspondence between StateCharts and MSCs: A state in a StateChart is essentially the period between two input signals on an object in the corresponding MSC; an input signal (to the application object) in an MSC is an event in a StateChart.

After the StateChart was created, design commenced by converting the StateChart to an initial SDL model using a tool from the SDL tool environment that was available at the site. Unfortunately, the result from the tool required extensive rework. The TT team then developed a set of rules that improved upon the rules provided by the tool and applied them to the RCA StateChart. The result was an initial model that was an improvement over the model produced by the SDL tool. The rules we developed took into account a fundamental difference between StateCharts and SDL: In StateCharts tasks can take place within a state; in SDL all tasks take place on transitions. A sampling of our rules is given below:

- StateChart *send events* correspond to SDL output signals. StateChart *input events* correspond to SDL input signals. (See Figure 1(a))
- A *conditional transition* on a StateChart, in which the condition is expressed in terms of parameters of the event causing the transition, is transformed to a SDL transition with a decision box. (See Figure 1(b)) If the condition is not expressed in terms of parameters of the transition event then the transition is transformed to a SDL enabling condition. (See Figure 1(c))
- If a StateChart state has an *entry action* then the corresponding SDL state will have the action on every transition leading to the state. If a StateChart state has an *exit action* then the corresponding SDL state will have the action on every transition leading away from it. (See Figure 2(a))
- An *activity* in a StateChart state is represented as a value-returning procedure. The state in which the activity is carried out is encapsulated in the procedure definition. See Figures 2(b), 3, and 5 for illustrations of the transformation rules involving internal activities. The value returned by a procedure representing an internal activity indicates whether another event occurred before completion of the activity or the activity was completed. The procedure is called from within a decision box in the SDL process definition and the branches lead to the states that the system moves into after leaving the state with the internal activity. For example, in Figure 3, the internal task, *task2*, is called from within a decision box. The procedure can return one of three values: *e2occur* (indicating the occurrence of event *e2*), *e3occur* (indicating the occurrence of event *e3*), and *t2completed* (indicating the completion of the task). Note that the state *S2* is not shown in Figure 3. This state is encapsulated in the procedure definition for *task2.*

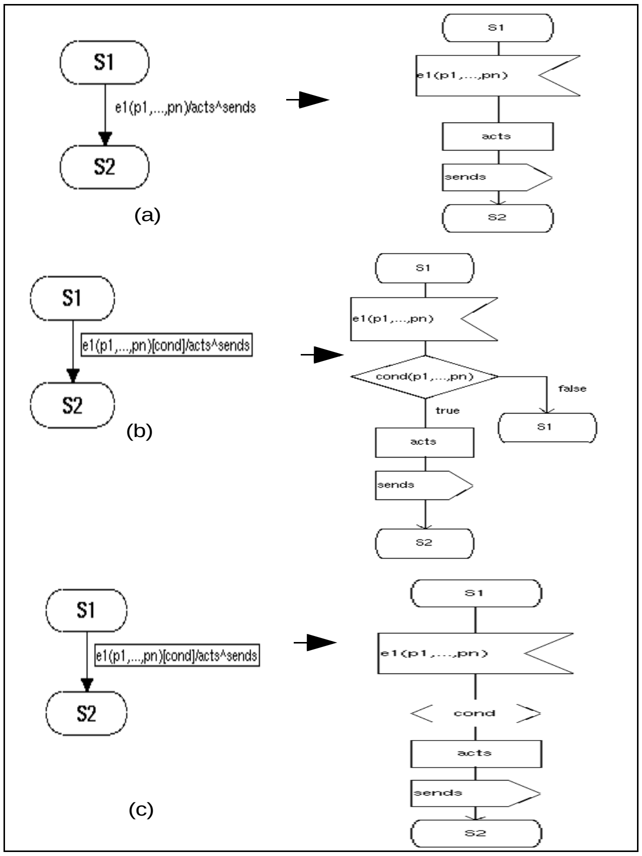

Figure1: SC to SDL Examples

The first-cut SDL model was extended and refined, and the resulting model was analyzed, simulated and validated. The simulation helped in debugging the design. Design flaws were uncovered during the simulation. In some cases the errors were a result of conflicting requirements, which resulted in modifications to the requirements models. After simulating the SDL models several times, the developers had a better understanding of the desirable features of the design. This resulted in the restructuring of the RCA process. Another sequence of simulations helped uncover other flaws which were corrected.

After simulation, the model was validated. This involved generating a state reachability graph and systematically going through the state space to identify problems such as deadlock. This type of automated analysis is more thorough than simulation. The validation uncovered some invalid interactions such as signals sent to stopped processe invalid interactions such as signals sent to stopped processes.

3.3 A Revised Development Process

After completion of the project we used our experiences to refine the development process. Our refinements were based on the following observations and insights:

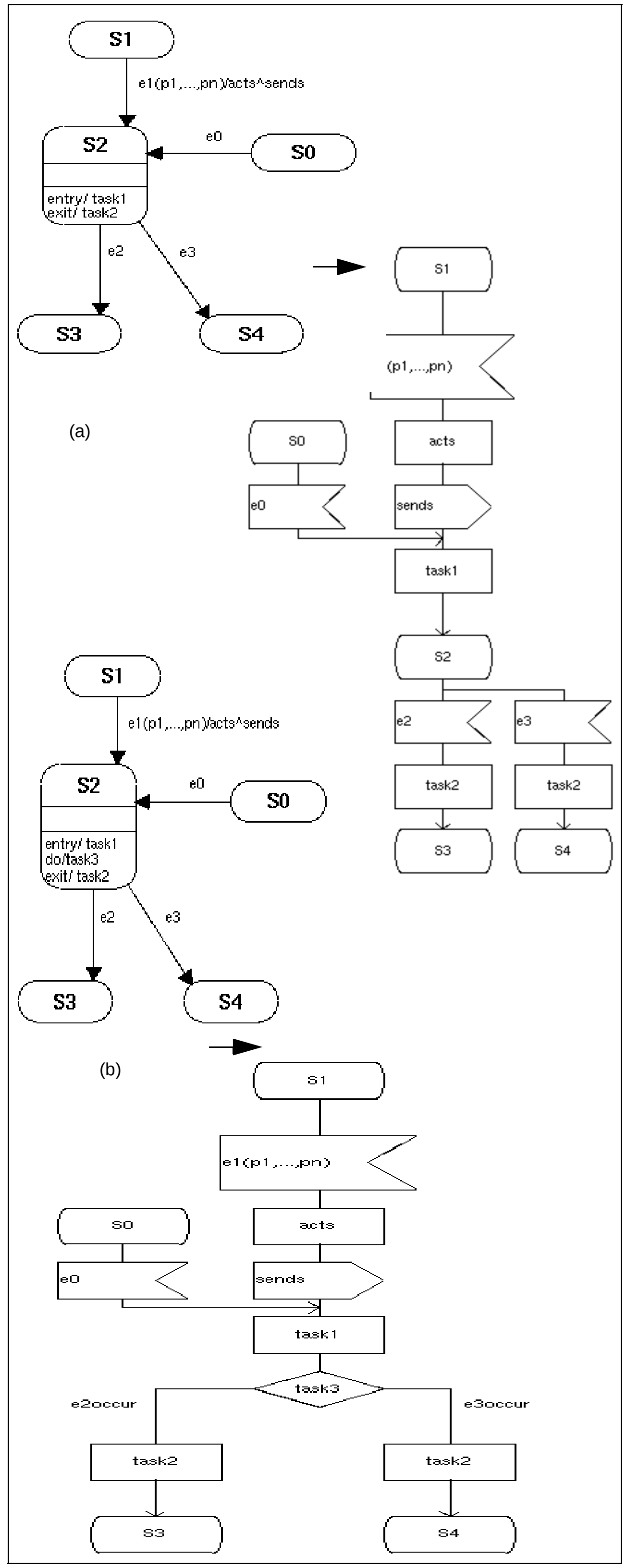

Figure2: SC to SDL Examples

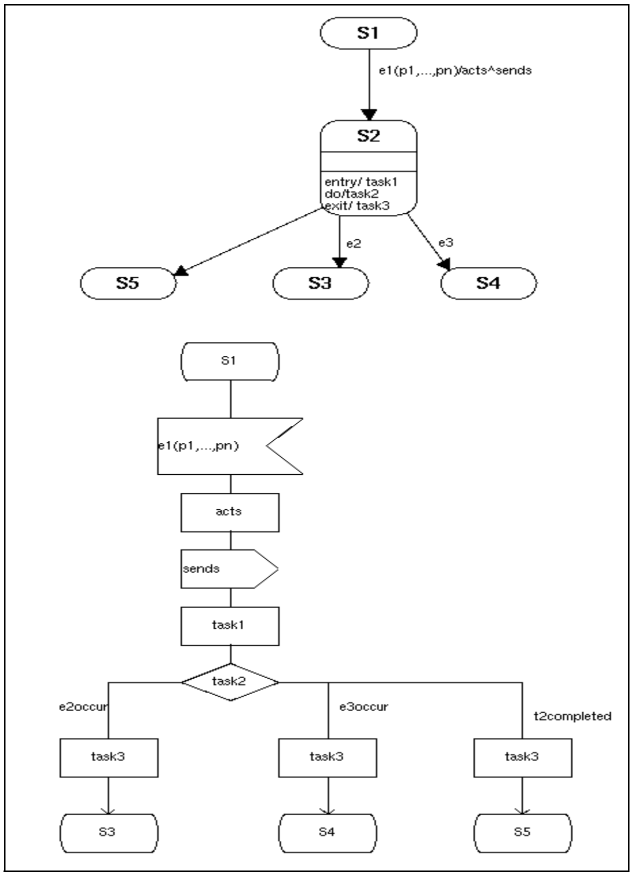

Figure3: SC to SDL Examples

- The current transformation from requirements to design results in bottom-up development of the initial design: The StateCharts are converted to partial process definitions, which are then grouped together to form SDL subsystems and blocks. A top-down approach might be more appropriate, thus a transformation approach that generates a block-level view that is consistent with the generated process-level view is desirable.
- A better structure can be obtained if some early architectural design decisions are made before transformation.
- There is the temptation to include design information in StateCharts. A clear distinction between requirements-level and design-level concepts is needed.

The above considerations led us to the following process model:

1. *Requirements Modeling*

   (i) *Exploratory phase*: Modeling of interactions between application and its environment using MSCs.

   (ii) *Detailing phase*: Modeling of externally observable behavior of the application using Class Diagrams and StateCharts. Each active class (i.e., a class with objects that have ongoing externally observable behavior once created) in the Class Diagram is associated with a StateChart that defines the behavior of its objects.

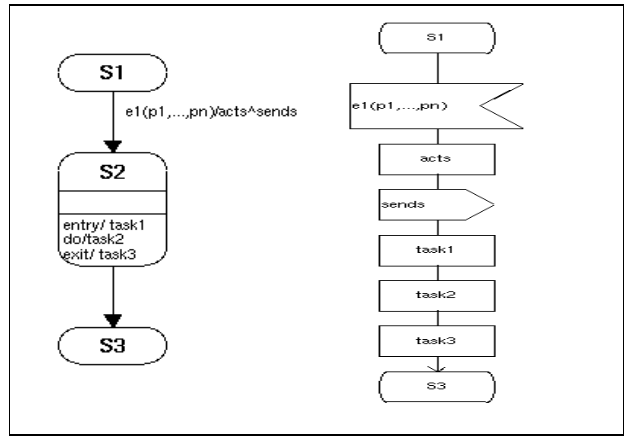

Figure4: Sc to SDL examples

2. *Design Modeling*

   (i) *Initial Architectural Modeling*: The classes in the requirements-level Class Diagram are grouped into packages. The requirements-level MSCs are refined by decomposing the MSC object representing the system into objects, each representing a Class Diagram package. The resulting *initial design MSCs* describes the interactions between the packages. Some of the design decisions made at this level may affect the requirements-level StateCharts, in which cases they are modified. The modified set of StateCharts are called *design-level StateCharts*. The initial design MSCs, design-level StateCharts, and the packaged Class Diagram describe the initial architecture of the design.

   (i) *Conversion phase*: The initial architectural design is transformed to a first-cut SDL model. The MSCs and packages are used to create block and subsystem-level descriptions, and the StateCharts are used, as before, to create process level definitions. Packages are modeled as blocks. Interactions between packages in the design MSCs are modeled as signal interactions between blocks. If the packages are further decomposed into other packages, then the blocks are decomposed into corresponding subsystems. Eventually packages are decomposed into classes. An active class is transformed into a process whose behavior is defined by the SDL process definition generated from the corresponding StateChart.

   (ii) *Elaboration phase*: Modification and refinement of first-cut design models.

We are currently developing a set of automatable rules to support the conversion phase as described above. While the above process has been tried on classroom problems we have not yet had the opportunity to apply it to an industrial problem.

4. Conclusion

In this paper we describe our experiences with applying rigorous requirements and design modeling techniques on a development project. The engineers that worked on the project deemed it a success because they were able to capture significant errors in design and requirements before they started to code. They also stated that the modeling activity led to a deeper understanding of the application and its design.

Coding designs that are not well-thought out can lead to considerable time spent on debugging software. Furthermore, it is difficult to trace code changes to requirements and design faults, leading to discrepancies among these models. Under such conditions, reuse and software evolution are extremely difficult and costly. It is easy to neglect the process to realize immediate time savings, and then disregard the effect on maintainability and reuse. Rigorous development requires that more time be spent analyzing requirements and designs, but the result is less time spent debugging code and higher quality software. The sooner errors are uncovered the less costly they are to fix [Blum92, Boe81].

The importance of precise documentation that is consistent with coded implementations should not be undervalued. In the RCA case the documents will be used to convey the RCA architecture to internal development groups that wish to incorporate the RCA into their products. The precision of the documents means that there will be less need for direct communication with the developers. More importantly, the documentation provides an executable model that is more abstract than the code implementation, thus allowing potential users to gain a quicker understanding of the software's functionality.

5. References

[Blum92] Bruce I. Blum, *Software Engineering: A Holistic View*, Oxford University Press, 1992.

[Boe81] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[Ells97] Jan Ellesberg, Dieter Hogrefe, Amarde Sarma, "SDL: Formal Object-Oriented Language of Communication Systems", Prentice Hall, 1997.

[Fran97] Robert France, Jean-Michel Bruel, Maria Larrondo Petrie, "An Integrated Object-Oriented and Formal Modeling Environment", Journal of Object-Oriented Programming (JOOP), 10(7), 1997.

[Hall96] J. Anthony Hall, "Using Formal Methods to Develop an ATC Information System", IEEE Software 13(2), 1996.

[Holz91] Gerard Holzmann, "Design and Validation of Computer Protocols", Prentice Hall, 1991.

[ISSRE98] Robert B. France, Maha Boughdadi, Robert Busser, "Incorporating a Formal Design Technique into an Industrial Environment: An Experience Report", Proceedings of ISSRE98, IEEE Press, 1998.

[Lars96] Peter Gorm Larsen, John Fitzgerald, Tom Brookes, "Applying Formal Specification in Industry", IEEE Software 13(3), 1996.

[Saie96] Hossein Saiedian (ed.), "An Invitation to Formal Methods", Computer 29(4), 1996.

[UML97] The UML Group, *Unified Modeling Language v 1.1*, OMG Standard.

[WIFT95] Susan Gerhart, Robert France, Maria Larrondo-Petrie, (eds.), *Proceedings of the IEEE Workshop on Industrial-Strength Formal Specification Techniques*, IEEE Computer Society Press, Los Alamitos, CA, 1995.