Incorporating a Formal Design Technique into an Industrial Environment: An Experience Report

Robert France
Department of Computer Science
Colorado State University
Fort Collins, Colorado

Maha Boughdadi, Robert Busser
Motorola, Boynton Beach, Florida
mboughda,rbusser@paging.mot.com

Abstract

Formal Specification Techniques (FSTs) are mathematically-based techniques that can be used by system designers to specify the desired behavior of a computer-based system and its structural properties. It has been hypothesized that the use of FSTs in software design has two primary advantages: (1) the act of formalizing design concepts provides the designer with insights that can lead to a deeper understanding of solutions and can uncover faults in designs, and (2) rigorous analysis of the design can reveal errors not easily captured in reviews of informal models. This paper describes a technology transfer pilot project that was initiated to test this hypothesis in an industrial, rather than academic, environment.

1. Introduction

Uncovering requirements and design defects at their point of generation can lead to an overall savings in development costs, time, and effort [Blum92]. It is well-known that cost and effort increases the later one uncovers and corrects defects in requirements and design in the software development process [Boe81]. Consider the case of correcting a design flaw identified during code testing. This involves re-designing and re-implementing the parts of the design that are affected by the flaw. If the design error was caught during the design phase then no time is wasted converting the bad part of the design to code. When design errors are uncovered during code testing, the re-design is sometimes done implicitly at the code level and is not reflected in the corresponding design documentation. This results in discrepancies between design and code documents which can later increase the cost of correcting and evolving the software. Impact analysis is more difficult at the code level than at the design level because of the different levels of abstraction, thus re-design at the code-level can result in inadequate impact analysis that can lead to future problems.

The need to uncover and correct software defects during the requirements analysis and design phases of software development has led to interest in *Formal Specification Techniques* (FSTs) [Hall96,Lars96,Saie96]. FSTs are mathematically based techniques that provide formal notations for precisely modeling system properties and mechanisms for analyzing the formal models. They are used at the requirements and design phases primarily to uncover ambiguities, missing details, and inconsistencies in software models. Their ability to do this stems from their use of precise notations that can be rigorously analyzed. FSTs provide two opportunities for uncovering defects:

- *Formalization*: The formalization of system concepts at the requirements and design levels requires developers to carefully select and precisely define the terms that they use to express the concepts. This can lead to a deeper understanding of system concepts (at the appropriate levels of abstraction) that, in turn, can lead to the identification of faults. The type of faults that can be uncovered in this way are those that arise from ambiguous statements, inconsistent statements, missing details, and incorrect assumptions.

- *Rigorous Analysis*: The formal models produced by FSTs can be rigorously analyzed. Such analysis can uncover subtle errors that are not easily identified using documentation review techniques. Rigorous analysis can be static or dynamic. Static analysis involves deriving properties from explicitly stated properties using derivation mechanisms (e.g., logical deduction). Dynamic analysis involves carrying out behavioral experiments and observing the effects (e.g., design animation/simulation, path analysis).

Though FSTs have existed for more than two decades their uptake in industry has been very slow. Early FSTs were mostly textual, made heavy use of mathematical notation, and required developers to carry out manipulations that required a mathematical maturity beyond their experiences. Furthermore, creating and understanding formal models using these FSTs can be difficult because of the large gap between the formal modeling 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. 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 [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 ongoing technology transfer project that is concerned with incorporating an effective RDE into an industrial software development environment. The RDE currently 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 UML is a recently approved *Object Management Group* (OMG) standard for OO modeling notations. 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 this paper we focus on the use of the SDL component of the RDE in a pilot project. A brief overview of a recently completed second project that utilized the full RDE is given in section 3.2.

In section 2 we give an overview of the technology transfer (TT) project and its objectives. In section 3 we present the experiences gained through the development and the application of the SDL-based RDE on a pilot product development project. In section 4 we list the lessons learned, and in section 5 we conclude with an update on the status of the TT project.

2. Facilitating FST Technology Transfer

The technology transfer (TT) project was initiated as a result of a summer project that explored the possibility of using FSTs effectively in the industrial sponsor's software development environment. This exploratory project was spearheaded 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. The manager had previous practical experience with formal techniques and recognized the benefits that could be gained if they were effectively applied in the division's product development environment. His good relationship with key product engineers in the domain was key to getting the TT pilot project initiated. Two university researchers (a graduate student and professor) and the Motorola manager made up the core of the TT project team.

The summer project involved identifying formal modeling concepts and notations that were consistent with, and reinforced the views and internalized conceptual models, held by the more experienced software developers in the application domain. Discussions with experienced engineers and examination of requirements and design documents for developed and in-development products led to the consideration of FSTs based on state machines. Following these discussions, a formal state-based modeling technique was used to model and analyze a component of a reusable framework that was currently being implemented at the site. The resulting analysis showed that FSTs can uncover subtle design errors. The problem that was uncovered pertained to the effect of concurrent accesses to a shared resource. This led to a reconsideration of what modules were responsible for maintaining the integrity of the resource. The promising results led to the initiation of the TT project.

The primary objective of the 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 fitting the FSTs to existing good practices within the environment. This meant 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.

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. An immediate goal of the pilot project was to equip the engineers with the tools and techniques that supported the identification and correction of errors during the design phase of a project. The other goals were (1) to understand and define the steps needed to incorporate an RDE into an existing software development group and (2) to mature the group to a point of self-sufficiency and effective utilization of the design tools and techniques.

A data-flow model of the tasks in the initial phase of the TT project is shown in Figure 1. The round-edged boxes represent major project tasks and the arrows represent information flowing into and out of tasks. The first task of the project focused on identifying suitable rigorous modeling concepts, modeling notations, and analysis techniques for the application domain. This was followed by a *Tool Evaluation* phase in which tool support for the techniques and notations selected in the first phase were identified and evaluated. The *Pilot Project Planning* task involved developing a plan for using the RDE in the targeted product development project. In *Project Execution*, the techniques, tools, and supporting personnel (the RDE) were deployed on the target product development project. Use of the RDE was monitored, that is, feedback from the project was used to fine tune the application of the RDE. After the target product development project was completed, the experiences were analyzed and distilled into a list of lessons learned.

The remainder of this section gives a brief overview of each of the activities shown in Figure 1

.

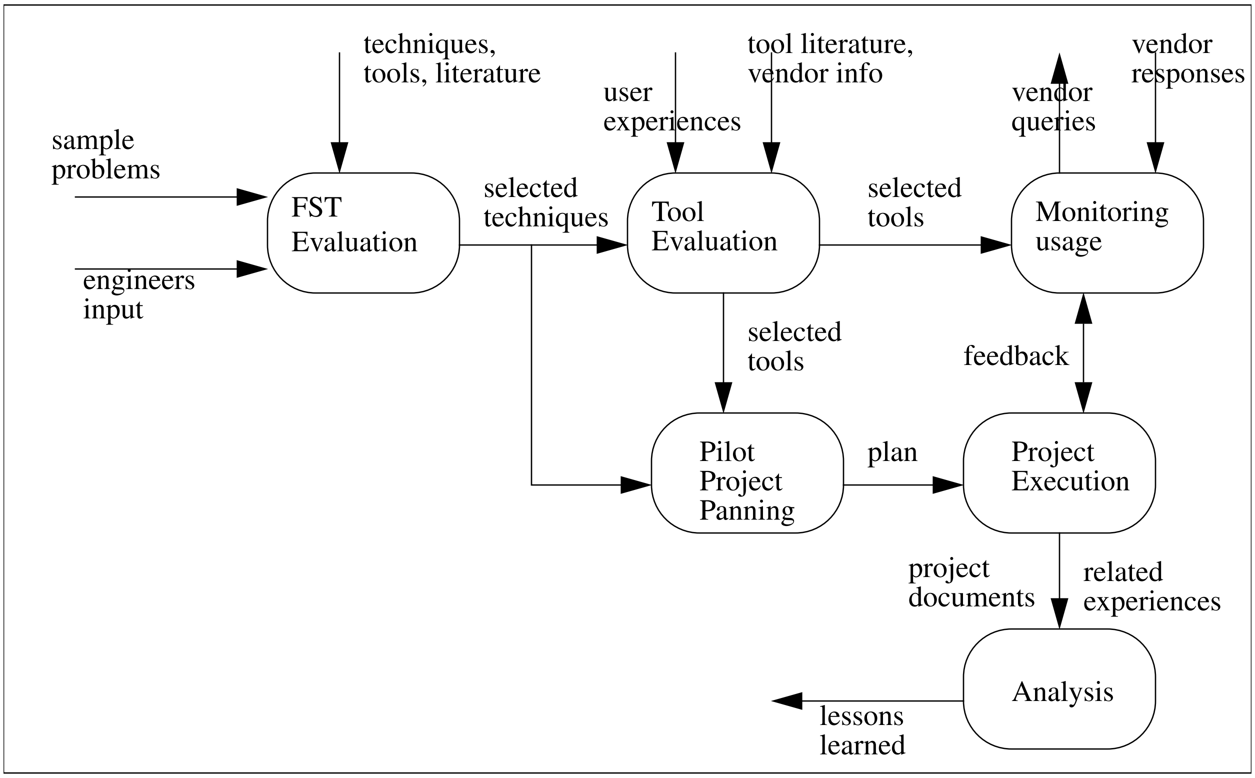

Figure1:Initial RDE Technology Transfer Activities

2.1. Evaluation of FSTs

Selecting an appropriate FST for use in a product development environment requires (1) understanding of the development culture in terms of actual development practices (as opposed to documented processes), (2) understanding of the internalized models of the product held by experienced product engineers in the domain, (3) knowledge of the development skills possessed by typical product engineers in the domain, (4) an understanding of product characteristics, and (5) an awareness of the types of requirements and design errors that typically occur in the domain.

The product engineers that we worked with on the TT project were responsible for developing software for paging devices. The programs developed in this domain are required to be efficient, that is, they must make efficient use of hardware resources, including memory. The software is best characterized as embedded, soft real-time, event-driven, with a high degree of interactions among software and hardware components. The products in the domain were becoming increasingly more complex as a result of a growing feature set for paging devices. The engineers that participated on the TT project recognized that techniques that helped them manage the complexity, in particular, techniques that helped them analyze feature interactions at the design level, could enhance the quality of the products, and could reduce the time typically spent on carrying out code fixes.

At the time the TT project was initiated the product engineers followed a well-defined process that produced mostly textual requirements and design documents. In some cases, variants of informal modeling notation from structured techniques were included in the 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.

In discussing the applicability of FSTs and the suitability of formal notations with the engineers it was generally felt that there was a wide semantic gap between the mathematical models found in FSTs based on logic and set theory and a product engineer's internalized conceptual model of the product. The engineers did not have the experience and background needed to bridge the gap effectively. Use of such FSTs was felt to be risky because the lack of experience could lead to the engineers producing faulty formal models, and the effort required to create and understand the formal models could have a negative impact on productivity and quality. For this reason we considered only FSTs that did not make heavy use of textual mathematical notation, and for which model analysis did not require developers to directly manipulate textual mathematical expressions. The engineers appreciated the need to identify errors at their points of generation and recognized the benefits of rigorous analysis. For a rigorous analysis technique to be effective the users must be able to relate the analysis results to the product concepts that are being modeled and analyzed. If this is not the case then there is a chance that analysis results can be misinterpreted or not understood.

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.

We identified and evaluated two state-machine notations for possible use in the environment: Promela [Holz91] and SDL [Ells97]. Promela was chosen because of its accessibility; the accompanying toolset (SPIN) and language documentation are freely available over the internet. We used Promela/SPIN primarily to demonstrate how model-checking techniques can be used to analyze software designs in the development environment. Promela/SPIN was used to model part of a reusable design framework that was currently being implemented in the domain. The modeling and analysis were carried out by the university researchers in consultation with the engineer responsible for designing and implementing the framework. Analysis of the model revealed the possibility of an undesirable interaction, as mentioned at the start of this section. Despite its functionality, SPIN lacks the mechanisms needed to support development of design models in a team environment and it was not clear how SPIN could be integrated with project management and development tools. Furthermore, the fact that Promela/SPIN was not a commercially supported tool meant that supporting, adapting, extending, and enhancing the tool would be a Motorola responsibility.

At the time we were creating models in Promela/SPIN we were also considering use of the SDL. We presented examples of models created in Promela/SPIN and SDL to the engineers participating in the TT project in a one day seminar, and discussed the pros and cons of each technique. 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. Given that SDL was developed over two decades by practicing engineers in the telecommunications domain, it was not surprising that the more experienced engineers found the core SDL modeling concepts intuitive. 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 the discussions with the product engineers, risks and problems associated with the use of SDL were identified. The significant risks and problems that were identified are listed below:

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 depen-

dent 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. Furthermore, careful selection of the parts of a product design that could benefit from SDL modeling was needed.

2.2. Tool selection

Once the decision was made to focus on SDL we contacted vendors of SDL tools and obtained evaluation copies of their products. The following were considered desirable features of tool support:

1. The tool should provide support for multi-person development of models. Not only should models be configuration units, but also components of models. This would allow engineers to work on different parts of a model concurrently.

2. The tool should provide support for linking requirements and other product information to the relevant SDL model components.

3. The tool should be robust (i.e., provide useful feedback when invalid user input is entered or when the system has entered an invalid state) and reliable (i.e., it should perform in the expected manner over a specified period of continuous use).

4. The user-interface should be consistent and should allow a properly trained engineer to build and analyze models in a timely manner. The time it would take to properly train an engineer on the use of the tool should not be excessive. For example, it is difficult for most Motorola engineers to spare more than three days away from their product development and support duties.

5. The tool should be capable of interfacing with a performance modeling tool. This feature requirement reflected a desire to integrate behavioral and performance modeling tools in the environment to avoid duplication of modeling effort.

There are two major vendors of comprehensive SDL toolsets and they both offered good support for building and analyzing SDL models. In doing the evaluation we determined the tool needs of the engineers (expressed as ranked criteria for evaluating the tools), solicited experiences from other Motorola groups that had used or were using the SDL tools, and we built and analyzed models of product subsystems using the evaluation copies.

2.3. Pilot project planning

Discussions with the team leader and other key engineers from a product development project resulted in plans to apply the SDL tool in the design phase of their project. At the time discussions were taking place the project was in the early requirements definition phase. SDL would only be used to model two components of the product design. The components utilized a new reusable framework that the engineers were not familiar with and it was felt that modeling of the interactions between the components and the framework would help them better understand how the framework could be integrated into their design.

To prepare them for the use of SDL in their project we held a two-day seminar that introduced the product engineers to the SDL notation. The training material used examples from the paging domain, and the engineers started an initial model in the class. A two-day class on SDL is not enough to make engineers self-sufficient in the use of SDL. To further assist the engineers one the TT project team members was assigned to the project to provide day-to-day assistance during the design period.

2.4. Pilot project execution

The engineers used SDL to model two significant subsystems of their product that had behaviors involving interacting concurrent components. The required behavior of the two subsystems was expressed in a requirements document using a mixture of text and informal state-machine models. The engineers were required to implement the subsystems using a reusable design framework that had been developed for the product domain.

In developing the SDL models the engineers referred to the informal models in the requirements document. Insights gained through SDL modeling resulted in changes being made to the initial requirements model. The building of the SDL models was done by the product engineers, but the analysis of the models was carried out by the TT project team. Team members also participated in formal reviews of SDL models. This allowed the team to gain useful insights into some of the problems the product engineers were having with the application of SDL, and allowed them to provide timely feedback to the engineers.

Coding from the SDL models was also done manually. The SDL tool used by the engineers came with a code generator, but no attempt was made to use it because it was not clear whether the code produced would satisfy the stringent code size and performance requirements for the product under development.

2.5. Monitoring and evaluating usage

The TT project team assisted the engineers in their use of the notation by reviewing their models and providing feedback on their usage of the notation. The only exposure the engineers had to SDL prior to this project was in the short two-day training they had at the start of the project. As expected, they encountered some problems when trying to create the SDL models, and it was important to provide timely feedback to avoid frustrations that could lead to disenchantment.

2.6. Analysis of experiences

At the end of the project, the experiences of the engineers were collected and distilled into a list of lesson learned. This paper can be viewed as a public record of the results of the analysis. The results of the analysis are being used as inputs to the ongoing development of a methodical approach to creating and analyzing SDL models that is tailored to the division's development environment.

3. Pilot Project Experiences

In this section we give an overview of SDL and describe some of the more significant experiences gained.

3.1. An SDL overview

SDL is an international standard developed by the ITU-T (recommendation Z-100) for formally specifying software systems that can be modeled as communicating, concurrent processes. It was developed in the early seventies initially as a specification language for telecommunication standards, but its applicability has grown beyond the telecommunication domain.

SDL is specifically concerned with the specification of interactions between components. Systems are modeled in terms of concurrently executing state machines that communicate by sending messages to each other. The notation has a textual and a graphical representation; the graphical notation is used mainly by model builders, and the textual notation is used by tools to generate an internal, analyzable, representation. SDL is tolerant of informal notation, that is, a developer can express some aspects of behavior informally in SDL. SDL models that contain informal descriptions cannot be formally analyzed, but the ability to express behavior informally provides the developer with a path from informal exploration of design considerations to more formal representations in a single notational framework.

An SDL model is hierarchical. The top level is the most abstract view, and it is essentially a representation of the system in terms of communicating black box components called *blocks*. The intermediate layers provide increasingly detailed descriptions of the internal structure of blocks. The leaf level consists of *processes* that are defined in terms of states and transitions. An SDL state represents a stable point in the execution of a process: no activities take place when a process is in a state. All activities take place during transitions. An SDL model is an idealized model of a real system in the sense that activities (transitions) take place instantaneously. For this reason, SDL is not suitable for modeling hard real-time systems.

In summary, SDL is suitable for systems:

- that communicate with other digital systems (there is very little support for modeling human-computer interactions),
- that do not need to perform complex calculations,
- that need to operate concurrently, and communicate asynchronously, and
- whose precise behavior in time is not critical.

3.2. SDL usage in the pilot project

The typical development lifecycle that existed at the time the pilot project was carried out can be summarized as follows:

1. *Requirements modeling and analysis*: This involves analyzing the document describing the required product features developed by the marketing group and producing a detailed technical requirements document. Formal reviews of requirements are carried out and the documents are placed under configuration control. Change control is supported by defect tracking tools throughout the development cycle.

2. *Product software design*: Designs are expressed using a mixture of diagrams, informal state-machine models, pseudo-code and text. The documents are formally reviewed and placed under configuration control. Changes to requirements that are generated by design considerations are noted and reflected in the requirements document.

3. *Product software implementation*: The designs are implemented and the code optimized. Code reviews are carried out formally, and the code is place under configuration control.

4. *Product software testing*: Unit and system tests are carried out on the software.

The pilot project enhanced the above process by including SDL models as an additional product of the design phase. The intent was that the SDL models will give the engineers the ability to identify subtle flaws in their designs that are not easily identified in formal reviews of informal design documents. An important point to note is that we did not advocate replacing the informal design document with SDL models. An informal design document serves a very important communication function. The SDL models can enhance the development of the informal document in that descriptions of design elements can now be based on insights gained through SDL modeling and analysis.

In the pilot project, the input to the design activity was a requirements document that consisted of a mixture of text and informal graphical descriptions of required behavior. Two of the more complex and critical subsystems of the product were chosen for SDL modeling and analysis. The other subsystems were either not amenable to SDL modeling or did not warrant formal modeling (i.e., the engineers had significant design experience they could draw upon in developing these subsystems). Those parts that were not amenable to SDL modeling were designed using previous techniques. Those subsystems that interacted with the subsystems modeled in SDL were treated as elements in the environment of the SDL models that provided and consumed signals to and from the models.

The engineers created the SDL models on their own, with assistance from the TT project team. A few of the engineers had difficulty thinking about their design in SDL terms. A notable problem was the tendency to use SDL as a flowcharting notation. The problem stems from the larger problem of determining an appropriate level of abstraction. The engineers that had this problem tended to understand their designs in programming language terms, thus they tried to express functional details that were not relevant to the modeling and analysis of interactions. Getting the engineers to accept that the design model is not an exact replica of the code, and that it is desirable to leave out irrelevant details, was not easy. Some engineers felt uncomfortable with the abstraction process (they essentially felt that analysis is incomplete if all details are not considered). As a compromise we showed the engineers how they could encapsulate detailed functionality as procedures and use procedure calls in SDL transitions. On the positive side this made the transition to code easier. On the negative side, this made the SDL models larger and more difficult to analyze.

Another problem the engineers encountered during the pilot project concerned modularization of the design. While there was an implicit architecture for the product family, this was not precisely defined. This led us to consider the use of object-oriented (OO) modeling techniques for structuring the design. We had the opportunity to apply these ideas on another project. The project was concerned with developing software support for a communication protocol. In the project requirements models and high-level designs were expressed using UML Class Diagrams and State Charts. These models were then transformed to partial SDL models using rules that are extensions of the rules implemented in a commercial SDL tool. Analysis of the SDL models produced by the engineers uncovered significant design errors. Furthermore, the engineers felt that the discussions that took place while they were developing the models led them to a deeper understanding of the requirements and the proposed solutions. The engineers on this project became advocates for the use of the techniques in their application domain.

The pilot project and the follow-on second project has generated much interest in the Motorola environment, as reflected by the number of project teams that have expressed an interest in using the techniques. A key to the success was the involvement of the engineers in all aspects of the TT project.

4. Lessons learned

The technology-transfer project has been successful in demonstrating that FSTs can enhance one's ability to detect design flaws during the design development phase. Among the lessons learned during the project was the importance of having a *champion* for the FST within the development community to promote its use. The champion plays a major role in providing the leadership necessary to encourage the engineers to try the new approach. In addition, the champion acts as a liaison between the development community, the tool vendors, and the computer support organization.

Another lesson learned was that a greater emphasis on integrating the design modeling process with the design analysis activities is required. During the design of one of the larger subsystems in the pilot project, the engineers waited until the end of the modeling activity to perform syntactic and semantic analyses. Delaying the analysis effort until after the modeling activity was complete, caused early mistakes to propagate throughout the model. This made the analysis and model correction activities tedious.

Finally, we found that providing the developers with an SDL expert as a part of the project team on a day-to-day basis helps ease the learning curve.

The lessons learned thus far can be summarized as follows:

1. Identify 'real' problems to which the formal techniques can be applied. Contrived problems are less likely to convince developers of the utility of the techniques;

2. Involve the developers in the development of the models;

3. Be sure to stress the limitations of the techniques so that engineers do not have unrealistic expectations;

4. Champions that are well-placed greatly facilitate the technology transfer process. If one does not exist, then there is the possibility that one can be cultivated;

5. Try to identify appropriate formal techniques and notations that utilize concepts and notation that are close to the concepts and notations used by developers. Pushing your favorite FST without first understanding the engineer's internalized models of their products can be an exercise in futility;

6. Some developers find it difficult to abstract over the code level. There is the impression that analysis of the abstract models has little relationship with the implementation. One needs to make clear the nature of the problem that is being investigated and convince such developers that the details abstracted are insignificant to the analysis of the model.

5. Conclusion

The TT project is ongoing, and the success of the recent application of the integrated UML/SDL RDE, has sparked the interest of other groups on the site. One of the TT project university team members was hired by Motorola to lead the effort to incorporate more rigorous development techniques into the development environment. She is currently working with a number of groups to identify suitable techniques for their development needs. Her responsibilities include communicating with product development engineers to make them aware of the availability of rigorous techniques, to encourage use of these techniques, and to provide training on the proper use of the techniques.

The TT project team is currently developing a development process that integrates appropriate UML modeling notations and SDL for use by product engineers in the domain. It is expected that this process will evolve as more is learned about how these techniques can be effectively integrated and utilized in the development environment.

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.

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