Experiences from Introducing UML and OO in an Organization

Anders Wall
Department of Computer Engineering
Mälardalen University
P.O. Box 883
S-721 23 Västerås, Sweden
anders.wall@mdh.se

Markus Lindgren*
Bombardier Transportation Sweden AB
Östra Ringvägen 2
S-721 73 Västerås, Sweden
markus.lindgren@realfast.se

Tage Tarkpea
Bombardier Transportation Sweden AB
Östra Ringvägen 2
S-721 73 Västerås, Sweden
tage.tarkpea@se.transport.bombardier.com

ABSTRACT

In this paper we present experiences from an ongoing investigation of whether Bombardier Transportation, a company within the railbound vehicle domain, can benifit from introducing object-oriented modeling and design, into the development of train applications, or not. Bombardier Transportation administrates a broad set of safety-critical products today including, for instance, autonomous trains and high-speed trains. The investigation was carried out as a pilot project focusing on tool support for the modeling language UML, taking also emergent safety requirements and software reuse into consideration. However, not only technical issues are dealt with, the paper also discusses how to receive acceptance from the organization, and how an organization may be affected by introducing software reuse and safety requirements. The results are presented as a set of findings we have made when using UML and the modeling tool Rhapsody throughout the software development process. In particular, we report the results in the perspective of safety-related software and how such a methodology facilitates and harmonizes with existing safety standards. The results from the investigating has, so far, been well received from the organization.

1. INTRODUCTION

Moving from the imperative programming paradigm to the object-oriented paradigm and to implement a reuse-oriented development may be hard, especially when this change involves many people and resources at a company. The international company Bombardier Transportation [1] with 40000 employees world-wide and the largest manufacturer of rail-bound vehicles is today performing this investigation. Bombardier Transportation's products include intercity trains, regional trains, metro trains, light rail vehicles (trams), locomotives and driver-less people movers. The work described in this paper is focused on control application development.

A train control system is a distributed safety related real-time system, where the real-time requirements on individual sub-systems varies from soft systems, e.g. air conditioning and door control, to critical like for instance, propulsion and brakes. The nature of train applications are quite diverse in many senses, having different characteristics and requirements.

A train consists of one or more motorized vehicles or locomotives, and possible additional non motorized train cars. For passenger trains with motorized vehicles is the smallest usable train configuration a *train-set*, which normally consists of two or more assembled train cars with many driven wheel axles. Within a train-set are several computing nodes that each control different sub-systems developed within Bombardier Transportation or by sub suppliers. To fulfill safety and availability requirements these nodes are often connected in a redundant fashion. Train-sets can be connected during service to form longer trains, and still be controlled by one driver. There is a second bus system, which also supports redundancy, that is used to enable cooperation between the different train-sets of a train. To summarize, train applications are distributed in their nature, and have safety and availability requirements that in most cases require redundancy.

Reuse, in some sence, is achivied today through the devlopment of *standardized vehicles*. A standardized vehicle is treated as a mould for the development of new products within similar market segments. A new product is developed based upon a copy of the standard vehicle in which the behavior of existing features are modified, and to which new features are added, in order to fulfil new requirements. This class of reuse is referred to as *code scavenging* [2]. Thus, software reuse in its strict meaning is not utilized since, for example, multiple copies of the same code exist across the product-line that all must be maintained separately.

*Markus is working at RealFast Software Consulting AB as a consultant for Bombardier Transportation.

Train applications are mainly developed using an in-house graphical function-block programming language similar to IEC 1131 [8] or an IEC 1131 tool. The current methods and tools do not support reuse to the expected future required extent. Furthermore, Bombardier transportation is a large organization consisting of several merged and geographically distributed company sites. As a consequence, there exists an abundance of tools and methods within the organization that varies between different company sites. This variety also inhibits reuse between the sites.

In order to truely reuse software across products as well as across company sites, should the implementation of a *product-line architecture* be considered [3][2]. The product-line architecture approach will, if succesfully implemented, increase reuse, standardize basic components, methods and tools, and hence, decrease development costs in the long run. Reuse typically decreases development costs and increases quality of the product. However, in this particular application, reuse must be implemented in such a manner that also safety is encompassed.

The company is now at a decision point, where it needs to decide whether to continue develop the in-house tools to support reuse better, or if commercial tools should be used instead. A major benefit with using commercial tools is that the need for maintaining the in-house tools will be eliminated, which reduces the overall cost.

1.1 Safety

Safety is a property of a system that it will not endanger human life or environment, cause human injury or great economical loss. Software in itself can never be unsafe, only when software is part of a system that interacts, electrically, chemically or mechanically, with its environment it becomes *safety-related*. A system is refered to as safety-related if it controls or supervises a safety-critical application. Properties that are somewhat related to safety are reliability and availability. Reliability is the probability of a component, or system, functioning correctly over a given period of time under a given set of operating conditions. Availability is the probability that the system will be functioning correctly at any given time [13].

There exist several standards that describe the development of safety-related systems. In the work we performed we used the standard IEC 61508 [9], which is an international generic standard regarding safety in systems implemented using electrical-, electronic- and programmable electronic systems (E/E/PES). The standard addresses relevant safety life-cycle phases, i.e. from initial concept, through design, implementation, verification, operation and maintenance to de-commissioning, when E/E/PESs are used to perform safety functions.

In order to give a concrete form to customer's safety requirements and their implications on the development of software, it is becoming increasingly popular to internally transform the safety requirements into the form of a *Safety Integrity Level* (SIL). SIL is a discrete scale reaching from one to four depending on the risk associated with the application. This is defined in IEC 61508 which also suggests how to fulfill the requirement in terms of tools, methods, documents and processes. Furthermore, different train applications face different requirements as some are associated with higher risks than other. This is not just a *technical* problem as such a safety requirement also affects the development process, the organization and the way in which one have to do business with customers and sub-contractors.

Customers and authorities already require safety classified train systems. Thus, tools, methods, and process chosen for future development must harmonize with the safety standards. If they do, and if the development phase is adequately documented, evidence can be provided in the safety assessment instead of performing a costly revision of the project. Consequently, this is also a source for reducing development costs.

1.2 Product Lifetime

In addition to trains having real-time and safety requirements, trains also have quite a long lifetime. Typically, a train system must be maintained for a period of 20–30 years. This is also an important parameter to take into consideration when choosing future tools and programming languages, since we must be confident (as much as one can be) that the chosen tools stay around until maintenance is terminated, i.e. 30 years from now, or we should be confident that maintenance can be performed even though the tools are not available.

This paper presents some of the experiences we have made in the ongoing investigation of whether to introduce UML, object-orientation, and reuse or not at Bombardier Transportation. The first phase was performed as a technical investigation where different languages, tools and approaches were evaluated. The investigation generated a set of requirements, e.g. safety, and open questions constituting the input to a pilot project at Bombardier Transportation. The reason for running the pilot project was twofolded; 1) getting acceptance from the organization, and 2) get practical, hands on experiences. The results from the pilot are presented in this paper as a set of findings, both technical and non-technical, that we made during this work. Particular focus is on safety and how well the tools and methods accomodate to the work of producing software that harmonizes with the safety standard IEC 61508.

The remainder of this paper is organized as follows: Section 2 discusses the background and motivations for this work. In Section 3 is the ongoing work of getting acceptance for the proposed changes outlined. Section 4 presents our experiences with respect to both *hard*, technological issues regarding tools, etc., and *soft* issues related to the organization. Finally, Section 5 concludes the paper.

2. MOTIVATION AND PROBLEM STATEMENT

The motivation for introducing new technologies and methods in industry are always driven by commercial interests, so also in this case.

A major and increasing cost of producing trains lies in application software development development. This cost is

therefore on the head of the list when it comes to cutting costs. However, there is also a second objective to take into consideration, the emerging requirement on safety. Even though safety in itself does not decrease the development cost, it is in the long term an argument for being competitative on the market. It is worth pointing out that the safety requirements will make the development more expensive due to, e.g. requirements on a more strict and rigid development process, as well as tougher requirements on test coverage.

Moreover, software in train applications constantly increases in terms of size and complexity. Being a company with a strong mechanical and electro-technical tradition, the increased complexity in software poses new challanges. The reason for increased size and complexity in the software is the ever growing demand for new features. The features themselves are also becoming more complex in terms of their behavior.

The overall business vision for Bombardier Transportation is to devlop products that attract customers with respect to functionality, quality and prize. Software reuse constitutes one way, among other, for the company to attain that vision. If successfully implemented will a software reuse program decrease cost while it will increase quality. The long term goal of this work is to develop a generic software platform, i.e. a *product-line architecture*, for train applications, taking also the safety strategy into consideration. The development of a product-line architecture is not within the scope of this work. Nevertheless, the investigation must take also such a future scenario into consideration.

In order to meet the safety requirement, manage the increased size and complexity in software, and to successfully implement a reuse program, the tool support is of extreme importance. Aiming for software reuse and product-line architectures, it is very important that components and the architecture are well documented and that the documentation provides a consistent view of the actual software. Thus, we need a tool that not only support a large portion of the development process, but also encourage engineers to actually treat the models made in tools as the source.

Today, applications are developed using an in-house developed software tool. Thus, there is an additional development cost associated with maintenance of that tool. Buying a commercially available software tool will, to some extent, decrease this cost.

On the other hand, buying and depending on third-party developed tools has its associated risks. The lifetime of train applications are typically in the range of 20–30 years and it is impossible to predict the status of tool vendors in such a long time frame. Consequently, a lot of legacy systems, which must be maintained, will hang around for quite a long time. Compared to using in-house developed tools, an organization will have little or no control and influence on the development of such a tool. Consequently, one must have an emergency plan in case the worst scenario occurs, e.g., the tool vendor the company relies on goes out of business.

However, a strong argument for using well known, widely spread technologies and tools is that the possibility of hiring engineers that already have knowledge about, and experiences from, similar technologies increases. As a consequence, the burden of internal education on languages and tools will be relaxed. However, the need for internal education will not be completely removed. We will further discuss the need for education and engineering competences in Section 3.

Finally, working with modern software technologies does attract people, both externally and internally. This will hopefully increase hiring opportunities as well as keep staff for longer periods of time.

Taking all the matters discussed above into account, we have been investigating, and as first candidates selected the well known object-oriented language *C++*, the *UML* modeling language and the design tool Rhapsody from I-Logix, which supports UML. Rhapsody was selected as it is a tool in the front line, but the choice is not strategical, there exists other similar tools on the market today. We think that these choices has the potential of accommodating the vision. Note that this might not be the only way to accomplish a software product-line architecture and software reuse. The object-oriented paradigm was selected based on the results from an earlier investigation made within Bombardier Transportation. The choice was also based on the fact that UML is a modeling language that have abstractions for classes, objects, etc. Since we selected Rhapsody the variety of implementation languages was narrowed down to C++, Java or C. Clearly, C++ is the obvious choice over Java when to be used in safety related real-time software due to, e.g. the non-deterministic garbage collector in Java. Yet another possible object-oriented language which has been used when developing safety-related systems is ADA95 [12]. There exists several UML design tools that have support for ADA95 but they have not been fully investigated within the scope of this pilot project. Moreover, we think that C++ is more widely spread. It is, however, clear that whatever language is finally selected, it must have good support with respect to development tools.

It is worth noting that C++ is quite a tricky language that, if used in a bad way, on its own will increase the development costs and maintenance cost. Using C++ will increase the number of possible design decisions compared to the function block way of working. Consequently, it is easy to construct complex and poor systems if not crafted with care. The solution to this problem is education and an architecture that restricts and guides developers in their design work. Since the language itself is such a crucial matter, the actual choice must be more thoroughly investigated until a decision can be made.

In conclusion, the purpose of this work was to examine how well object-orientation and UML accommodate the development of reusable software for safety-critical train applications. The main reasons for considering these changes are to manage the increased complexity in the software, decrease development costs while increasing the quality of their systems. This, mainly through an increased level of software reuse and utilization of state-of-the-art software development tools.

3. ACCEPTANCE FROM ORGANIZATION

Considering the vast impact that the paradigm shift will have on the company, it is clear that this change will not be done without strong arguments that it outperforms today's situation. Moreover, such a big change will not come for free. Management must realize that it takes both time and money before software reuse pays off. Typically is the time to return on investment dependent on the *product cycle*. The product cycle defines how long time it takes to develop a product, or in other words, the frequency with which new products are developed. Another important factor is the similarity among the products, the more equalities, the more components can be reused, and the faster will the required investment pay off.

Studies have shown that producing a reusable component ranges from approximately 120 to 480 percent of the cost of creating a non-reusable version, and integration cost ranges from approximately 10 to 65 percent of the cost of creating a non-reusable version. The actual cost is dependent on the complexity of the implementation and the complexity of reusing it in an application [5][11].

In this case, the need for changes has been perceived in parts of the organization. Even though management has not yet accepted the change, they are interested in investigating the possibilities. As a consequence, a series of pilot projects have been financed and initiated. The decision whether to go in the direction of UML and object-orientation will be judged based upon the results from the pilots. Thus, one of the primary goals for the pilot project is to get acceptance.

However, receiving acceptance from an organization is much more than getting acceptance from management. The software engineers are the ones that will be affected the most. After all, they are the ones that will be working with the new techniques, methods, and tools. In general, people tend to be more tolerant to changes if they have taken part in the actual changing process. Thus, the key to get acceptance is *openness*. Consequently, the pilots must be carried out in such manner that engineers can be involved and have the possibility to provide their domain knowledge, experiences, and opinions. In our case, we involved, not only a reference group mainly consisting of people representing management, but also several *observers*. Typically, the observers were train application engineers having a lot of experiences working with current methods, but no or little knowledge about object-orientation and UML. They were involved at an early stage of the pilot so that they could influence the goals, i.e. the interesting results and measurements. Observers were present at all project meetings and took part of all documents, such as specifications and UML-diagrams.

The advantages of involving observers from the projects point are obvious. They informally spread the word around colleagues in the organization and as a consequence of that they also picked up their colleagues opinions and brought them back to the pilot project. Secondly, as they had no or little knowledge in object-orientation and UML, the project could estimate the *understandability* of the produced UML diagrams.

The results from the pilot has been presented in a workshop were all interested employees were invited to listen to presentations, discuss results and ask questions. The results from the first pilot have been well received, and clearance has been given to start developing an object-oriented architecture for trains. The architecture task will start late autumn 2001.

3.1 The first pilot project

As discussed in Section 3, a pilot project has been run whose main purpose was to investigate how suitable UML and object-orientation are for train application development. In this section we present the problems we were to solve in the pilot project Pilot 1. The experiences we have made concerning UML and Rhapsody for train application development are discussed in Section 4.

Even though there are results from several other industrial companies that indicate that the UML is beneficial from several points of view [6], Bombardier Transportation still wanted to perform its own investigation, focusing on UML for train applications.

The main purpose of Pilot 1 was to investigate whether UML and object-orientation is suitable for train application development taking also safety, and its implications on the development process, into consideration. In performing this investigation, a prototype for door control on a train was to be developed using UML and the supporting tool Rhapsody from i-Logix [7]. However, focus is not on the tool itself, rather it is on UML which Rhapsody, to some degree, supports. Reuse was also considered in this pilot project since the door control application developed should be reused in a succeeding pilot project.

Last but not least is the pilot project an instrument for receiving acceptance from the organization regarding the new methods. The work itself may raise interest among the engineers and, if the results are satisfactory, convincing the organization that this way of developing software indeed can be used for train applications. Openness is, as discussed in Section 3, one key issue when working with acceptance. Thus, it is important that the pilot project highlights all aspects of the method, i.e. both pros and cons.

The application we were to design and implement in Pilot 1 was a simplified version of door control on a single train car. The developed software was also supposed to be scalable, meaning that it should be able to be instantiated to control any number of train cars, with any number of doors per train car (among other things). A second pilot, Pilot 2, will be run during autumn of 2001, which will check if Pilot 1 succeeded in creating scalable software. Pilot 2 will also be collecting metrics on reuse. Note that the reusable door control framework is only designed to be reusable in Pilot 2.

A sketch of the Pilot 1 hardware is presented in Figure 1. The system consists of three main parts. First, there is a rack-based PowerPC system with Ethernet and fieldbus access for I/O. Second, there is a physical model of a train car with four doors that can be moved by electrical engines. Digital and analog I/O units are connected to the train model and to the fieldbus, enabling the PowerPC to access sensors and actuators in the train model. Part of the train model

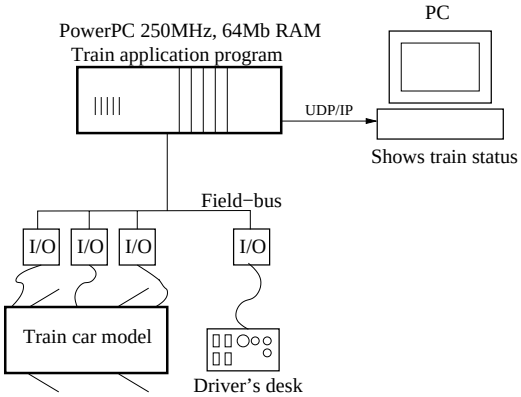

Figure 1: The system hardware setup

is also a simplified version of a driver's desk. Third, there is a standard PC with a display program that shows status information from the PowerPC system. All application logic is executed on the PowerPC system and all other devices only follows the commands from the PowerPC. On the train model there are push buttons at each door that passengers can use to open and close the doors (if allowed by the driver). The driver's desk contains push buttons and switches for controlling the doors on the train model, e.g. the driver can decide when passenger are allowed to open the doors. On the driver's desk there is also a driving stick for controlling the propulsion system of the train (a very simplified simulation of the propulsion system is implemented).

The application requirements and parts of the hardware requirements have been developed together with people from the division on Inter City Trains and the division for Metro Trains, which has resulted in requirements relevant for trains, but also an increase in acceptance within the organization.

The application was successfully developed within time and budget constraints, which is considered to be quite unique for software development in general, considering that both the tool and parts of the hardware was entirely new. We have performed several demonstrations of the prototype, as well as held seminars on the methods and techniques we have used for the employees. This has raised quite a bit of interest among the employees.

As a conclusion in Pilot 1 we identified the development of a stable architecture for entire trains as the next step. In a way, parts of the vision contains the results from Pilot 1. Also, this architecture is an investment that must be placed before any trains are developed using UML, object-orientation, and the concept of a product-line architecture.

4. EXPERIENCES FROM PILOT 1

In this section we will present the results from the first pilot project. The results in this case is a set of experiences gained when working with the tool and the languages C++ and UML. However, the project also delivered considerations regarding non-technical matters. These are non-technical issues regarding the development organization and the educational efforts required due to the paradigm shift and the safety requirements.

4.1 UML and Rhapsody

The UML-tool Rhapsody in C++ (version 3.0) from I-Logix has been used for application development during Pilot 1. In this section we present our experiences from using UML and Rhapsody for train application development.

Our general impression of Rhapsody/UML is positive, but there are also aspects that we do not find that good. The main things we find positive about Rhapsody and UML are class diagrams, statecharts, automatic code-generation, and the possibility to automatically generate parts of the system documentation.

- Class diagrams play an important role when constructing the domain object model and also when designing the system. Developers can easily and rapidly express different solutions using UML. Even though people at the company mainly use a language similar to IEC 1131, they did not need that much training to understand class diagrams.
- Statecharts provide a powerful, yet easy to understand, way to express behavior of objects. Statecharts also enable behavior to be specified at a reasonable abstraction level. Compared to IEC 1131-based languages UML is a higher-level language. Another benefit is that state-machines are known by many, and the specifics of statecharts can easily be learned.
- If statecharts are used to specify the behavior of the system then Rhapsody can automatically generate the implementation, which considerably shortens implementation time. This also enables the design model and the implementation to be kept consistent; a strong benefit when it comes to maintenance.
- If the systems documentation can be automatically generated from the models, it is more likely that the documentation will stay consistant with the actual implementation. This is very important, not least with respect to the work of harmonizing the development process to the safety standard.

For the Pilot 1 application we used statecharts for the major part of the classes in the system, and we where therefore able to use automatic code-generation. Unfortunately, the application we developed in Pilot 1 did not have any strict time critical requirements and was focused on door control alone, we are therefore unable to judge how well Rhapsody could be used for entire trains. However, there are still a number of issues with Rhapsody that we consider need improvement, namely:

- State naming
- Code size
- Overhead

We explain these issues in detail below in Section 4.1.1 through Section 4.1.3.

4.1.1 State Names

When Rhapsody generates code for statecharts it uses the state names we have provided to name operations and classes. This enable us to perform code reviews quite efficiently. However, *and*-states are not required to be labeled. Should a developer forget to label an *and*-state then Rhapsody will automatically generate a label for it, such as *state_8*, which prohibits efficient reviews. (*And*-states, in contrast to *Or*-states, allows developers to express parallel activities in the UML. In an *Or*-state there can be only one active state, and *And*-states can have several active states.)

When considering the 20–30 years life-cycle of trains, it may at some point in time be the case that a bug should be fixed and that the Rhapsody tool is no longer available on the market. In such cases, it might be necessary to make the changes directly to the generated source code, and not the UML model in Rhapsody. Hence, the quality, in terms of maintainability, code size and performance, of the generated code is essential for success in the train application area.

4.1.2 Code Size

The first generated version of the application code required 899KB of memory, and still it only contained door control functionality. This is too large. Would it even possible to develop an entire train's functionality using Rhapsody and fit it within the 64Mb of memory available in the target system? Fortunately, the code size can be reduced by paying attention to Rhapsody's settings.

By changing some of Rhapsody's settings and changing the application to use *triggered operations* instead of *events*, we were able to reduce the code size to 315KB. The code size can be reduced even further by configuring Rhapsody's code generator. It is quite remarkable that each event costs about 2KB in code size. An equivalent application implementing similar functionality using current tools requires about 100KB also including extended fault handling.

Even though tools, in this case Rhapsody, allow developers to work at a reasonable high abstraction level, developers will be reluctant to use these tools if they cannot produce code that match the target systems constraints. In this case we are not certain that the "price" we pay in terms of increased code size is worth the benefits.

4.1.3 Overhead

A framework that supports execution of statecharts is delivered with Rhapsody. This framework takes care of, for example, event passing between state-machines.

For the Pilot 1 application we wanted to make as much use of Rhapsody's code generation features as possible, therefore we implemented most of the classes using statecharts. We even used statecharts for periodic activities, such as polling some sensors every 40 ms. During design we experienced the benefits of using this approach, e.g. reasonably high-level description of behavior using statecharts. As far as we have experienced, the overhead introduced by the framework is acceptable. However, we can not estimate how this scales to full train functionality.

When investigating the overhead further, we discovered that, in several cases, sending a single event is computationally equally expensive as the reaction to on an event in the application. The application might use events more than advisable, but anyhow, Rhapsody's framework should be improved in terms of overhead with using events.

In our opinion events should only be used for objects that execute infrequently and do not have strict performance requirements. Triggered operations, which is an alternative way of communicating between statecharts, require less computation time. A very rough estimate indicates that events require about 30 times the execution time of invoking a triggered operation.

The source code for the framework is delivered with Rhapsody, so anyone that want to improve it can do so. However, this is not desirable, since it will create a need to maintain the framework within the organization. This is not an attractive solution when considering that one of the main reasons for Bombardier Transportation for looking at third party tools is to avoid tool maintenance within the organization.

A way to reduce the overhead is to use ordinary C++ methods instead of events/triggered operations. However, this would make it impossible to use statecharts and the code generation feature of Rhapsody, which we see as the major benefits of the tool. However, we find the extra overhead introduced by invoking triggered operations acceptable.

4.2 Safety

When it comes to safety and software, most of the efforts required in IEC 61508 affects the development process. Not that much is mentioned about actual programming paradigms and software constructions. For instance, only by using a modeling language and a, as homogeneous tool suit as possible, a step toward SIL classification is taken. The use of semi-formal methods is promoted for the SIL targeted in this case, which is provided by the UML statecharts. The statecharts were proven to be helpful when reasoning about the design. Quite a few errors where found and removed, which would have been much harder to find in pure C++ source code.

However, it is important that the code generated from the statecharts is efficient enough, else they will not be used. As a semi-formal language they provide a good base for informal reasoning. Also, in the favor of this technique is the fact that the implementation will indeed perform the modeled behavior since the models, in a way, also is the implementation. Thus, the semantical gap between implementation and model is over-bridged leading to an increased confidence in that the implementation is executional-wise equivalent with the models.

The use of a code generator may seem a little bit provocative in the safety community. Nevertheless, we argue that it is fully possible to make use of such methods. The very same concerns were raised when moving from assembly languages to "high-level" compiled languages such as C. We may think of UML and code generators as yet another level of abstraction.

Working on the level of abstraction provided by UML and its statecharts helped us keep some of the complexity imposed by a programming language such as C++, away. This is not least important when performing reviews of the implementation. In our case, we used the UML statecharts for design reviews and the actual source code produced by the generator were used when reviewing the implementation. The code reviews were performed on this level of abstraction mainly for the reason of evaluating the quality of generated code. We found the code to be quite good as it was indeed readable and, as far as we could see, implemented the modeled behavior correctly. We think that if these tools become "proven-in-use", which is somewhat recommended in IEC 61508, the review will mainly concern the models.

On the other hand, the code generated by Rhapsody heavily depends on the provided framework. We may very well treat the framework as an reusable component but it must be fully verified according to the requirements in the safety standard before being used in a safety related application.

A very important issue when it comes to safety is the handling of requirements. Requirement must be traceble all the way through the models and down to the implementation. UML is rather limited when it comes to requirements. The concept of use-cases is not very useful for specifying non-functional requirements as they focus on the functionality as perceived from a user's point of view. Consequently, the need for additional tools that manage requirements, and smoothly integrates with the modeling tool, was identified by the pilot project.

4.3 General technical findings

The pilot project was not confronted with any bigger challenges regarding *configuration management* (CM). We used Visual Source Safe (VSS), from Microsoft which could be integrated in the Rhapsody tool. We are aware of the shortcomings in VSS and its possibilities was not investigated as such, it was only used for keeping the pilot's artifacts consistent in a rather uncomplicated fashion.

Moving towards reuse oriented development based on product-line architectures will, however, dramatically increase the complexity regarding CM. Especially if the reuse program is implemented across the geographically distributed sites within the company, the demands on CM gets even higher. This issue was not at all considered in the first pilot project. Nevertheless, it is an ever so important subject that must be carefully dealt with by the company.

4.4 Non-technical results

So far we have discussed experiences regarding technological changes. However, the project also identified and brought up changes needed in the organization itself. These changes arise mainly from two sources, *SIL classification* and *software reuse*. Moreover, the way in which the company is doing business will be affected

4.4.1 Organization

When it comes to safety, processes and project organizations are restricted by the standard. Developing SIL-classified software is mainly a matter of producing the software according to the standard. For instance, the roles needed in a project, the competence of project members, the required documents, etc., is regulated in those standards. In our opinion is the assumption indirectly postulated in IEC 61508 true, that a solid organization and development process delivers solid software.

The introduction of software product-lines and software reuse is also most likely to affect a company's organization [10][4]. Typically should the owner and the developer of the reusable assets, i.e. components and architecture, be somehow separated organizational from the reusers. Exactly how such a company should be organized in order to achieve successful and profitable reuse is dependent on the current structure. Parameters that influence the reuse-organization are typically, the size of the company, the number of products and the geographical distribution of working sites.

The reusers are engineers developing applications based on reusable software and project specific functionality. Moreover, a third organizational unit may be required taking care of maintenance of legacy systems. Unfortunally, such an organization may give engineers the impression that they are divided into an *A-team* and a *B-team* where A-team members are working with new, exciting technology and the B-team mainly takes care of correcting bugs and maintaining old system using old technology. In Bombardier Transportation's business, where the lifetime of products are in the range of 20–30 years, this is an extremely important matter. Most engineers get motivated from new technology and new challenges and there may be a possible risk of employees leaving the company if they feel stuck in such a B-team situation. Moreover, recruiting new staff to the maintenance organization may be difficult. The solution to this problem is not obvious. Nevertheless, the problem must be dealt with.

4.4.2 Education

Shifting technology paradigm also implies an investment in education for software engineers. This may not be so controversial since in the current situation, newly employed engineers must be trained before they can use the in-house developed tools. In the long term, this need will be reduced even further since today universities usually teach both UML and object-orientation in their basic courses in computer engineering.

However, the competence profile in the company may change due to the introduction of object-orientation. Currently engineers at the company are slightly oriented towards electrical engineering, working with "signal-oriented" function-blocks that are quite close to electrical schemas. The signal-oriented view also harmonizes well with the traditional electrical view of train implementations. Object-orientation may potentially deteriorate that intuitive system knowledge. Consequently, in the long run there may be a need for changing the internal training from software related to system- and electrical related in order to preserve the feeling for the system among software engineers.

5. CONCLUSION

In this paper we have reported our experiences from an ongoing investigation of whether Bombardier Transportation should switch from a function-block oriented development to

an object-, and reuse-oriented approach. As part of this investigation we also considered emerging safety requirements on train applications and their implications on the supporting tools. Currently are train applications constructed using an in-house developed tool that is similar to IEC 1131. As the current tool is not likely to meet future requirements on support for reuse, we have also investigated commercially available software tools.

The investigation has, up until now, been performed as a pilot project. The purpose of the pilot was not only to determine the technical feasibility of using UML and object-orientation for train application development, but also to get acceptance from the organization. By involving observers, a reference group, and by conducting workshops that all employees interested in the work could attend, we have established acceptance for the new technologies. The observers were applications engineers with no or little knowledge about object-orientation and UML. The reference group mainly consisted of managers from different development departments. However, it is not sufficient to run only a single and short pilot to decide whether to perform a paradigm shift that is initially very expensive. The issue has to be further investigated in order to make sure that the methodology can indeed be adopted by the complete organization. In our opinion are pilots an appropriate line of actions when working with acceptance, but it requires lengthy and resolute work.

For application development we have evaluated the UML-supporting tool Rhapsody and found both pros and cons. Our impression is that the real strengths of the tool lies in the design phase. Also, by using automatic code generation from models made using state-machines, which is supported by Rhapsody, the implementation phase can be significantly shortened. The semi-formalism provided by state-machines is also beneficial and endorsed by safety standards. However, if the code generated from state-machines is not efficient enough they will not be used. Thus, the efficency of the generated code is not only important with respect to performace of the application, it will also be of vital importance for the safety aspects.

The generated code exhibited some performance overhead due to the framework supporting the execution of state-machines. However, the generated performance overhead is acceptable as long as complexity in the software systems is managed, and the cost of developing it is decreased, due to the new method. Worth mentioning on the counter-side is that the resulting executable consumes too much memory in respect to the complexity of the implemented application. This must be further investigated.

If the proposed changes are implemented, it will also have great impact on the organization, mainly due to issues regarding reuse based development and safety. However, if successfully implemented, reuse will return in form of reduced development costs and increased software quality.

One of the core findings from the first pilot is that the design of a solid architecture is crucial. As a consequence there is a plan to run an investigation focusing on the generic system architecture for future train applications.

Acknowledgement

We would like to thank Bombardier Transportation for giving us the opportunity to take part in the investigation and for granting the publication of these results.

6. REFERENCES

[1] Bombardier Transportation web-page. `http://www.transportation.bombardier.com`.

[2] J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

[3] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying Software Product-Line Architecture. *IEEE Software*, 30(8):49–55, August 1996.

[4] D. Fafchamps. Organizational Factors and Reuse. *IEEE Software*, 11(5):31–41, September 1994.

[5] J. Favaro. What Price Reusability?: A Case Study. In *Proc. ACM First Symposium on Environments and tools for Ada*, pages 115–124, 1990.

[6] J. J. Fernandes, R. J. Machado, and H. D. Santos. Modeling Industrial Embedded Systems with UML. In *Proc. Eighth International workshop on Hardware/software codesign*, pages 18–22, 2000.

[7] I-Logix web-page. `http://www.ilogix.com`.

[8] Application and Implementation of IEC 1131-3, May 1995. Standard provided by the International Electrotechnical Commision.

[9] IEC 61508 - Functional Safety of electrical/electronic/programmable safety related systems. Standard provided by the International Electrotechnical Commision.

[10] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse - Architecture, Process and Organization for Business Sucess*. Addison-Wesley, 1997.

[11] M. Mrva. Reuse Factors in Embedded Systems Design. Technical report, High-Level Design Techniques Dept. at Siemens AG, Munich, Germany, 1997.

[12] E. H. Shokri and K. S. Tso. Ada95 Object-Oriented and Real-Time Support for Development of Software Fault Tolerance Resusable Components. In *Proc. Second International Workshop on Object-oriented Real-time Dependeble Systems*, pages 93–100, 1996.

[13] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.