Software Systems In-House Integration Strategies: Merge or Retire – Experiences from Industry

Rikard Land[*], Laurens Blankers[#], Stig Larsson[*], Ivica Crnkovic[*]

[*]*Mälardalen University, Department of Computer Science and Electronics*
PO Box 883, SE-721 23 Västerås, Sweden
{rikard.land, stig.larsson, ivica.crnkovic}@mdh.se
[#]*Eindhoven University of Technology, Department of Mathematics and Computing Science*
PO Box 513, 5600 MB Eindhoven, Netherlands
l.blankers@student.tue.nl

ABSTRACT

When an organization faces different types of collaboration, for example after a company merger, there is a need to consolidate the existing in-house developed software. A main challenge is to select a suitable strategy, such as merging the systems, evolve one of the existing systems to be able to retire others, or start a new development effort in order to retire the existing systems. This should arguably be done at a high abstraction level, i.e. architectural level. In order to investigate how a strategy should be chosen, we have performed a multiple case study, consisting of nine integration projects. Two major concerns have been found that can be used to exclude some strategies: 1) architectural compatibility, and 2) what we have labeled 'retireability', i.e. all considerations influencing whether or not the existing systems can be allowed to be retired.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering.*

K.6.3 [**Management of Computer and Information Systems**]: Software Management – *Software maintenance.*

General Terms

Design, Economics, Human Factors, Management.

Keywords

Architectural Compatibility, Architectural Concern, Case Study, Integration Strategy, Software Integration, Software Merge, Retireability.

1. INTRODUCTION

Organizations have spent large sums of money on development of software systems as part of their core business and want to capitalize on their investment. At the same time reorganizations and mergers force the organizations to integrate their software systems. This leads to a variety of problems such as functional overlap and architectural and platform incompatibility.

The software may be the core products of the companies, or a support systems for the core business. If the software systems are mainly used in-house, performing further evolution and maintenance of two systems in parallel seems wasteful. If the software systems are products of the company, it makes little sense to offer customers two similar products. In either case, the organization would ideally want to take the best out of the existing systems and integrate or merge them with as little effort as possible.

The available published experience on integration does not directly address this context, where the existing systems are developed separately but now completely controlled within a single organization. We have chosen to label integration in this context *in-house integration* in contrast to other types of integration found in literature: integrating third-party components or systems, and interoperability based on standards in open systems. These approaches might be applicable also in the situation when an organization has full control over the development and evolution of the systems to integrate, but there are other options as well, such as modifying arbitrary parts of the existing systems in order to be able to merge them, starting a new development effort based on the previous experience, or selecting one of the existing systems and evolve it in order to replace all existing systems. To collect all the existing experience from organizations that have faced this challenge we have performed a multiple case study, consisting of nine integration projects in six organizations.

Based on the case study we present four *integration strategies* and two *concerns* that are crucial to address when selecting a strategy, which we have labeled *compatibility* and (lacking a better term) *retireability*. The audience this paper aims for consists of both researchers and industrial architects.

The paper is organized as follows: section 2 describes the methodology. Section 3 presents a model of four integration

strategies which is used to classify the cases, which are described in section 4. The cases are further analyzed in section 5 in terms of strategies and concerns. Section 6 presents related work, and section 7 concludes the paper.

2. RESEARCH METHODOLOGY

The multiple case study [37] consists of nine cases from six organizations that have gone through an integration process. Our main data source has been interviews. To collect the data, people willing to participate in the interviews were found through personal contacts. The interviews were to be held with persons in the organization who:

1. Had been in the organization and participated in the integration project long enough to know the history first-hand.
2. Had some sort of leading position, with first-hand insight into on what grounds decisions were made.
3. Is a technician, and had knowledge about the technical solutions considered and chosen.

All interviewees fulfilled either criteria 1 and 2 (project leaders with less insight into technology), or 1 and 3 (technical experts with less insight into the decisions made). In all cases, people and documentation complemented each other so that all three criteria are satisfactory fulfilled. There are guidelines on how to carry out interviews in order to e.g. not asking leading questions [25], which we have strived to follow. The questions were open-ended, focused around architecture and processes, and the copied out interview notes were sent back to the interviewees for feedback and approval. The interviewees were asked to describe their experiences in their own words, as opposed to answering questions, however we used a set of open-ended questions, to ensure we got information about the history of the systems, problems, solutions, and more. The questions are reprinted in the Appendix. Due to space limitations the answers are not reprinted. They can be found however in a technical report [17], together with further details regarding the research design. In some cases, the interviewees offered documents of different kinds (system documentation as well as project documentation), which in our opinion was mostly useful only to confirm the interviewees' narratives. In one case (F1) one of the authors (R.L.) participated as an active member during two periods (fall 2002 and winter 2004-2005).

The research can be considered to be *grounded theory* [29] in the sense that we collected data to build models for previously un-researched questions, as contrasted to validating a pre-defined hypothesis. Strictly, there is no external validity in the traditional sense (of course the data fits the model, because the model is built from the data); validation would mean repeating the case study and comparing the current model with the new data. On the other hand, the new data would probably be used to modify or refine the model, leading to the same validity problem. It has been argued that for grounded theory research, the validation that can be achieved is a proper *understanding*, which can only be judged by other researchers [19,29].

We have deliberately avoided commenting the outcome of the cases as being good or bad, as the criteria as to how to do this are not at all obvious and are practically difficult to determine. Problems in answering this question include: how many years need to pass before all effects of the integration are known? How can the quality of the resulting systems be evaluated, if at all? Or is the competitiveness and financial situation of the company a certain number of years a more interesting measure? And by making case studies, it is impossible to know what the result of some other choice would have been. All value statements therefore come from the interviewees themselves, based on their opinions based on their perception of e.g. whether time and money was gained or spoiled.

3. STRATEGIES

In order to classify the decisions made in the cases we present a model dividing the numerous possibilities for integration into four strategies that are easily understood analytically: *No Integration*, *Start from Scratch*, *Choose One*, and *Merge*. In reality, we can expect that these strategies are not strictly independent; some real cases can be seen as a combination of two, or as something in between. However we find it useful to use this model as a framework for discussion. The strategies, especially the *Merge* strategy, are discussed in more detail in section 5.1.

No Integration (NI) Develop existing software systems in parallel, which clearly will not result in an integrated or common system. However it is mentioned for the sake of completeness.

Start from Scratch (SFS) Start the development of a new system, aimed to replace the existing systems, and plan for discontinuing the existing systems. In most cases (parts of) requirements and architecture of the existing systems will be carried over to the new system. This strategy can be implemented by buying a commercial solution or building the new system.

Choose One (CO) Evaluate the existing systems and choose the one that is most satisfactory, officially discontinue development of all others and continue development of the selected system. It may be necessary to evolve the chosen system before it can fully replace the other systems.

Merge (M) Take parts from the existing systems and integrate them to form a new system that has the strengths of both and the weaknesses of none. This is, of course, a very idealized strategy and as it will turn out the most complicated and broad strategy of the model.

4. THE CASES

The cases come from different types and sizes of organizations operating in different domains, the size of the systems range from a maintenance and development staff of a few people to several hundred people, and the demands on extra-functional requirements are very different depending on the system domain. What the cases have in common though is that the systems have a significant history of development and maintenance.

The cases are summarized in Table 1. They are labeled A, B, etc. Cases E1, E2, F1, F2, and F3 occurred within the same organizations (E and F). For the data sources, the acronyms used are I_X for interviews, D_X for documents, and P_X for participation, where X is the case name (as e.g. in I_A, the interview of case A), plus an optional lower case letter when several sources exist for a case (as e.g. for interview I_{Da}, one of the interviews for case D). I_X:n refers to the answer to question n in interview I_X. In this paper, we have provided explicit pointers into this source of data. This paper focuses on architectural issues; for processes the reader is referred to [18].

Table 1: Summary of the cases.

	Organization	System Domain	Goal	Information Resources
A	Newly merged international company	Safety-critical systems with embedded software	New HMI* platform to be used for many products	*Interview:* project leader for "next generation" development project (I_A)
B	Organization within large international enterprise	Administration of stock keeping	Rationalizing two systems within corporation with similar purpose	*Interview:* experienced manager and developer (I_B)
C	Newly merged international company	Safety-critical systems with embedded software	Rationalizing two core products into one	*Interviews:* leader for a small group evaluating integration alternatives (I_{Ca}); main architect of one of the systems (I_{Cb})
D	Newly merged international company	Off-line management of power distribution systems	Reusing HMI* for Data-Intensive Server	*Interviews:* architects/developers (I_{Da}, I_{Db}).
E1	Cooperation defense research institute and industry	Off-line physics simulation	Creating next generation simulation models from today's	*Interview:* project leader and main interface developer (I_{E1}) *Document:* protocol from startup meeting (D_{E1})
E2	Different parts of Swedish defense	Off-line physics simulation	Possible rationalization of three simulation systems with similar purpose	*Interview:* project leader and developer (I_{E2}) *Documents:* evaluation of existing simulation systems (D_{E2a}); other documentation (D_{E2b}, D_{E2c}, D_{E2d}, D_{E2e}, D_{E2f})
F1	Newly merged international company	Managing off-line physics simulations	Possible rationalization by using one single system	*Participation:* 2002 (R.L.) (P_{F1a}); currently (R.L.) (P_{F1b}). *Interviews:* architects/developers (I_{F1a}, I_{F1b}); QA responsible (I_{F1c}) *Documentation:* research papers (D_{F1a}); project documentation (D_{F1b})
F2	Newly merged international company	Off-line physics simulation	Improving the current state at two sites	*Interviews:* software engineers (I_{F2a}, I_{F2b}, I_{F2f}); project manager (I_{F2c}); physics experts (I_{F2d}, I_{F2e})
F3	Newly merged international company	Software issue reporting	Possible rationalization by using one single system	*Interview:* project leader and main implementer (I_{F3}) *Documentation:* miscellaneous related (D_{F3a}, D_{F3b})

The remainder of this section describes the cases in some more detail illustrating how strategies were considered and how one was chosen. Due to space limitations this is done in a very brief manner.

Case A Each of the previous separate companies had developed software human-machine interfaces (HMIs) for their large hardware products (I_A:1,2). To rationalize, it was decided that a single HMI should be used throughout the company (I_A:2,3), that is, there was a wish to discontinue at least all but one of the existing HMIs. The two existing systems with the highest influence had a very different underlying platform: one was based on open source platforms and the other on commercial solutions (I_A:1,2,8). The differences were maybe largest when it came to the cultural clash associated with the platforms excluding the possibility of a *Merge*. As resource constraints were not a major influence, the decisive factor when choosing between the remaining strategies was the new consolidated set of requirements, especially larger configurability of the system, and the availability of new technology (I_A:3,5). Therefore, *Start from Scratch* was desired by the architects over *Choose One* and thus selected (I_A:7,8).

Case B Two existing systems with similar functionality had to be merged in order to reduce cost (I_B:3). One system was used throughout the company the other only in one daughter-company (I_B:1). Discontinuing of the large system was not considered thereby excluding *Start from Scratch*. The smaller system was built on the tight integration paradigm, while the large system was build as a loose integration of many subsystems (I_B:1,6,7,13). The difference in approach made *Merg*ing the systems infeasible, therefore the *Choose One* strategy was chosen (I_B:7).

Case C The systems and the development staff of the organization is the largest among the cases: several MLOC and hundreds of developers (I_{Cb}:1,9). Two such systems with very similar functionality were being developed within the now merged company and both were nearing release (I_{Ca}:1, I_{Cb}:1). Management did not want to retire either of them, but wanted the best parts of both systems to form the new system within six months, i.e. *Merge* (I_{Ca}:6, I_{Cb}:6). The systems were similar in

* HMI=Human-Machine Interface

many ways (I_{Ca}:7, I_{Cb}:1), but there were differences as well: some technology choices, framework mechanisms such as failover, supporting different (natural) languages, and error handling, as well as a fundamental difference between providing an object model or being functionally oriented (I_{Cb}:1,6,7). The differences prevented *Merg*ing the systems in a short period of time, but allowed for *Merg*ing over a longer period. The architects reported to management that the *Merge* would take 2 years, much longer than desired, and that a better option was to *Choose One* (I_{Ca}:6, I_{Cb}:6). Eventually management changed its position so that either (but not both) could be discontinued, allowing for the *Choose One* strategy which was implemented (I_{Ca}:6). However the decision was not made until both systems were independently released and deployed at customers. This caused an estimated loss of one year of development effort of a team of several hundred developers (I_{Cb}:6), confusion to the customers who did not know which of the two products to choose, and required addition effort in migrating the customers of the retired system to the chosen one (I_{Ca}:6). One of the interviewees points out that although the process seems suboptimal it is difficult to say whether other approaches would have been more successful (I_{Ca}:12).

Case D The two systems, both consisting of an HMI and a server, have a common ancestry, but have evolved independently for 20 years (I_{Da}:1). Five years before the merger the HMI of one of the systems was decomposed into components and significantly modernized (I_{Db}:7). The other system built on more aged technologies, and around the time of the merger the customers of this system considered that HMI to be outdated; it was therefore decided that the dated HMI should be replaced by the modern one, thus *Choose One* (I_{Da}:1, I_{Db}:3). In order to do this the server that used to be controlled by the dated HMI had to be changed, but thanks to the common ancestry it was relatively easy to make the same modifications to the server that had been made five years ago when the modern HMI was developed (I_{Db}:8). The servers themselves have not been integrated yet; they both implement the same industry standards and the plans are to perform a gradual *Merge* but there are no concrete plans (I_{Da}:1, I_{Db}:6). In the summary and analysis of the cases we will therefore discuss the integration of HMIs and servers separately.

Case E1 The goal in this cooperation project was not only to integrate several existing systems, but also to add another, higher level of functionality (I_{E1}:1,3). Retiring the existing systems was possible since all parties would benefit from the new system (I_{E1}:1). Many of the existing systems (but not all) were written in the same language (I_{E1}:6). However, a new language was considered better suited for a higher level of system complexity and would also bring a number of additional benefits such as reusable code, robustness, commonality within the organization (I_{E1}:6,7). Thus *Start from Scratch* strategy was chosen (I_{E1}:6).

Case E2 A certain functional overlap among three simulation systems was identified (I_{E2}:1, D_{E2a}). The possibility of retiring any, but not all, of these systems was explicitly left open, partly because of limited resources and partly because (some of) the functionality was available in the others (D_{E2a}, I_{E2}:13). Two systems were somewhat compatible, but due to limited resources the only integration has been reuse of the graphical user interface of one into the other, although this was more complicated than anticipated (I_{E2}:6). We thus have some reuse but no *Merge*, as there are no resources and no concrete plans for integrating these two systems into one. Although not directly replaced by the others, the third system has in practice been retired (I_{E2}:6,13) and we consider this case to be a *Choose One* strategy (actually *Choose Two* out of three).

Case F1 After the company merger, there has been a need to improve and consolidate management and support for certain physics simulations with data management mechanisms and user interfaces (I_{F1a}:1, I_{F1b}:1, I_{F1c}:1,2, D_{F1a}, P_{F1a}, P_{F1b}). Initially, three systems were considered for integration or replacement; the two possibilities outlined were a tight merge with a result somewhere between *Merge* and *Choose One* and a loose integration in an *Merge* manner, which became the decision (I_{F1a}:3, I_{F1c}:3, P_{F1a}, D_{F1a}). However, the many incompatibilities indicated a very long development time. It was not considered possible to discontinue development on the existing systems before a full replacement was available and the limited resources and other local priorities in practice resulted in no progress towards a common system (I_{F1c}:6, D_{F1a}, P_{F1a}, P_{F1b}). Currently, stakeholders are favoring *Choose One*, but the scope is unclear and integration activities still have a low priority (I_{F1a}:1,6, I_{F1b}:6,9, I_{F1c}:1, P_{F1b}); some participants have seriously begun to question the value of integration altogether (I_{F1b}:3,9, I_{F1c}:6,9) and the result so far, after four years, has been *No Integration*. This apparent lack of results is not due to lack of will or effort, because throughout these years there have been numerous attempts to identify a proper integration strategy (I_{F1a}:3,6, I_{F1b}:9,11, I_{F1c}:6, P_{F1b}).

Case F2 The two systems both consist of four programs run in sequence, with very similar roles, communicating via input and output files: pre-processor, 2D simulator, post-processor, and 3D simulator (I_{F2a}:1,9, I_{F2b}:7, I_{F2c}:10,11, I_{F2d}:8, I_{F2e}:5, I_{F2f}:8). To create a common system, it was considered possible to discontinue the first three parts, as long as there is a satisfactory replacement, although the simulators need to be validated which makes time to release longer (I_{F1c}:6, I_{F1f}:6). The 3D simulator is considered very large and complex to replace, so this is not realistic within the next few years (I_{F1f}:6). So far there are common pre- and post-processors; they have been rewritten, i.e. *Start from Scratch*, although the post-processor started as an attempt to *Choose One* of the post-processors and evolve it, but due to insufficient analysis of requirements it had to be almost completely rewritten (I_{F2a}:9, I_{F2b}:1,7, I_{F2c}:7,9, I_{F2d}:6,7, I_{F2e}:7). Although the 2D simulators are branched from a common ancestor, they are no longer very similar, and one of the 2D simulators is currently being evolved to replace the other, i.e. *Choose One* (I_{F2a}:1,9, I_{F2b}:7, I_{F2d}:7,8). Parts of the 3D simulators have been re-developed commonly, i.e. *Merge* (I_{F2a}:3, I_{F2c}:3, I_{F2d}:7, I_{F2e}:7,8, I_{F2f}:3,6,7). Due to the different states and choices of integration of the four parts we will treat them separately.

Case F3 Three different software systems for tracking software issues (errors, requests for new functionality etc.) were used at three different sites within the company, two developed in-house and one being a ten-year old version of a commercial system (I_{F3}:1). The two systems developed in-house where somewhat compatible (I_{F3}:1). All involved saw a value in a common system supporting the best processes within the company, and apart from the fact that a transition to such a common system would mean some disruption at each site, independently of whether the common system would be completely new or a major evolution of the current system used there was no reluctance to the change (I_{F3}:3,10,11). Being a mature domain, outside of the company's core business, and realizing the effort required to creating a new issue tracking system from scratch themselves, the decision was to *Start from Scratch* by acquiring a commercial system (I_{F3}:6).

5. ANALYSIS

Many aspects affect the choice of strategy, which will be discussed in this section.

5.1 Refining the Model

In section 3 we presented a model to assist us in the discussion of the cases. Based on the cases we extend the *Merge* strategy, introduce two concerns that aid exclusion of strategies, and discuss influences on the selection of a final strategy.

5.1.1 Subdividing the Merge Strategy

To aid the discussion, we first present two types of *Merge*, labeled *Instant* and *Evolutionary*, only distinguished by their associated time scale. By introducing them, some events in the cases and some conclusions are more easily explained, although there is no strict borderline between them. These two types of *Merge* should not be understood as two strategies with distinct identities in the same sense as the four originally presented.

Instant Merge (IM) With "instant" we mean that the existing components can be rearranged with little effort, i.e. basically without modification or development of adapters. How to evaluate and select components is the responsibility of the architect. This strategy was desired in case C, but could not be implemented.

Evolutionary Merge (EM) Continue development of all existing systems towards a state in which architecture and most components are identical or compatible, in order to allow for *Instant Merge* sometime in the future. The architects in case C asserted that if a *Merge* was desired an *Evolutionary Merge* was the only possibility. Case F2 clearly demonstrates this strategy. Indications are that case D_{Server} will also follow this path in the future.

5.1.2 Concerns

Strategy selection in the cases was influenced by many factors of many different kinds, including the current state of the existing systems, both technically and from a management perspective, both in themselves and in relation to each other; the level of satisfaction with existing systems, among users, customers, and the development organization; the completeness or scope of the existing systems with respect to some desirable set of features; development resources; desired time to market; the impact of retiring any or all of the systems. To suggest a systematic procedure for selecting a strategy, one starting point would be to identify issues that would not only suggest one or more strategies as appropriate, but also exclude some strategies as inappropriate. Of the concerns listed, we identified only two of these as being able to, when properly addressed, exclude strategies: the architectural compatibility of the systems and the retireability of the systems.

Compatibility Architectural mismatch can make integration hard if not impossible [9]. In order for (parts of) systems to be integrated they therefore need to be compatible to some extent. Also when systems are not based on components or clearly defined interfaces, differences in the framework used can also have a negative impact on compatibility. Ideally systems are compatible to such an extent that it is possible to pick the best components from both resulting in an *Instant Merge*. However this situation has not been observed among the cases even though some systems share a common ancestry (I_{Db}:1, I_{F2a}:1) or are based on common standards (I_{Cb}:1). In reality systems may be somewhat compatible possibly allowing for an *Evolutionary Merge*, but there are also the options to also *Choose One* or *Start from Scratch*. If systems are totally incompatible neither *Evolutionary Merge* nor *Instant Merge* is possible. We want to emphasize that compatibility is a greyscale and there is no universal compatibility measure. Similar structures seem to be a necessary condition as well as similar in the sense of environment that defines components [16]. What compatibility may mean in every new situation must be evaluated by the architect.

Retireability Stakeholders may consider retiring a system unfeasible for various reasons, such as market considerations, user satisfaction, or potentially the loss of essential functionality. If all systems can be retired, all integration strategies are possible. If not all systems can be retired, it is not possible to *Start from Scratch* because this would require discontinuing all systems. If none of the existing systems can be discontinued, both of the integration strategies *Choose One* and *Start from Scratch* are excluded.

Table 2 summarizes the exclusion of strategies based on these concerns, where black denotes exclusion of a strategy. Although compatibility is a continuous scale, for the sake of discussion it is divided into three classes: High, Modest, and None. "High" means that the systems are compatible to such an extend that components can be picked from either one and combined into a new system with very little modification, "None" means that the systems are fundamentally different, and "Modest" is somewhere in between. Retireability is divided into All, Not all, and None. "All" means that it is possible to retire all systems, "Not all" means that out of the two or more systems one or more systems can be retired, but at least one can not, and "None" means that none of the systems can be retired.

We note that evaluating compatibility mainly involves finding information and facts about the current state of the systems, which is not the case for statements about retireability. To what extent a system is retireable is not solely determined by the architect, but involves the opinions of other stakeholders such as management, users, and marketing. This also means that retireability is not a static property, but that it can be renegotiated with involved stakeholders. This is especially true when this concern excludes a strategy that is for other reasons considered desirable.

Although many influences on the decisions made were found in the cases, these two are the only ones that result in the exclusion of strategies. It appears as other influences, such as satisfaction with the existing systems, scope, available development resources, and availability of commercial products, can not be considered in isolation to exclude some strategies, but taken together they can motivate the choice of one strategy over another, and also influence retireability considerations. General influences found are discussed in section 5.3.3.

Table 2: The exclusion of possible strategies based on concerns (black indicates exclusion)

		SFS	CO	EM	IM
Compatibility	High				
	Modest				■
	None			■	■
Retireability	All				
	Not all	■			
	None	■	■		

Table 3: Concerns per case
(question mark indicates that the information was not available)

	Retireability	Compatibility
A	All	None
B	Not all	None
C (initial)	None	Modest
C (final)	Not all	
D_{HMI}	Not all	None
***D_{Server}**	(?)	Modest (?)
E1	All	Modest
E2	Not all	Modest
F1 (initial)	No	None
F1 (final)		
$F2_{Pre}$	All	None/Modest (?)
***$F2_{2D}$**	All	Modest
$F2_{Post}$	All	None
***$F2_{3D}$**	None	Modest
F3	All	None/Modest (?)

Table 4: Possible and desired strategies, per case
(black indicates exclusion, circle indicates selected strategy)

	SFS	CO	EM	IM
A	●		■	■
B	■	●	■	■
C (initial)	■	■		●
C (final)	■	●		■
D_{HMI}	■	●	■	■
***D_{Server}**	(?)	(?)	●	(?)
E1	●			■
E2	■	●		■
F1 (initial)	■	■	●	■
F1 (final)	■	●	■	■
$F2_{Pre}$	●		(?)	■
***$F2_{2D}$**		●		■
$F2_{Post}$	●		■	■
***$F2_{3D}$**	■	■	●	■
F3	●		(?)	■

5.2 Revisiting the Cases

Table 3 summarizes the concerns for all cases. Table 4 combines tables 2 and 3 by showing which strategies are excluded, marked with black, based on the concerns, which was desired and (where applicable) eventually chosen, which is indicated with a circle. Three rows in the tables (D_{Server}, $F2_{2D}$, and $F2_{3D}$) are marked with an asterisk ("*"), indicating that the systems are not yet integrated and the information summarized is preliminary. A question mark indicates that the classification is unsure, but we have chosen to show the interpretation that could falsify our proposed scheme, i.e. excluding the most strategies. In case C retireability was clearly renegotiated, and in case C and F1 the decision changed, illustrated in Table 4 with multiple entries for these cases showing these iterations.

5.3 Strategy Exclusion and Selection

In section 5.1.2 we presented two important concerns for selecting a strategy and proposed that these concerns can exclude the selection of certain strategies (e.g. if systems are not compatible they can not be instantly merged). Table 4 shows that most cases have selected a strategy that, according to this model, is not excluded. In these cases integration was successful or is making progress. There are three rows (C', F1', and F1'') where a strategy was desired that was excluded, and there have in fact been significant problems due to that: in case C the decision had to be changed, and in case F1 all alternatives are still excluded, and no significant progress has been made. Thus, two cases directly support our premise that the concerns compatibility and retireability exclude certain strategies, and the other cases do not contradict it.

The rest of this section describes observations concerning architectural compatibility, followed by a number of influences to retireability and the final choice of strategy.

5.3.1 Compatibility

Compatibility, unlike *retireability*, is not a concern that can be negotiated or modified because it is a static property of a collection of systems. Given an assessment of architectural compatibility, it cannot be changed only because it gives a dissatisfactory answer. There are two things however that can be done to improve the compatibility in order to make a merge possible. First, if a subset of the candidate systems (or some subsystems) is considered compatible it may be possible to change the scope of the integration project to include only these subsystems, thus enabling the possibility of a *Merge*. Case F1 exemplifies a change in scope, but unfortunately no suitable set of systems to merge have been found (I_{F1a}:1, I_{F1b}:1,6, I_{F1c}:1, P_{F1a}, P_{F1b}). Second, it is possible to evolve one or all systems towards a state in which they are compatible, i.e. performing an *Evolutionary Merge*. However, given the time required, some other strategy may be considered preferable, as shown by case C (I_{Ca}:6, I_{Cb}:6).

A definition of architectural (in-)compatibility would be subject to the same semi-philosophical arguments as definitions of architecture, and we will not attempt to provide one. Exactly what aspects of compatibility are the most important to evaluate will arguably differ for each new case. Some observations from the cases are provided in the following, which can be a complement to other reports of architectural incompatibility [9].

Similar high-level structures seem to be a pre-requisite for a *Merge*, i.e. if there are components with similar roles in the existing systems. In case D, both systems consisted of an HMI and server, which made it possible to reuse the HMI from one system into the other. In case E2 two of the existing systems consisted of a graphical user interface (GUI) and a simulation engine, loosely coupled, which made reuse of the GUI possible. In case F2, the two existing pipe-and-filter structures were strikingly similar. Reuse of components and architectural solutions into a common system in the cases is analyzed in depth elsewhere [16].

Similarity of frameworks could also be one measure of compatibility. In case F2, the framework can be said to describe separate programs communicating via input and output files. Two of the existing systems in case F3 were developed in Lotus Notes, and they were, with the words of the interviewee, "surprisingly

similar" (I_{F3}:1). In case C, the hardware topology and communication standards define one kind of framework.
One common source of incompatibility in systems is differences in the data model. Both syntactical and semantically differences can require vast changes in order to make system compatible. This has been a recurring problem in case F1 (I_{F1a}:6; D_{F1a}, P_{F1a}, P_{F1b}). In case F2, a new data model was defined and the existing systems adapted (I_{F2e}:7, I_{F2f}:6). In case F3, the three existing systems all implemented similar workflows, however the phases were different (I_{F3}:3).
Some systems in the cases shared a common ancestry (cases D and F2) or were based on common standards (C and D), but in no case were the systems compatible enough to allow for an *Instant Merge*. This indicates that these factors in themselves do not guarantee total compatibility.
As compatibility is not re-negotiable, and has such profound impact on the possible integration strategies, it must be carefully evaluated and communicated prior to a decision. Obvious as this may sound, the cases illustrate that this is not always the case. In case C, management insisted on an *Instant Merge*, although considered impossible by the architects (I_{Ca}:6, I_{Cb}:6) resulting in several hundred person-years being lost. In case F1 an *Evolutionary Merge* was decided upon because the systems could not be retired, even though the systems were incompatible (I_{F1c}:6, D_{F1a}, P_{F1a}), resulting in no progress after 4 years of work. The decisions were, when considered in isolation, perfectly understandable: it is easier to not bother about the complexities associated with retiring systems, and it is easier to assume that technicians can merge the systems. This is a typical trade-off situation with no simple solution.

5.3.2 Retireability

Retireability, unlike *compatibility*, can be reevaluated or renegotiated. While all cases considered the *retireability* of their existing systems this is an integral interwoven part of the decision process and is often not done explicitly. However it appears that cost plays an important role in the decision on *retireability*. Cost has many aspects; we will discuss several of them.
Existing systems represent a value to the company. Throwing away something that is known to work is often not an appealing option because it would mean discarding part of the investment. Some of the systems discussed represent hundreds of person-years of development. Also discarded parts would have to be replaced, requiring another investment. The organization, at present, may not have the necessary resources to do this. In case F2 implementation and validation of a 3D simulator (*Start from Scratch*) would take the better part of 10 years (I_{F2f}:6), while an *Evolutionary Merge*, even though the systems were not totally compatible, was estimated to take less time and therefore preferred.
Another aspect that affects retireability is satisfaction. Satisfaction is very broad and can involve many stakeholders (architects, users, management, etc.) and many aspects (functionality, architecture, performance, modifiability, etc.). When one or more of the existing systems are considered dissatisfactory there is a tendency to favor replacing the dissatisfactory system(s). If some of the existing systems are considered aged, they are candidates for retirement, i.e. *Choose One* or *Start from Scratch*, as was the case for the HMIs in case D (I_{Db}:3).

5.3.3 Selection of a Strategy

Start from Scratch can be implemented as either build in-house or acquire an external system. If the domain is mature and the software systems are not core products but supporting the organization, it may be well worth to transition to a commercial or open source solution. This was the case in case F3, where a commercial software issue tracking system was acquired.
Most organizations have formalized processes for development, evolution, and retirement of software systems, and these strategies can be cast in these terms. *No Integration* means the existing systems are evolved independently. If *Choose One* is possible to implement, it appears to be the simplest strategy as it only involves retiring some systems. *Start from Scratch* means planning the retirement of the existing systems while starting a new development project. *Evolutionary Merge* cannot readily be expressed in terms of existing processes, which might indicate that it is more difficult to implement, especially since it often involves a long time scale. Once again though, the costs must be weighed against other influences.
Availability of resources, such as time, money, people, and skills, has a big influence on the choice of strategy. Fundamentally, the architect and the organization must ask whether a certain strategy can be afforded. Even if the expected outcome would be a common, high-quality system, the costs could simply be prohibitive. In case E2, resource constraints resulted in some integration of two existing systems, and the retirement of the third system without replacement.
The question 'Do we want one common system?' is a very fundamental question and a positive answer to this is usually the starting point of the integration project. One fundamental reason we do not have an example where *No Integration* was selected explicitly is that the selection of cases was based on their actually trying some sort of integration. However, case F1 illustrates the situation when the existing systems are incompatible and cannot be discontinued, which leaves only strategy *No Integration*. In this situation, either retireability has to be reconsidered (together with the associated costs, assignment of resources, etc.) or the fundamental question about the value of integrating at all has to be revisited (I_{F1b}:3, I_{F1c}:9).

5.4 Feedback

There are numerous sources of influences both on whether any of the existing systems can be retired and on the selection of a strategy. It is almost impossible to distinguish between how the factors mentioned influences the retireability decision, which is often not made explicitly at all, and how these factors were re-evaluated given the resulting set of possible strategies. In case C, this feedback was fairly explicit: faced with the time needed for an *Evolutionary Merge* (the only available possibility), it was decided that retiring one system would cause less harm, and the decision could be changed to *Choose One*. However this decision took quite some time to reach resulting in a cost of at least several hundred person-years. Although is it impossible to predict what the outcome would have been if the decision to retire was taken earlier it is likely that it would have saved a lot of time and thus money. In many other cases analyses, decisions, and reconsiderations were more interleaved. We believe that all influencing factors involved in this feedback should be based on a proper analysis, documented explicitly, and made in a timely manner.

6. RELATED WORK

Existing research in related areas is presented below, starting with software integration, continuing with software architecture and architectural evaluation methods, and strategic planning.

6.1 Software Integration

In our previous literature survey [15], we found that there are two classes of research on the topic of software integration:

- Basic research describing integration rather fundamentally in terms of a) interfaces [13,34,35], b) architecture [1,10,12], architectural mismatch [9], and architectural patterns [2,8,27], and c) information/taxonomies/data models [11]. These foundations are directly useful in this context.
- There are three major fields of application: a) Component-Based Software Engineering [6,20,30,33], including component technologies, b) standard interfaces and open systems [20,21], and c) Enterprise Application Integration (EAI) [7,26]. These existing fields address somewhat different problems than ours:
  - Integration in these fields means that components or systems complement each other and are assembled into a larger system, while we consider systems that overlap functionally. The problem for us is therefore not to assemble components into one whole, but to take two (or more) whole systems and reduce the overlap to create one single whole, containing the best of the previous systems.
  - These fields typically assume that components (or systems) are acquired from third parties controlling their development, meaning that modifying them is not an option. We also consider systems completely controlled in-house, and this constraint consequently does not apply.
  - The goals of integration in these fields are to reduce development costs and time, while not sacrificing quality. In our context the goals are to reduce maintenance costs (while not sacrificing quality).

There is no existing literature that directly addresses the context of the present research: integration or merge of software completely controlled and owned within an organization. While we certainly can and should utilize the knowledge and approaches of these fairly mature fields, our research fills an apparent gap.

The architect is considered being the person who understands the language and concerns of other stakeholders [28,36], and/or the person who monitors and decides about all changes being made to the system to ensure conceptual integrity and avoid deterioration [22,32]. The academic focus of software architecture has been the (static) structure of the system, in terms of "components" (or "entities") and "connectors" [1,10,23]. The present paper describes many important issues an architect needs to consider during in-house integration, which only partly involves the structure of the existing systems.

There are several proposed methods for architectural analysis, mainly designed to be used during new development, such as the Architecture Trade-Off Analysis Method (ATAM) and Cost-Benefit Analysis Method (CBAM) [3]. Closely related to our description of compatibility is the seminal "architectural mismatch" paper, which points out many issues to be assessed as part of the architectural compatibility [9]. Also related to assessing architectural compatibility are architectural documentation good practices [4,12,14]. It has been observed that a similar structure of the existing systems is a necessary but not sufficient condition for compatibility [16]. There exist catalogues of generally useful structural patterns, however these are specifically intended for use during new development [2,8,27].

Many issues are not purely technical but require insight into business, and many decisions require awareness of he organization's overall strategies. Strategic planning (and strategic management) is known from business management as a tool for this kind of reasoning, that is to systematically formulate the goals of the organization and compare with the current and forecasted environment, and take appropriate measures to be able to adapt (and possibly control) the environmental changes [5,31]. In our case, investigating retireability clearly fits within the framework of strategic planning, by explicitly considering the money already invested, existing (dis)satisfaction, risk of future dissatisfaction, estimated available resources, and weigh this based on the perceived possible futures. In fact, the whole process we have described, and perhaps much of an architect's activities should be cast in terms of strategic planning such as the PESTEL framework or the Porter Five Forces framework [24]. (It should perhaps be noted that our term "integration strategy" is a plan, which is not synonymous to a company strategy in the sense of strategic planning.)

7. SUMMARY AND CONCLUSIONS

In this paper we present strategies an organization may select when faced with two or more systems with similar functionality, of which the source code is available and changes can be made. The strategies are based on actual integration projects and are: *No Integration*, *Start from Scratch*, *Choose One*, and two types of *Merge*: *Evolutionary Merge* and *Instant Merge*.

No Integration describes the situation in which no process towards a common system is made. *Start from Scratch* involves either a new development process or the acquisition of a commercial product while retiring the old systems, *Choose One* means selecting one system to replace the others, and *Merge* will see components from both existing systems combined into the new system that replaces them. *Evolutionary Merge* seems to be the most complex strategy to implement, and *Instant Merge* seems to be theoretical, possible only in the rare situation of systems with very similar structures, built using the same or very similar technologies, standards and other conventions. *Instant Merge* strategy was the only strategy not observed in the cases, although it appears this sometimes is what management has in mind when demanding a merge of the existing systems into one.

There are two concerns to consider which will limit the set of strategies that can possibly by selected, namely architectural *compatibility* and *retireability*.

There is no exhaustive definition of architectural *compatibility*, but some observations in the cases are that the structures, data models, and environment that defines the components must be similar. Standards and a common ancestry can make systems somewhat compatible, although it is not a guarantee for total compatibility. Besides that compatibility is very system specific and should be thoroughly analyzed by the architects before choosing a strategy to avoid problems during integration. With somewhat compatible systems, *Evolutionary Merge* is possible but *Instant Merge* is not. If the systems are not compatible at all neither of the *Merge* strategies are possible.

Determining the *retireability* of the existing systems depends on numerous factors. Major influences found in the cases were investments made, cost, satisfaction, time to market, and available resources. Typically there is a tight feedback loop between

evaluating these influences and the resulting possible strategies. Hurdles in this feedback loop, either because architects are unable to communicate their findings to management or because management delays taking decisions, cause the integration cost to increase. If it is not possible to retire all existing systems this excludes *Start from Scratch* as a possible strategy. If some of the existing systems can be retired, it is possible to *Choose One*. If none can be retired this leaves *Merge* as the only possibility.

The worst case scenario is that the existing systems are considered impossible to retire and are also incompatible. This leaves no strategy left but *No Integration*, which of course brings none of the potential benefits from integration. This was observed in one of the cases.

Drawing on the experiences from the cases, we suggest that future architects together with other stakeholders make these influences explicit thus making strategy selection faster, better founded, and decrease the cost associated with uninformed decisions. The message to management is that delaying a decision comes with a cost.

7.1 Future work

We have also analyzed the reuse of components and architectural solutions into a common system elsewhere [16], as well as good practices for the strategy selection process [18]. We would like to compile a more complete catalogue of different aspects of architectural compatibility, for example by investigating structural patterns and styles [2,8,27] suitable for *Merge*, the environment defining the components, and the impact of cross-cutting concerns whose implementation is scattered throughout the system. This would also provide insight in how to make *Evolutionary Merge* closer to the desired *Instant Merge*.

7.2 Acknowledgements

We would like to thank all interviewees and their organizations for sharing their experiences and allowing us to publish them. We would also like to express our gratitude for the helpful suggestions from the anonymous reviewers.

8. REFERENCES

[1] Bass L., Clements P., and Kazman R., *Software Architecture in Practice* (2nd edition), ISBN 0-321-15495-9, Addison-Wesley, 2003.

[2] Bushmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., *Pattern-Oriented Software Architecture - A System of Patterns*, ISBN 0-471-95869-7, John Wiley & Sons, 1996.

[3] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Evaluating Software Architectures*, ISBN 0-201-70482-X, Addison-Wesley, 2001.

[4] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Documenting Software Architectures: Views and Beyond*, ISBN 0-201-70372-6, Addison-Wesley, 2002.

[5] Courtney H., *20|20 Foresight : Crafting Strategy in an Uncertain World*, ISBN 1-57851-266-2, Harvard Business School Press, 2001.

[6] Crnkovic I. and Larsson M., *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[7] Cummins F. A., *Enterprise Integration: An Architecture for Enterprise Application and Systems Integration*, ISBN 0471400106, John Wiley & Sons, 2002.

[8] Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns - Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2, Addison-Wesley, 1995.

[9] Garlan D., Allen R., and Ockerbloom J., "Architectural Mismatch: Why Reuse is so Hard", In *IEEE Software*, volume 12, issue 6, pp. 17-26, 1995.

[10] Garlan D. and Shaw M., "An Introduction to Software Architecture", In *Advances in Software Engineering and Knowledge Engineering*, volume I, 1993.

[11] Guarino N., *Formal Ontology in Information Systems*, ISBN 9051993994, IOS Press, 1998.

[12] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, ISBN 0-201-32571-3, Addison-Wesley, 2000.

[13] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, IEEE, 1990.

[14] IEEE Architecture Working Group, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000, IEEE, 2000.

[15] Land R. and Crnkovic I., "Existing Approaches to Software Integration – and a Challenge for the Future", In *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, Linköping University, 2004.

[16] Land R., Crnkovic I., Larsson S., and Blankers L., "Architectural Reuse in Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of First International Conference on the Quality of Software Architectures (QoSA)*, Springer, 2005.

[17] Land R., Larsson S., and Crnkovic I., *Interviews on Software Integration*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-177/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2005.

[18] Land R., Larsson S., and Crnkovic I., "Processes Patterns for Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI)*, 2005.

[19] Maxwell Joseph A., "Understanding and validity in qualitative research", In *Harvard Educational Review*, volume 62, issue 3, pp. 279-300, 1992.

[20] Meyers C. and Oberndorf P., *Managing Software Acquisition: Open Systems and COTS Products*, ISBN 0201704544, Addison-Wesley, 2001.

[21] Meyers C. and Oberndorf T., *Open Systems: The Promises and the Pitfalls*, ISBN 0-201-70454-4, Addison-Wesley, 1997.

[22] Parnas D. L., "Software Aging", In *Proceedings of The 16th International Conference on Software Engineering*, pp. 279-287, IEEE Press, 1994.

[23] Perry D. E. and Wolf A. L., "Foundations for the study of software architecture", In *ACM SIGSOFT Software Engineering Notes*, volume 17, issue 4, pp. 40-52, 1992.

[24] Porter M. E., *Competitive Strategy: Techniques for Analyzing Industries and Competitors*, ISBN 0684841487, Free Press, 1998.

[25] Robson C., *Real World Research* (2nd edition), ISBN 0-631-21305-8, Blackwell Publishers, 2002.

[26] Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.
[27] Schmidt D., Stal M., Rohnert H., and Buschmann F., *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, Wiley Series in Software Design Patterns, ISBN 0-471-60695-2, John Wiley & Sons Ltd., 2000.
[28] Sewell M. T. and Sewell L. M., *The Software Architect's Profession - An Introduction*, Software Architecture Series, ISBN 0-13-060796-7, Prentice Hall PTR, 2002.
[29] Strauss A. and Corbin J. M., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (2nd edition), ISBN 0803959400, Sage Publications, 1998.
[30] Szyperski C., *Component Software - Beyond Object-Oriented Programming*, ISBN 0-201-17888-5, Addison-Wesley, 1998.
[31] Thompson Jr. A. A. and Strickland III A. J., *Strategic Management : Concepts and Cases* (11th edition), ISBN 0-07-303714-1, Irwin/McGraw-Hill, 1999.
[32] van Gurp J. and Bosch J., "Design Erosion: Problems & Causes", In *Journal of Systems & Software*, volume 61, issue 2, pp. 105-119, 2002.
[33] Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.
[34] Wegner P., "Interoperability", In *ACM Computing Surveys*, volume 28, issue 1, 1996.
[35] Wileden J. C. and Kaplan A., "Software Interoperability: Principles and Practice", In *Proceedings of 21st International Conference on Software Engineering*, pp. 675-676, ACM, 1999.
[36] WWISA, *Worldwide Institute of Software Architects*, URL: http://www.wwisa.org, 2002.
[37] Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.

APPENDIX: INTERVIEW QUESTIONS

1. Describe the technical history of the systems that were integrated: e.g. age, number of versions, size (lines of code or other measure), how was functionality extended, what technology changes were made? What problems were experienced as the system grew?
2. Describe the organizational history of the systems. E.g. were they developed by the same organization, by different departments within the same organization, by different companies? Did ownership change?
3. What were the main reasons to integrate? E.g. to increase functionality, to gain business advantages, to decrease maintenance costs? What made you realize that integration was desirable/ needed?
4. At the time of integration, to what extent was source code the systems available, for use, for modifications, etc.? Who owned the source code? What parts were e.g. developed in-house, developed by contractor, open source, commercial software (complete systems or smaller components)?
5. Which were the stakeholders of the previous systems and of the new system? What were their main interests of the systems? Please describe any conflicts.
6. Describe the decision process leading to the choice of how integration? Was it done systematically? Were alternatives evaluated or was there an obvious way of doing it? Who made the decision? Which underlying information for making the decision was made (for example, were some analysis of several possible alternatives made)? Which factors were the most important for the decision (organizational, market, expected time of integration, expected cost of integration, development process, systems structures (architectures), development tools, etc.)?
7. Describe the technical solutions of the integration. For example, were binaries or source code wrapped? How much source code was modified? Were interfaces (internal and/or external) modified? Were any patterns or infrastructures (proprietary, new or inherited, or commercial) used? What was the size of the resulting system?
8. Why were these technical solutions (previous question) chosen? Examples could be to decrease complexity, decrease source code size, to enable certain new functionality.
9. Did the integration proceed as expected? If it was it more complicated than expected, how did it affect the project/product? For example, was the project late or cost more than anticipated, or was the product of less quality than expected? What were the reasons? Were there difficulties in understanding the existing or the resulting system, problems with techniques, problems in communication with people, organizational issues, different interests, etc.?
10. Did the resulting integrated system fulfill the expectations? Or was it better than expected, or did not meet the expectations? Describe the extent to which the technical solutions contributed to this. Also describe how the process and people involved contributed – were the right people involved at the right time, etc.?
11. What is the most important factor for a successful integration according your experiences? What is the most common pitfall?
12. Have you changed the way you work as a result of the integration efforts? For example, by consciously defining a product family (product line), or some components that are reused in many products?