Merging In-House Developed Software Systems – A Method for Exploring Alternatives

Rikard Land, Jan Carlson, Ivica Crnković, Stig Larsson

Mälardalen University, Department of Computer Science and Electronics
PO Box 883, SE-721 23 Västerås, Sweden
+46 21 10 70 35

{rikard.land, jan.carlson, ivica.crnkovic, stig.larsson}@mdh.se, http://www.idt.mdh.se/{~rld, ~jcn, ~icc}

Abstract

An increasing form of software evolution is software merge – when two or more software systems are being merged. The reason may be to achieve new integrated functions, but also remove duplication of services, code, data, etc. This situation might occur as systems are evolved in-house, or after a company acquisition or merger. One potential solution is to merge the systems by taking components from the two (or more) existing systems and assemble them into an existing system. The paper presents a method for exploring merge alternatives at the architectural level, and evaluates the implications in terms of system features and quality, and the effort needed for the implementation. The method builds on previous observations from several case studies. The method includes well-defined core model with a layer of heuristics in terms of a loosely defined process on top. As an illustration of the method usage a case study is discussed using the method.

1. Introduction

When organizations merge, or collaborate very closely, they often bring a legacy of in-house developed software systems. Often these systems address similar problems within the same business and there is usually some overlap in functionality and purpose. A new system, combining the functionality of the existing systems, would improve the situation from an economical and maintenance point of view, as well as from the point of view of users, marketing and customers. During a previous study involving nine cases of such in-house integration [10], we saw some drastic strategies, involving retiring (some of) the existing systems and reusing some parts, or only reutilizing knowledge and building a new system from scratch. We also saw another strategy of resolving this situation, which is the focus of the present paper: to merge the systems, by reassembling various parts from several existing system into a new system. From many points of view, this is a desirable solution, but based on previous research this is typically very difficult and is not so common in practice; there seem to be some prerequisites for this to be possible and feasible [10].

There is a need to relatively fast and accurately find and evaluate merge solutions, and our starting point to address this need has been the following previous observations [10]:

1. Similar high-level structures seem to be a prerequisite for merge. Thus, if the structures of the existing systems are not similar, a merge seems in practice unfeasible.
2. A development-time view of the system is a simple and powerful system representation, which lends itself to reasoning about project characteristics, such as division of work and effort estimations.
3. A suggested beneficial practice is to assemble the architects of the existing systems in a meeting early in the process, where various solutions are outlined and discussed. During this type of meeting, many alternatives are partly developed and evaluated until (hopefully) one or a few high-level alternatives are fully elaborated.
4. The merge will probably take a long time. To sustain commitment within the organization, and avoid too much of parallel development, there is a need to perform an evolutionary merge with stepwise deliveries. To enable this, the existing systems should be delivered separately, sharing more and more parts until the systems are identical.

This paper presents a systematic method for exploring merge alternatives, which takes these observations into account: by 1) assuming similar high-level structures, 2) utilizing static views of the systems, 3) being simple enough to be able to learn and use during the architects' meetings, and 4) by focusing not only on an ideal future system but also stepwise deliveries of the existing systems. The information gathered from nine

case studies was generalized into the method presented in this paper. To refine the method, we made further interviews with participants in one of the previous cases, which implemented the merge strategy most clearly.

The rest of the paper is organized as follows. We define the method in Section 2 and discuss it by means of an example in Section 3. Section 4 discusses important observations from the case and argues for some general advices based on this. Section 5 surveys related work. Section 6 summarizes and concludes the paper and outlines future work.

2. Software Merge Exploration Method

Our software merge exploration method consists of two parts: (i) a model, i.e., a set of formal concepts and definitions, and (ii) a process, i.e., a set of human activities that utilizes the model. The model is designed to be simple but should reflect reality as much as possible, and the process describes higher-level reasoning and heuristics that are suggested as useful practices.

To help explaining the method, we start with a simple example in Section 2.1, followed by a description of the method's underlying model (Section 2.2) and the suggested process (Section 2.3).

2.1 An Explanatory Example

Figure 1a shows two simple music sequencer software systems structured according to the "Model-View-Controller" pattern [2]. The recorded music would be the model, which can be viewed as a note score or as a list of detailed events, and controlled by mouse clicks or by playing a keyboard.

The method uses the module view [3,5] (or development view [8]), which describes modules and "use" dependencies between them. Parnas defined the "use" dependency so that module α is said to use module β if module α relies on the correct behavior of β to accomplish its task [14].

In our method, the term *module* refers to an encapsulation of a particular functionality, purpose or responsibility on an abstract level. A concrete implementation of this functionality is called a *module instance*. In the example, both systems have a EventView module, meaning that both systems provide this particular type of functionality (e.g., a note score view of the music). The details are probably different in the two systems, though, since the functionality is provided by different concrete implementations (the module instances EventView_A and EventView_B, respectively). The method is not restricted to module instances that are present in the existing systems but also those that are possible in a future system; such new module instances could be either a planned implementation (e.g., $\text{EventView}_{new\_impl}$), an already existing module to be reused in-house from some other program (e.g., $\text{EventView}_{pgm\_name}$), or an open source or commercial component ($\text{EventView}_{component\_name}$).

2.2 The Model

Our proposed method builds on a model consisting of three parts: a set of model elements, a definition of inconsistency in terms of the systems' structures, and a set of permissible user operations.

2.2.1 Concepts and Notation

The following concepts are used in the model:

- We assume there are two or more existing *systems*, (named with capital letters, and parameterized by X, Y, etc.).
- A *module* represents a conceptual system part with a specific purpose (e.g., EventView in Figure 1). Modules are designated with capital first letter; in the general case we use Greek letters α and β.
- A *module instance* represents a realization of a module. It is denoted α_X where α is a module and X is either an existing system (as in EventView_A) or an indication that the module is new to the systems (as in $\text{EventView}_{pgm\_name}$ or $\text{EventView}_{component\_name}$).
- A *"use" dependency* (or *dependency* for short) from module instance α_X to module instance β_Y means that α_X relies on the correct behavior of β_Y to accomplish its task. We use the textual notation $\alpha_X \rightarrow \beta_Y$ to represent this.
- A *dependency graph* captures the structure of a system. It is a directed graph where each node in the graph represents a module instance and the edges (arrows) represent use dependencies. In Figure 1a, we have for example the dependencies $\text{NoteView}_A \rightarrow \text{MusicModel}_A$ and $\text{MouseCtrl}_B \rightarrow \text{MusicModel}_B$.
- An *adaptation* describes that a modification is made to α_X in order for it to be compatible, or *consistent* with β_Y, and is denoted $\langle \alpha_X, \beta_Y \rangle$ (see 2.2.2 below).
- A *scenario* consists of a dependency graph for each existing system and a single set of adaptations.

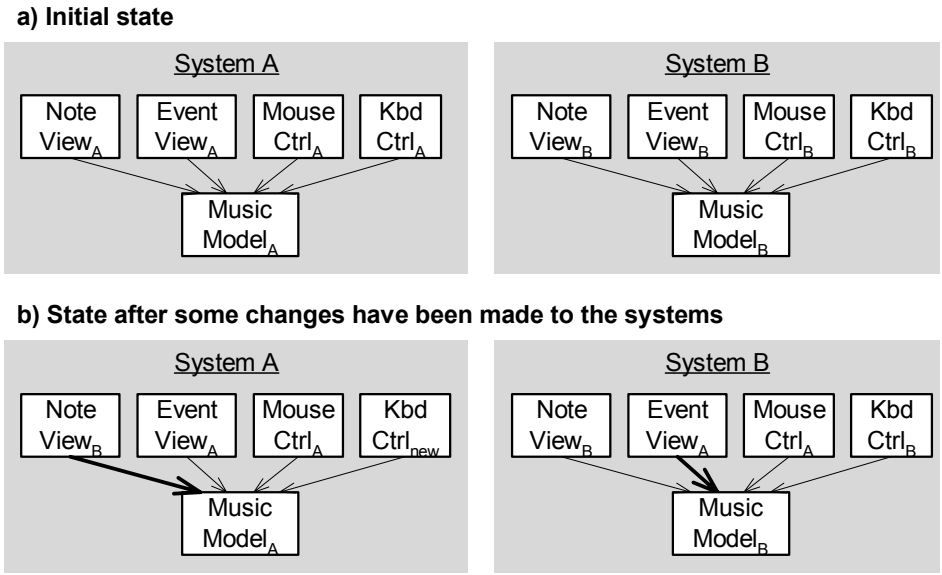


Figure 1. Two example systems with the same structure being merged.

2.2.2 Inconsistency

A dependency from α_X to β_Y can be *inconsistent*, meaning that β_Y cannot be used by α_X. Trivially, the dependency between two module instances from the same system is consistent without further adaptation. For the dependency between two modules from different systems we cannot say whether they are consistent or not. Most probably they are inconsistent, which has to be resolved by some kind of adaptation if we want to use them together in a new system. The actual adaptations made could in practice be of many kinds: some wrapping or bridging code, or modifications of individual lines of code; see further discussion in 4.1.

Formally, a dependency $\alpha_X \rightarrow \beta_Y$ is *consistent* if $X = Y$ or if the adaptation set contains $\langle \alpha_X, \beta_Y \rangle$ or $\langle \beta_Y, \alpha_X \rangle$. Otherwise, the dependency is *inconsistent*. A dependency graph is consistent if all dependencies are consistent; otherwise it is inconsistent. A scenario is consistent if all dependency graphs are consistent; otherwise it is inconsistent.

Example: The scenario in Figure 1b is inconsistent, because of the inconsistent dependencies from NoteView$_B$ to MusicModel$_A$ (in System A) and from EventView$_A$ to MusicModel$_B$ (in System B). The dependencies from KbdCtrl$_{new}$ to MusicModel$_A$ (in System A) and from MouseCtrl$_A$ to MusicModel$_B$ (in System B) on the other hand are consistent, since there are adaptations $\langle$KbdCtrl$_{new}$, MusicModel$_A\rangle$ and $\langle$MusicModel$_B$, MouseCtrl$_A\rangle$ representing that KbdCtrl$_{new}$ and MusicModel$_B$ have been modified to be consistent with MusicModel$_A$ and MouseCtrl$_A$ respectively.

2.2.3 Scenario Operations

The following operations can be performed on a scenario:

1. Add an adaptation to the adaptation set.
2. Remove an adaptation from the adaptation set.
3. Add the module instance α_X to one of the dependency graphs, if there exists an α_Y in the graph. Additionally, for each module β, such that there is a dependency $\alpha_Y \rightarrow \beta_Z$ in the graph, a dependency $\alpha_X \rightarrow \beta_W$ must be added for some β_W in the graph.
4. Add the dependency $\alpha_X \rightarrow \beta_W$ if there exist a dependency $\alpha_X \rightarrow \beta_Z$ (with $Z \neq W$) in the graph.
5. Remove the dependency $\alpha_X \rightarrow \beta_W$ if there exists a dependency $\alpha_X \rightarrow \beta_Z$ (with $Z \neq W$) in the graph.
6. Remove the module instance α_X from one of the dependency graphs, if there are no edges to α_X in the graph, and if the graph contains another module instance α_Y (i.e., with $X \neq Y$).

Note that these operations never change the participating modules of the graphs (if there is an α_X in the initial systems, they will always contain some α_Y). Similarly, dependencies between modules are also preserved Note also that we allow two or more instances for the same module in a system; when this could be suitable for a real system is discussed in 4.2.

2.3 The Process

The suggested process consists of two phases, the first consisting of two simple preparatory activities (P-I and P-II), and the second being recursive and exploratory (E-I – E-IV).

The scope of the method is within an early meeting of architects, where they (among other tasks) outline various merge solutions. To be able to evaluate various alternatives, some evaluation criteria should be provided by management, product owners, or similar stakeholders. Such criteria can include quality attributes for the system, but also considerations regarding development parameters such as cost and time limits. Other boundary conditions are the strategy for the future architecture and anticipated changes in the development organization. Depending on the circumstances, evaluation criteria and boundary conditions could be renegotiated to some extent, once concrete alternatives are developed.

2.3.1 Preparatory Phase

The *Preparatory* phase consists of two activities:

Activity P-I: Describe Existing Systems. First, the dependency graphs of the existing systems must be prepared, and common modules must be identified. These graphs could be found in existing models or documentation, or extracted by reverse engineering methods, or simply created by the architects themselves.

Activity P-II: Describe Desired Future Architecture. The dependency graph of the future system has the same structure, in terms of modules, as the existing systems. For some modules it may be imperative to use some specific module instance (e.g., α_X because it has richer functionality than α_Y, or a new implementation α_{new} because there have been quality problems with the existing α_X and α_Y). For other modules, α_X might be preferred over α_Y, but the final choice will also depend on other implications of the choice, which is not known until different alternatives are explored. The result of this activity is an outline of a desired future system, with some annotations, that serve as a guide during the exploratory phase. This should include some quality goals for the system as a whole.

2.3.2 Exploratory Phase

The result of the preparatory phase is a single scenario corresponding to the structure and module instances of the existing systems. The exploratory phase can then be described in terms of four activities: E-I "Introduce Desired Changes", E-II "Resolve Inconsistencies", E-III "Branch Scenarios", and E-IV "Evaluate Scenarios".

The order between them is not pre-determined; any activity could be performed after any of the others. They are however not completely arbitrary: early in the process, there will be an emphasis on activity E-I, where *desired changes* are introduced. These changes will lead to *inconsistencies that need to be resolved* in activity E-II. As the exploration continues, one will need to *branch scenarios* in order to explore different choices; this is done in activity E-III. One also wants to continually *evaluate the scenarios* and compare them, which is done in activity E-IV. Towards the end when there are a number of consistent scenarios there will be an emphasis on evaluating these deliveries of the existing systems. For all these activities, decisions should be described so they are motivated by, and traceable to, the specified evaluation criteria and boundary conditions. These activities describe high-level operations that are often useful, but nothing prohibits the user from carrying out any of the primitive operations defined above at any time.

Activity E-I: Introduce Desired Changes. Some module instances, desired in the future system, should be introduced into the existing systems. In some cases, it is imperative where to start (as described for activity P-II); the choice may e.g., depend on the local priorities for each system (e.g., "we need to improve the MusicModel of system A"), and/or some strategic considerations concerning how to make the envisioned merge succeed (e.g., "the MusicModel should be made a common module as soon as possible").

Activity E-II: Resolve Inconsistencies. As modules are exchanged in the graphs, dependencies $\alpha_X \rightarrow \beta_Y$ might become inconsistent. There are several ways of resolving these inconsistencies:

- Either of the two module instances could be modified to be consistent with the interface of the other. In the model, this means adding an adaptation to the adaptation set. In the example of Figure 1b, the inconsistency between NoteView_B and MusicModel_A in System A can be solved by adding either of the adaptations $\langle \text{NoteView}_B, \text{MusicModel}_A \rangle$ or $\langle \text{MusicModel}_A, \text{NoteView}_B \rangle$ to the adaptation set. (Different types of possible modifications in practice are discussed in Section 4.1.)
- Either of the two module instances could be exchanged for another. There are several variations on this:
  - A module instance is chosen so that the new pair of components is already consistent. This means that α_X is exchanged either for α_Y (which is consistent with β_Y as they come from the same system Y) or for some other α_Z for which there is an adaptation $\langle \alpha_Z, \beta_Y \rangle$ or $\langle \beta_Y,$

$\alpha_Z\rangle$. Alternatively, β_Y is exchanged for β_X or some other β_Z for which there is an adaptation $\langle\beta_Z, \alpha_X\rangle$ or $\langle\alpha_X , \beta_Z\rangle$. In the example of Figure 1b, MusicModel_A could be replaced by MusicModel_B to resolve the inconsistent dependency $\text{NoteView}_B \rightarrow \text{MusicModel}_A$ in System A.

  - A module instance is chosen that did not exist in either of the previous systems. This could be either of:
    i) a module reused in-house from some other program (which would come with an adaptation cost),
    ii) a planned or hypothesized new development (which would have an implementation cost, but low or no adaptation cost), or
    iii) an open source or commercial component (which involves acquirement costs as well as adaptation costs, which one would like to keep separate).

- One more module instance could be introduced for one of the modules, to exist in parallel with the existing; the new module instance would be chosen so that it already is consistent with the instance of the other module (as described for exchanging components). The previous example in Figure 1a and b is too simple to illustrate the need for this, but in Section 4 the industrial case will illustrate when this might be needed and feasible. Coexisting modules are also further discussed in Section 4.1.

Some introduced changes will cause new inconsistencies, that need to be resolved (i.e., this activity need to be performed iteratively).

Activity E-III: Branch Scenarios. As a scenario is evolved by applying the operations to it (most often according to either of the high-level approaches of activities E-I and E-II), there will be occasions where it is desired to explore two or more different choices in parallel. For example, several of the resolutions suggested in activity E-II might make intuitive sense, and both choices should be explored. It is then possible to copy the scenario, and treat the two copies as branches of the same tree, having some choices in common but also some different choices.

Activity E-IV: Evaluate Scenarios. As scenarios evolve, they need to be evaluated in order to decide which branches to evolve further and which to abandon. Towards the end of the process, one will also want to evaluate the final alternatives more thoroughly, and compare them – both with each other and with the pre-specified evaluation criteria and boundary conditions (which might at this point be reconsidered to some extent). The actual state of the systems must be evaluated, i.e., the actually chosen module instances plus the modifications to reduce inconsistencies). Do the systems contain many shared modules? Are the chosen modules the ones desired for the future system (richest functionality, highest quality, etc.)? Can the system as a whole be expected to meet its quality goals?

2.3.3 Accumulating Information

As these activities are carried out, there is some information that should be stored for use in later activities. As operations are performed, information is accumulated. Although this information is created as part of an operation within a specific scenario, the information can be used in all other scenarios; this idea would be particularly useful when implemented in a tool. We envision that any particular project or tool would define its own formats and types of information; in the following we give some suggestions of such useful information and how it would be used.

Throughout the exploratory activities, it would be useful to have some ranking of modules readily available, such as "EventView_A is preferred over EventView_B because it has higher quality". A tool could use this information to color the chosen modules to show how well the outlined alternatives fit the desired future system.

For activity E-II "Resolve Inconsistencies", it would be useful to have information about e.g., which module could or could not coexist in parallel. Also, some information should be stored that is related to how the inconsistencies are solved. There should at least be a short textual description of what an adaptation means in practice. Other useful information would be the efforts and costs associated with each acquirement and adaptation; if this information is collected by a tool, it becomes possible to extract a list of actions required per scenario, including the textual descriptions of adaptations and effort estimates. It is also possible to reason about how much of the efforts required that are "wasted", that is: is most of the effort related to modifications that actually lead towards the desired future system, or is much effort required to make modules fit only for the next delivery and then discarded? The evaluation criteria and boundary conditions mentioned in Section 2.2 could also be used by a tool to aid or guide the evaluation in the activity E-IV.

3. An Industrial Case Study

In a previous multiple case study on the topic of in-house integration, the nine cases in six organizations had implemented different integration solutions [10]. We returned to the one case that had clearly chosen the merge strategy and successfully implemented it

(although it is not formally released yet); in previous publications this case is labelled "case F2". The fact that this was one case out of nine indicates that the prerequisites for a merge are not always fulfilled, but also that they are not unrealistic (two more cases involved reusing parts from several existing systems in a way that could be described as a merge). To motivate the applicability of the proposed method, this section describes the events of an industrial case and places them in the context of our method.

3.1 Research Method

This part of the research is thus a single case study [17]. Our sources of information have been face-to-face interviews with the three main developers on the US side (there is no title "architect" within the company) and the two main developers on the Swedish side, as well as the high-level documentation of the Swedish system. All discussion questions and answers are published together with more details on the study's design in a technical report [9].

Although the reasoning of the case follows the method closely, the case also demonstrates some inefficiency due to not exploring the technical implications of the merge fully beforehand. It therefore supports the idea of the method being employed to analyze and explore merge alternatives early, before committing to a particular strategy for the in-house integration (merge or some other strategy).

3.2 The Case

The organization in the case is a US-based global company that acquired a slightly smaller global company in the same business domain, based in Sweden. To support the core business, computer simulations are conducted. Both sites have developed software for simulating 3D physics, containing state-of-the-art physics models, many of the models also developed in-house.

As the results are used for real-world decisions potentially affecting the environment and human lives, the simulation results must be accurate (i.e., the output must correspond closely to reality). As the simulations are carried out off-line and the users are physics specialists, many other runtime quality properties of the simulation programs are not crucial, such as reliability (if the program crashes for a certain input, the bug is located and removed), user-friendliness, performance, or portability. On the other side, the accuracy of the results are crucial.

Both systems are written in Fortran and consist of several hundreds of thousands lines of code, and the staff responsible for evolving these simulators are the interviewees, i.e., less than a handful on each site. There was a strategic decision to integrate or merge the systems in the long term. This should be done through cooperation whenever possible, rather than as a separate up-front project.

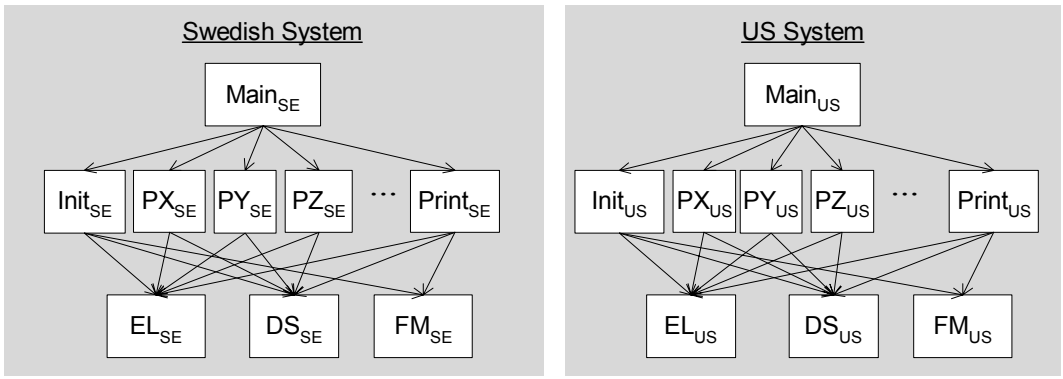
The rest of this section describes the events of the case in terms of the proposed activities of the method. It should be noted that although the interviewees met in a small group to discuss alternatives, they did not follow the proposed method strictly (which is natural, as the method has been formulated after, and partly influenced by, these events).

Activity P-I: Describe Existing Systems. Both existing systems are written in the same programming language (Fortran), and it was realized early that the two systems have very similar structure, see Figure 2a). There is a main program (Main) invoking a number of physics modules (PX, PY, PZ, …) at appropriate times, within two main loops. Before any calculations, an initialization module (Init) reads data from input files and the internal data structures (DS) are initialized. The physics modeled is complex, leading to complex interactions where the solution of one module affects others in a non-hierarchical manner. After the physics calculations are finished, a file management module (FM) is invoked, which collects and prints the results to file. All these modules use a common error handling and logging library (EL), and share the same data structures (DS). A merge seemed plausible also thanks to the similarities of the data models; the two programs model the same reality in similar ways.

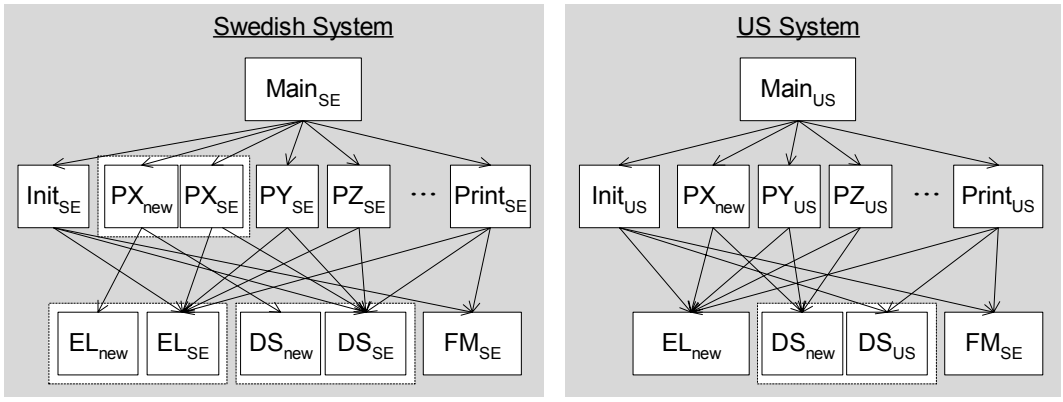
Activity P-II: Describe Desired Future Architecture. The starting point was to develop a common module for one particular aspect of the physics (PX_{new}), as both sides had experienced some limitations of their respective current physics models. Now being in the same company, it was imperative that they would join efforts and develop a new module that would be common to both programs; this project received some extra integration funding. Independent of the integration efforts, there was a common wish on both sides to take advantage of newer Fortran constructs to improve encapsulation and enforce stronger static checks.

Activity E-I: Introduce Desired Changes. As said, the starting point for integration was the module PX. Both sides wanted a fundamentally new physics model, so the implementation was also completely new (no reuse), written by one of the Swedish developers. The two systems also used different formats for input and output files, managed by file handling modules (FM_{SE} and FM_{US}). The US system chose to incorporate the Swedish module for this, which has required some changes to the modules using the file handling module.

a) Initial state



b) Current state



Adaptation set: <$Main_{SE}$, PX_{new}> <PX_{new}, EL_{new}> <PX_{new}, DS_{new}> <$Main_{US}$, PX_{new}> <PY_{US}, EL_{new}> <PY_{US}, DS_{new}> <PZ_{US}, EL_{new}> <PZ_{US}, DS_{new}>

c) Future System

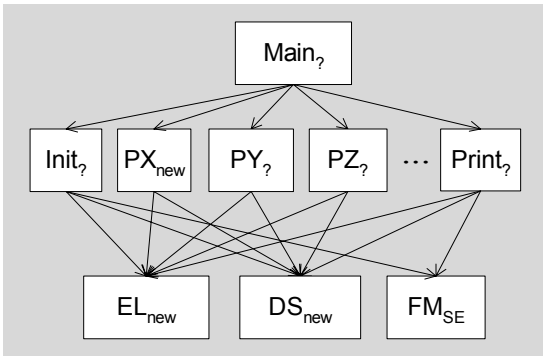


Figure 2: The current status of the systems of the case.

Activity E-II: Resolve Inconsistencies. The PX module of both systems accesses large data structures (DS) in global memory, shared with the other physics modules. An approach was tried where adapters were introduced between a commonly defined interface and the old implementations, but was abandoned as this solution became too complex. Instead, a new implementation of data structures was introduced. This was partially chosen because it gave the opportunity to use newer Fortran constructs which made the code more structured, and it enabled some encapsulation and access control as well as stronger type checking than before.

This led to new inconsistencies that needed to be resolved. In the US system, six man-months were spent on modifying the existing code to use the new data structures. The initialization and printout modules remained untouched however; instead a solution was

chosen where data is moved from the old structures (DS_{SE} and DS_{US}) to the new (DS_{new}) after the initialization module has populated the old structures, and data is moved back to the old structures before the printout module executes. In the Swedish system, only the parts of the data structures that are used by the PX module are utilized, the other parts of the program uses the old structures; the few data that are used both by the PX module and others had to be handled separately.

The existing libraries for error handling and logging (EL) would also need some improvements in the future. Instead of implementing the new PX module to fit the old EL module, a new EL module was implemented. The new PX module was built to use the new EL module, but the developers saw no major problems to let the old EL module continue to be used by other modules (otherwise there would be an undesirable ripple effect). However, for each internal shipment of the PX module, the US staff commented away the calls to the EL library; this was the fastest way to make it fit. In the short term this was perfectly sensible, since the next US release would only be used for validating the new model together with the old system. However, spending time commenting away code was an inefficient way of working, and eventually the US site incorporated the EL library and modified all other modules to use it; this was not too difficult as it basically involved replacing certain subroutine calls with others. In the Swedish system, the new EL library was used by the new PX module, while the existing EL module was used in parallel, to avoid modifying other modules that used it. Having two parallel EL libraries was not considered a major quality risk in the short run.

Modifying the main loop of each system, to make it call the new PX module instead of the old, was trivial. In the Swedish system there will be a startup switch for some years to come, allowing users to choose between the old and the new PX module for each execution. This is useful for validation of PX_{new} and is presented as a feature for customers.

E-III Branch Scenarios. As we are describing the actual sequence of events, this activity cannot be reported as such, although different alternatives were certainly discussed – and even attempted and abandoned, as for the data structure adapters.

E-IV Evaluate Scenarios. This activity is also difficult to isolate after the fact, as we have no available reports on considerations made. It appears as functionality was a much more important factor than non-functional (quality) attributes at the module level. At system level, concerns about development time qualities (e.g., discussions about parallel module instances and the impact on maintenance) seem to have been discussed more than runtime qualities (possibly because runtime qualities in this case are not crucial).

Figure 2 shows the initial and current state of the systems, as well as the desired outlined future system. (It is still discussed whether to reuse the module from either of the systems or create a new implementation, hence the question marks).

4. Discussion

This section discusses various considerations to be made during the exploration and evaluation, as highlighted by the case.

4.1 Coexisting Modules

To resolve an inconsistency between two module instances, there is the option of allowing two module instances (operation 2). Replacing the module completely will have cascading effects on the consistencies for all edges connected to it (both "used-by" and "using"), so having several instances has the least direct impact in the model (potentially the least modification efforts). However, it is not always feasible in practice to allow two implementations with the same purpose. The installation and runtime costs associated with having several modules for the same task might be prohibiting if resources are scarce. It might also be fundamentally assumed that there is only one single instance responsible for a certain functionality, e.g., for managing central resources. Examples could be thread creation and allocation, access control to various resources (hardware or software), security, etc. Finally, at development time, coexisting components violates the conceptual integrity of the system, and results in a larger code base and a larger number of interfaces to keep consistent during further evolution and maintenance. From this point of view, coexisting modules might be allowed as a temporary solution for an intermediate delivery, while planning for a future system with a single instance of each module (as in the case for modules EL and DS). However, the case also illustrates how the ability to choose either of the two modules for each new execution was considered useful (PX_{SE} and PX_{new} in the Swedish system).

We can see the following types of relationships between two particular module instances of the same module:

- **Arbitrary usage.** Any of the two parallel modules may be invoked at any time. This seems applicable for library type modules, i.e., modules that retains no state but only performs some action and returns, as the EL module in the case.
- **Alternating usage.** If arbitrary usage cannot be allowed, it might be possible to define some rules

for synchronization that will allow both modules to exist in the system. In the case, we saw accesses to old and new data structures in a pre-defined order, which required some means of synchronizing data at the appropriate points in time. One could also imagine other, more dynamic types of synchronization mechanisms useful for other types of systems: a rule stating which module to be called depending on the current mode of the system, or two parallel processes that are synchronized via some shared variables. (Although these kinds of solutions could be seen as a new module, the current version of the method only allows this to be specified as text associated to an adaptation.)

- **Initial choice.** The services of the modules may be infeasible to share between two modules, even over time. Someone will need to select which module instance to use, e.g., at compile time by means of compilation switches, or with an initialization parameter provided by the user at run-time. This was the case for the PX_{SE} and PX_{new} modules in the Swedish system.

The last two types of relationships requires some principle decision and rules at the system (architectural) level, while the signifying feature of the first is that the correct overall behaviour of the program is totally independent of which module instance is used at any particular time.

4.2 Similarity of Systems

As described in 2.1.1, the model requires that the structures of the existing systems are identical, which may seem a rather strong assumption. It is motivated by the following three arguments [10]:

- The previous multiple case study mentioned in Section 3.1 strongly suggests that similar structures is a prerequisite for merge to make sense in practice. That means that if the structures are dissimilar, practice has shown that some other strategy will very likely be more feasible (e.g., involving the retirement of some systems). Consequently, there is little motivation to devise a method that covers also this situation.
- We also observed that it is not so unlikely that systems in the same domain, built during the same era, indeed have similar structures.
- If the structures are not very similar at a detailed level, it might be possible to find a higher level of abstraction where the systems are similar.

A common type of difference, that should not pose large difficulties in practice, is if some modules and dependencies are similar, and the systems have some modules that are only extensions to a common architecture. For example, in the example system one of the systems could have an additional View module (say, a piano roll visualization of the music); in the industrial case we could imagine one of the systems to have a module modeling one more aspect of physics (PW) than the other. However, a simple workaround solution in the current version of the method is to introduce virtual module instances, i.e., modules that do not exist in the real system (which are of course not desired in the future system).

5. Related Work

There is much literature to be found on the topic of *software integration*. Three major fields of software integration are component-based software [16], open systems [13], and Enterprise Application Integration, EAI [15]. However, we have found no existing literature that directly addresses the context of the present research: integration or merge of software *controlled and owned within an organization*. These existing fields address somewhat different problems than ours, as these fields concern components or systems *complementing* each other rather than systems that *overlap* functionally. Also, it is typically assumed that components or systems are acquired from third parties and that modifying them is not an option, a constraint that does not apply to the in-house situation. *Software reuse* typically assumes that components are initially built to be reused in various contexts, as COTS components or as a reuse program implemented throughout an organization [7], but in our context the system components were likely not being built with reuse in mind.

It is commonly expressed that a software architecture should be documented and described according to different views [3,5,6,8]. One frequently proposed view is the module view [3,5] (or development view [8]), describing development abstractions such as layers and modules and their relationships. The dependencies between the development time artifacts were first defined by Parnas [14] and are during ordinary software evolution the natural tool to understand how modifications made to one component propagate to other.

The notion of "architectural mismatch" is well known, meaning the many types of incompatibilities that may occur when assembling components built under different assumptions and using different technologies [4]. There are some methods for automatically merging software, mainly source code [1], not least in the context of configuration management systems [12]. However, these approaches are unfeasible for merging large systems with complex requirements, functionality, quality, and stakeholder interests. The abstraction level must be higher.

6. Conclusions and Future Work

The problem of integrating and merging large complex software systems owned in-house is essentially unexplored. The method presented in this paper addresses the problem of rapidly outlining various merge alternatives, i.e., exploring how modules could be reused across existing systems to enable an evolutionary merge. The method makes visible various merge alternatives and enables reasoning about the resulting functionality of the merged system as well as about the quality attributes of interest (including both development time and runtime qualities).

The method consists of a formal model with a loosely defined heuristics-based process on top. The goal has been to keep the underlying model as simple as possible while being powerful enough to capture the events of a real industrial case. One of the main drivers during its development has been simplicity, envisioned to be used as a decision support tool at a meeting early in the integration process, with architects of the existing systems. As such, it allows rapid exploration of multiple scenarios in parallel. We have chosen the simplest possible representation of structure, the module view. For simplicity, the method in its current version mandates that the systems have identical structures. This assumption we have shown is not unreasonable but can also be worked around for minor discrepancies. The method is designed so that stepwise deliveries of the existing systems are made, sharing more and more modules, to enable a true evolutionary merge.

Assisted by a tool, it would be possible to conveniently record information concerning all decisions made during the exploration, for later processing and presentation, thus giving an advantage over only paper and pen. We are implementing such a tool, which already exist as a prototype [11]. It displays the graphs of the systems, allows user-friendly operations, highlights inconsistencies with colors, and is highly interactive to support the explorative process suggested. The information collected, in the form of short text descriptions and effort estimations, enables reasoning about subsequent implementation activities. For example, how much effort is the minimum for a first delivery where some module is shared? What parts of a stepwise delivery are only intermediate, and how much effort is thus wasted in the long term?

There are several directions for extending the method: First, understanding and bridging differences in existing data models and technology frameworks of the existing systems is crucial for success and should be part of a merge method. Second, the model could be extended to allow a certain amount of structural differences between systems. Third, the module view is intended to reveal only static dependencies, but other types of relationships are arguably important to consider in reality. Therefore, we intend to investigate how the method can be extended to include more powerful languages, including e.g., different dependency types and different adaptation types, and extended also to other views.

6.1 Acknowledgements

We would like to thank all interviewees and their organization for sharing their experiences and allowing us to publish them. Also thanks to Laurens Blankers for previous collaboration that has led to the present paper, and for our discussions on architectural compatibility.

7. References

[1] Berzins V., "Software merge: semantics of combining changes to programs", In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16, issue 6, pp. 1875-1903, 1994.

[2] Buschmann 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., *Documenting Software Architectures: Views and Beyond*, ISBN 0-201-70372-6, Addison-Wesley, 2002.

[4] 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.

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

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

[7] Karlsson E.-A., *Software Reuse : A Holistic Approach*, Wiley Series in Software Based Systems, ISBN 0 471 95819 0, John Wiley & Sons Ltd., 1995.

[8] Kruchten P., "The 4+1 View Model of Architecture", In *IEEE Software*, volume 12, issue 6, pp. 42-50, 1995.

[9] Land R., *Interviews on Software Systems Merge*, MRTC report, Mälardalen Real-Time Research Centre, Mälardalen University, 2006.

[10] Land R. and Crnkovic I., "Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", In *Information & Software Technology*, Accepted for publication, 2006.

[11] Land R. and Lakotic M., "A Tool for Exploring Software Systems Merge Alternatives", In *Proceedings of International ERCIM Workshop on Software Evolution* , 2006.

[12] Mens T., "A state-of-the-art survey on software merging", In *IEEE Transactions on Software Engineering*, volume 28, issue 5, pp. 449-462, 2002.

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

[14] Parnas D. L., "Designing Software for Ease of Extension and Contraction", In *IEEE Transaction on Software Engineering*, volume SE-5, issue 2, pp. 128-138, 1979.

[15] Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.

[16] Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.

[17] Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.