# Making Component-by-Component Selection Criteria Explicit: Enabling Stepwise Software Systems Merge

**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

*Xxx*

.

## 1. Background and Introduction

As in-house developed software systems are evolved, their initial scope and purpose may change and grow, until a point where there is some overlap in functionality and purpose. The same situation occurs, only more drastically, as a result of company acquisitions and mergers. 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.

To resolve this situation, it is perfectly possible to retire one of the existing systems and evolve the other to include some of the features of the retired system, or even to start developing a new generation of the systems and plan for retiring both the existing ones [15,17]. Reusing experience instead of implementations might be the best choice under some circumstances [18], for example if the existing systems are considered aged, or if users are dissatisfied and improvements would require major efforts [15].

There is another option: to merge the systems, by picking some parts from one, some from the other, and assemble them into a new system [15,17]. If this is possible, the potential benefits include decreased costs for development and time to delivery, as well as reduced risk in the sense that components are of known quality. If the differences between the systems are too large, it is probably not worth the effort to attempt a merge. The purpose of the present paper is to address the merge strategy by providing a method for exploring different merge alternatives early in the process.

### 1.1 Problem Context

The envisioned context of the method is the small group of architects who typically meet and outline various solutions [19,20]. Many alternatives are partly developed and evaluated until (hopefully) one or a few high-level alternatives are fully elaborated, including some estimates on effort and time required for implementation.

Merging large complex systems with limited resources can be expected to take numerous years, and there is a need to perform an evolutionary merge with stepwise deliveries [15,17]. In practice this means delivering the existing systems separately, with more and more parts being common, until some point in the future where all parts are common. This brings complexities in terms of making new components fit together with several existing ones. One would like to put as little effort as possible into modifications that are required only for an intermediate system delivery but will be obsolete in the final system. There is a delicate tradeoff between long-term and short term costs and goals with no simple solution, but the proposed method helps in exploring alternative partial deliveries and their implications.

The starting points for our proposed method are several: first, the common use of module dependency diagrams to understand propagation of changes during system evolution, second, our previous observation from multiple cases in industry that similar high-level structures seem to be a prerequisite for merge [18], and third, one particular industrial case (presented in section 4) where the work actually carried out was an informal version of the method we present.

### 1.2 The Present Paper

We first provide some definitions and introduce our proposed method by means of an example in section **Error! Reference source not found.**. Section 4 describes the events in an industrial case that supports the applicability of our method, and section 5 discusses some important observations from the case and argues for some general advices based on this. Section **Error! Reference source not found.** surveys related literature, and section 6 summarizes and concludes the paper by motivating how the goals of the paper have been met and outlining future work.

More Motivation? conceptual integrity, AOP. List problems - not here? conceptual integrity, divergence. In discussion in section 5?

## 2. Related Work

In our previous literature survey [16], we found that there are two classes of research on the topic of software integration. First, there is basic research describing integration rather fundamentally in terms of a) interfaces [12,28,29], b) architecture [1,9,11], architectural mismatch [8], and architectural patterns [3,7,25], and c) information/taxonomies/data models [10]. Second, there are three major fields of application: a) Component-Based Software Engineering [5,21,26,27], including component technologies, b) standard interfaces and open systems [21,22], and c) Enterprise Application Integration (EAI) [6,24]. These existing fields are not directly applicable to the in-house integration context, as they 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. Finally, the goals of integration in these fields are to reduce development costs and time, while in the in-house context the goals are to lower maintenance costs and present a coherent system to users and customers.

It is commonly known that a software architecture should be documented and described according to different views [4,11,13,14]. One commonly proposed view is the module view [4,11] (or development view [14]), describing development abstractions such as layers and modules and their relationships. The dependencies between the development time artifacts were first defined by Parnas [23] and are during ordinary software evolution the natural tool to understand how modifications made to one component propagate to other. The assumption underlying the present research is that dependency graphs should be a viable way to also understand the consequences of some choice during software merge.

Although there are methods for merging source code [2], we consider this to be unrealistic for the task of in-house integration of large systems with complex requirements and stakeholder interests. The abstraction level must be higher.

Evolution..

Degradation/deterioration.

SA:

- Arch styles/patterns.
- ADLs (including UML)

Merge of source code [Berzins?][Bosch]

Arch mismatch [Garlan] – relate to this

Take from Laurens paper, the idea of detecting mismatches with a tool.

If the merge process is largely unexplored, a little more is known about the types of incompatibilities that may occur when assembling components built under different assumptions and using different technologies [8]**[Laurens' survey]**.

## 3. Presenting the Method

The method consists of two parts that are presented in this section: a model, i.e. a set of formal concepts and definitions, and a process, i.e. a set of human activities that utilizes the model. The model is designed to be simple but reflect reality as well as possible, and the process describes higher-level reasoning and heuristics that are suggested as useful practices in many cases.

Figure 1 describes an example of two computer car games that after a company merger are candidates for merge. The figure describes a scenario where some modifications have been made – initially, all modules of System A would be denoted $\alpha_A$, and all of System B $\alpha_B$. The figure is further explained in the rest of the present section, as it is used to exemplify the method.

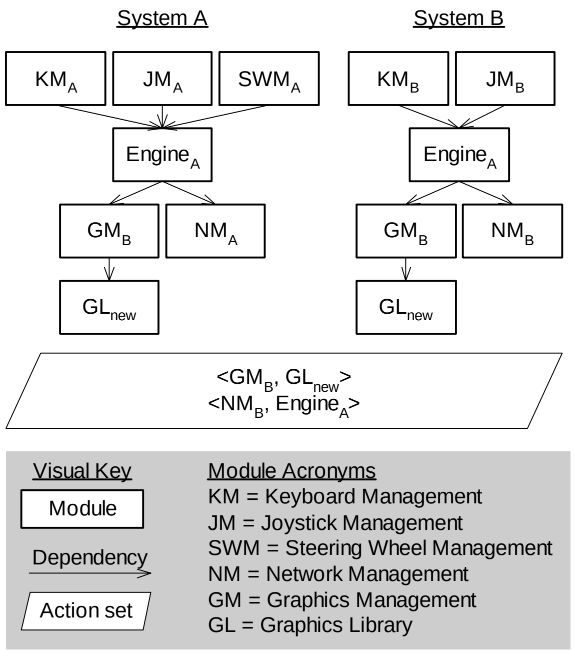

**Figure 1: Two example systems.**

### 3.1 The Underlying Model

Our proposed method builds on a model consisting of three parts, presented and defined below: a set of model elements, an inconsistency pattern defined in terms of these structures, and a set of permissible user operations. Once the systems are described in terms of the model elements, users can apply any of the possible operations, in any order, which includes annotating the structures with information how to solve the individual inconsistencies. Whenever the pattern is no longer found, the model represents systems that are internally consistent given that the annotated actions are taken.

#### 3.1.1 Concepts and Notation

The following concepts are used in the model:

- Existing *systems* are denoted by capital letters A, B, etc.
- *Module roles* represent conceptual system parts that have been identified in the existing systems, such as "Engine" or "Graphics Library". We assume a substantial overlap between the module roles of the existing systems. Module roles are written with capital first letter, and parameterized by Greek letters $\alpha$ and $\beta$.
- A *module instance* corresponds to a concrete realization of a module role, and is denoted $\alpha_X$ where *X* is either a system (A, B, etc.) meaning that the module instance is the one that already exist in that system (e.g. $\text{Engine}_A$), or a "new" marker, representing a module that is new to the systems. Such a new module could be either a non-existing, planned implementation, or an already existing module, to be reused in-house from some other program or a commercial component. A module instance is denoted by its module role subscribed by the origin marker, e.g. $\text{Engine}_A$ and $\beta_{new}$.
- A *action* is a pair of module instances. The pair $\langle\alpha_X, \beta_Y\rangle$ represents that $\alpha_X$ is modified to be consistent with $\beta_Y$.
- A *dependency graph* captures the structure of a system. It is a directed graph where each node in the graph is a module instance and each edge $\alpha_X \rightarrow \beta_Y$ represents a dependency from $\alpha_X$ to $\beta_Y$. In Figure 1, we have for example the dependencies $\text{JM}_A \rightarrow \text{Engine}_A$ and $\text{GM}_B \rightarrow \text{GL}_{new}$. Graphically, nodes with the same module role are written together inside a dashed box.
- A *scenario* consists of a dependency graph for each existing system and a single set of actions. A *consistent scenario* is a scenario without inconsistencies (see definition below in section 3.1.2).
- The full *model* consists of a set of scenarios and an information database with general information related to individual actions and development of new module instances (including effort estimates). This information is not associated with a particular scenario, but can be used to derive scenario-specific information, e.g. the estimated total effort associated with the modifications in a given scenario. We envision that any particular project or tool would define its own formats and types of information. At the least, there would be short textual descriptions of what each action means in practice. We can note that a model might contain several different new modules for the same role $\alpha$, used in different scenarios, such as $\alpha_{\text{new implementation}}$, $\alpha_{\text{open source}}$, $\alpha_{\text{commercial}}$, etc.

Figure 1 depicts a scenario where the Engine from system A and the GM module from system B are used in both systems, and a new GL module is planned to be developed or acquired and is used in both systems.

### 3.1.2 Inconsistency

An inconsistency means that two modules, one dependent on the other, are not functioning together. Trivially, two modules from the same systemare consistent with no further action. Two modules from different systems are consistent only if some measure has been taken to, i.e. if either module have been modified to work with the other.

Formally, a dependency from $\alpha_X$ to $\beta_Y$ is *consistent* if $X = Y$ or if the action set contains $\langle\alpha_X, \beta_Y\rangle$ or $\langle\beta_Y, \alpha_X\rangle$. Otherwise, the dependency is *inconsistent*.

Example: The scenario in Figure 1 is inconsistent, because of the inconsistent dependencies from KMB and JMB to $\text{Engine}_A$, and from $\text{Engine}_A$ to $\text{GM}_B$. The dependencies from $\text{GM}_B$ to $\text{GL}_{new}$ and from $\text{Engine}_A$ to $\text{NM}_B$ on the other hand are consistent, as there are actions $\langle\text{GM}_B, \text{GL}_{new}\rangle$ and $\langle\text{NM}_B, \text{Engine}_A\rangle$ meaning that $\text{GM}_B$ and $\text{NM}_B$ have been modified to be consistent with $\text{GL}_{new}$ and $\text{Engine}_A$ respectively.

### 3.1.3 Scenario Operations

The following operations can be performed on a scenario:

1. Add or remove a action to/from the action set.
2. Add the module instance $\alpha_X$ to one of the dependency graphs.
3. Remove the module instance $\alpha_X$ from one of the dependency graphs, if it does not participate in any dependencies.
4. Replace the dependency $\alpha_X \rightarrow \beta_Y$ by $\alpha_X \rightarrow \beta_Z$, if $\beta_Z$ exists in the graph.
5. Replace the dependency $\alpha_X \leftarrow \beta_Y$ by $\alpha_X \leftarrow \beta_Z$, if $\beta_Z$ exists in the graph.

Note that these operations never change the participating module roles of the graph, nor the dependencies between them. Note also that we allow two instances for the same role in a system; whether this is suitable for a real system.

## 3.2 Suggested Process

The suggested process consists of two phases, the first consisting of two simple preparatory activities, and the second being recursive and exploratory.

### 3.2.1 Preparatory Phase

The *Preparatory* phase consists of two activities:

P-I: Describe Existing Systems

P-II: Describe Desired Future Architecture

**Activity P-I: Describe Existing Systems**

First, the dependency graphs of the existing systems must be created, and common module roles identified. This activity could and should arguably be kept informal as it occurs early, meaning that the people meet for the first time. We do not suggest any particular systematic method to arrive at these descriptions.

**Activity P-II: Describe Desired Future Architecture**

The dependency graph of the future system is the union of the graphs depicting the existing systems, i.e. where each existing role is present and each dependency is present. It is assumed that the systems show a considerable degree of similarity, so there is no need to formalize this further here (this assumption is further discussed in section 5.x).

Not any module instance is desired in the future system. For some roles it is imperative to use some specific instance (e.g. $\alpha_X$ because it is superior to $\alpha_Y$, or a new implementation $\alpha_{\text{new}}$ because there have been problems with the existing $\alpha_X$, $\alpha_Y$). For other roles, $\alpha_X$ might be preferred over $\alpha_Y$, 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.

### 3.2.2 Exploratory Phase

Initially, the model contains a single scenario corresponding to the structure and contents (?) of the existing systems. The exploratory phase can be described in terms of four activities, but the order between them is not pre-determined. Any activity could be performed after any of the others, but they are 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 wants 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, and towards the end when there are a number of consistent scenarios there will be an emphasis on evaluating these deliveries of the existing systems. It should once again be noted that these activities describe high-level things that are often useful to do, 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), and sometimes there might be several possible starting points. The choice may e.g. depend on the local priorities for each system (e.g. "we need to improve the Engine of system A"), and/or some strategic considerations concerning how to make the envisioned merge succeed (e.g. "the Engine should be made a common component as soon as possible").

**Activity E-II: Resolve Inconsistencies**

As components are exchanged in the graphs, there will be mismatches between modules $\alpha_X \rightarrow \beta_Y$ that need to be resolved. There are three main ways of resolving these:

1. Exchanging either module instance for another by combining operations 2, 4, 5 and 3, so that the new pair of components are consistent, i.e. either exchange $\alpha_X$ for $\alpha_Y$, or $\beta_Y$ for $\beta_X$. In Figure 1, the inconsistent dependency between $\text{Engine}_A$ and $\text{GM}_B$ could be solved by changing $\text{GM}_B$ for $\text{GM}_A$ in both systems.
2. Modifying either module to be consistent with the interface of the other, i.e. adding an action to the action set (operation 1). In the example, this means adding either the action $\langle \text{GM}_B, \text{Engine}_A \rangle$ or $\langle \text{Engine}_A, \text{GM}_B \rangle$ to the action set. The actual modification could be of many different kinds, depending on the system domain and the type of module, e.g. adapters, modifying lines of source code throughout, etc. This is further discussed… This information should be added to the model information database for the particular action.
3. To introduce one more module instance for one of the roles, to exist in parallel with the existing. (This could be suitable for some library with many using modules, where using both libraries in parallel does not cause any problems. This is discussed further in section 5.x.)

These introduced changes will likely cause new inconsistencies that need to be resolved (i.e. this activity need to be performed again).

**Activity E-III: Branching 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. For example, two of the ways to resolve an inconsistence might make sense, and both choices should be explored. It is possible to copy the scenario and treat them as branches coming from different choices.

(At this point, it could be noted that we have avoided formalizing operations such as create, delete, copy, or branch scenarios, because this is not essential neither to the model nor the process. In a tool implementation, one is free to visualize the information for example in several ways, that facilitate the users' work, . (unordered) set of actions as a tree of choices, with some branches.

In the example, …

**Activity E-IV: Evaluate Scenarios**

As scenarios evolve, they need to be evaluated in order to decide which branch to evolve further and which to abandon. Towards the end of the process, one probably wants to evaluate the final alternatives more thoroughly, and compare them. There seems to be at least two evaluation criteria for which data can be inferred directly from the model:

1. The actual state of the systems (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 (highest quality etc.)?
2. The work that needs to be done in order to arrive at the final scenario, i.e. the sum of all individual modification efforts (and new development effort for new components). This requires that the effort for each action has been added to the information database.

It also becomes possible to extract all information entered concerning the actions needed, such as short notes outlining what each modification means in practice, into a draft project plan.

In addition, it becomes possible to reason about how much of the efforts required 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?