# A Tool for Exploring Software Systems Merge Alternatives

**Rikard Land[1], Miroslav Lakotic[2]**
[1]*Mälardalen University, Department of Computer Science and Electronics*
*PO Box 883, SE-721 23 Västerås, Sweden*
[2]*University of Zagreb, Faculty of Electrical Engineering and Computing*
*Unska 3, HR-10000 Zagreb, Croatia*
*rikard.land@mdh.se, miroslav.lakotic@fer.hr, http://www.idt.mdh.se/~rld*

## Abstract

*The present paper presents a tool for exploring different ways of merging software systems, which may be one way of resolving the situation when an organization is in control of functionally overlapping systems. It uses dependency graphs of the existing systems and allows intuitive exploration and evaluation of several alternatives.*

## 1. Introduction

It is well known that successful software systems has to evolve to stay successful, i.e. it is modified in various ways and released anew [11,15,16]. Some modification requests concern error removal; others are extensions or quality improvements. A current trend is to include more possibilities for integration and interoperability with other software systems. Typical means for achieving this is by supporting open or de facto standards [13] or (in the domain of enterprise information systems) through middleware [4]. This type of integration concerns information exchange between systems of mainly *complementary functionality*. There is however an important area of software systems integration that has so far been little researched, namely of systems that are developed in-house and *overlap functionally*. This may occur when systems, although initially addressing different problems, evolve and grow to include richer and richer functionality. More drastically, this also happens after company acquisitions and mergers, or other types of close collaborations between organizations. 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.

### 1.1 Background Research

To investigate how organizations have addressed this challenge, which we have labeled *in-house integration*, we have previously performed a qualitative multiple case study [21] consisting of nine cases in six organizations.

At a high level, there seems to be four strategies that are analytically easy to understand [10]: *No Integration* (i.e. do nothing), *Start from Scratch* (i.e. initiate development of a replacing system, and plan for retiring the existing ones), *Choose One* (choose the existing system that is most satisfactory and evolve it while planning for retiring the others), and – the focus of the present paper – *Merge* (take components from several of the existing systems, modify them to make them fit and reassemble them).

There may be several reasons for not attempting a *Merge*, for example if the existing systems are considered aged, or if users are dissatisfied and improvements would require major efforts. Reusing experience instead of implementations might then be the best choice. Nevertheless, *Merge* is a tempting possibility, because users and customers from the previous systems would feel at home with the new system, no or very little effort would be spent on new development (only on modifications), and the risk would be reduced in the sense that components are of known quality. It would also be possible to perform the *Merge* in an evolutionary manner by evolving the existing systems so that more and more parts are shared; this might be a necessity to sustain commitment and focus of the integration project. Among the nine cases of the case study, only in one case was the *Merge* clearly chosen as the overall strategy and has also made some progress, although there were elements of reuse between existing systems also in some of the other cases. Given this background research, we considered the *Merge* strategy to be the least researched and understood and the least performed in practice, as well as the most intellectually challenging.

## 1.2 Continuing with Merge

To explore the *Merge* strategy further, we returned to one of the cases and performed follow-up interviews focused on compatibility and the reasons for choosing one or the other component. The organizational context is a US-based global company that acquired a slightly smaller global company in the same business domain, based in Sweden. The company conducts physics computer simulations as part of their core business, and both sites have developed their own 3D physics simulator software systems. Both systems are written in Fortran and consist of several hundreds of thousands lines of code, a large part of which are a number of physics models, each modeling a different kind of physics. The staff responsible for evolving these simulators is less than a handful on each site, and interviews with these people are our main source of information [9].

At both sites, there were problems with their model for a particular kind of physics, and both sites had plans to improve it significantly (independent of the merge). There was a strategic decision to integrate or merge the systems in the long term, the starting point being this specific physics module. This study involved interviewing more people. It should be noted that although the interviewees met in a small group to discuss alternatives, they did not use our tool, since the tool has been created after, and partly influenced by, these events. The case is nevertheless used as an example throughout the present paper, to illustrate both the possibilities of the tool and motivate its usefulness in practice.

In an in-house integration project, there is typically a small group of architects who meet and outline various solutions [10]. This was true for the mentioned case as well as several others in the previous study. In this early phase, variants of the *Merge* strategy should be explored, elaborated, and evaluated. The rest of the paper describes how the tool is designed to be used in this context. The tool is not intended to automatically analyze or generate any parts of the real systems, only serve as a decision support tool used mainly during a few days' meeting. One important design goal has therefore been simplicity, and it can be seen as an electronic version of a whiteboard or pen-end-paper used during discussions, although with some advantages as we will show.

## 1.3 Related Work

Although the field of software evolution has been maturing since the seventies [11,16], there is no literature to be found on software in-house integration and merge. Software integration as published in literature can roughly be classified into: Component-Based Software Engineering [19,20], b) standard interfaces and open systems [13], and c) Enterprise Application Integration (EAI) [6,18]. These fields typically assume that components or systems are acquired from third parties and that modifying them is not an option, which is not true in the in-house situation. Also, these fields address components or systems *complementing* each other (with the goal of to reducing development costs and time) rather than systems that *overlap* functionally (with rationalization of maintenance as an important goal).

Although there are methods for merging source code [3,12], these approaches are unfeasible for merging large systems with complex requirements, functionality, quality, and stakeholder interests. The abstraction level must be higher.

We have chosen to implement a simple architectural view, the module view [5,7] (or development view [8]), which is used to describe development abstractions such as layers and modules and their relationships. Such dependency graphs, first defined by Parnas [14], are during ordinary software evolution the natural tool to understand how modifications propagate throughout a system.

## 2. The Tool

The tool was developed by students as part of a project course. The foundation of the tool is a method for software merge. As this is ongoing work, this paper is structured according to the method but focuses on the tool. We also intend to publish the method separately, as it has been refined during the tool implementation – after which it is time to further improve the tool.

The method makes use of dependency graphs of the existing systems. There is a formal model at the core, with a loosely defined process on top based on heuristics and providing some useful higher-level operations. The tool conceptually makes the same distinction: there are the formally defined concepts and operations which cannot be violated, as well as higher-level operations and ways of visualizing the model, as suggested by the informal process. In this manner, the user is gently guided towards certain choices, but never forced. A fundamental idea with both the method and the tool is that they should support the exploratory way of working – not hinder it.

The actual tool is implemented as an Eclipse plug-in [1]. The model of the tool is based on the formal model mentioned above, and its design follows the same rules and constraints. The model was made using Eclipse Modeling Framework, and presented by Graphics Eclipse Framework combined using the Model-Controller-View architecture. This makes the tool adaptable and upgradeable.

## 2.1 Preparatory Phase

There are two preparatory activities:

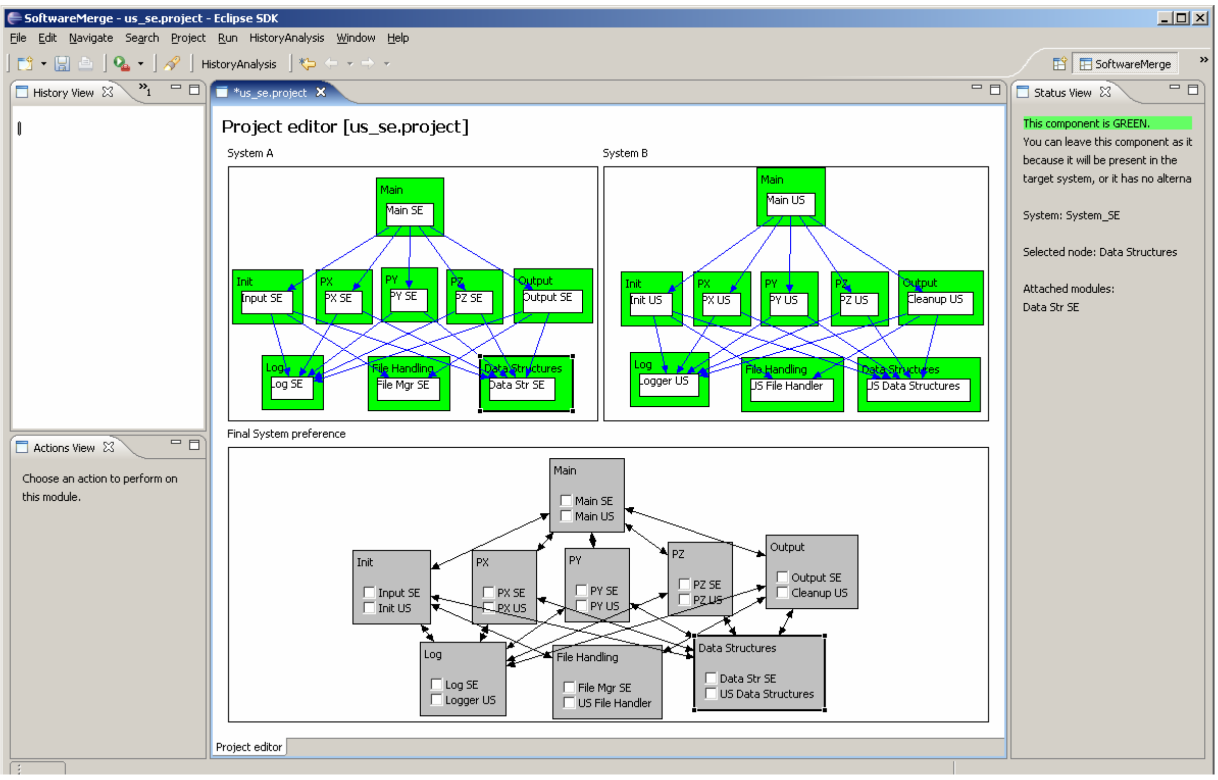
**Activity P-I: Describe Existing Systems.** The user first needs to describe the existing systems as well as outline a desired future system. The current implementation supports two existing systems, but the underlying model is not limited to only two.

**Activity P-II: Describe Desired Future Architecture.** The suggestion of the final system is determined simply by choosing which modules are preferred in the outcome. Any system, A or B can then be experimented upon, and the progress can be followed through a scenario tree. Figure 1 shows a snapshot of the tool with the two existing systems at the top and the future system at the bottom. It might be noted that the existing systems have – and must have – identical structures (this assumption is further discussed in section 2.3).

## 2.2 Exploratory Phase

The goal of the exploration is two system descriptions where some modules have been exchanged, so that the systems are evolved in parallel towards the desired future, merged system. The goal is not only to describe the future system (one graph would then be enough, and no tool support needed) but to arrive at next releases of the systems, in order to perform the merge gradually, as a sequence of parallel releases of the two existing systems until they are identical. This will involve many tradeoffs on the behalf of the architects (and other stakeholders) between e.g. efforts to be spent only on making things fit for the next release and more effort to include the more desired modules, which will delay next release of a system. The tool does not solve these tradeoffs but supports reasoning about them. There are four activities defined in the exploratory phase, with a rough ordering as follows, but also a number of iterations.

**Activity E-I: Introduce Desired Changes.** The starting point for exploration is to introduce some desired change. In the case, it was imperative to start by assuming a newly developed physics module (PX in the figures) to be shared by both systems. In other situations, the actual module to start with might not be given. In the tool, this is done by choosing the preferred module in the final system view, by clicking on the checkboxes. A new module can also be attached to the old system. This is done by clicking on the node in final system, and then clicking on the button "Create" in the *Actions View*. This will also require user input for the name of the new module and effort needed for its implementation (this could be zero for a pre-existing component such as a commercial or open source component, or a component to be reused in-house). After the module has been created, it can be used as any other module. The change to the system



Choose an action to perform on this module.

This component is GREEN.
You can leave this component as it because it will be present in the target system, or it has no alterna

System: System_SE

Selected node: Data Structures

Attached modules:
Data Str SE

**Figure 1: Initial systems state.**

structure is made by clicking on the nodes and links in the input systems A and B. The modules the systems are using can be set up in the *Status View* for every node in any input system.

**Activity E-II: Resolve Inconsistencies.** As changes are introduced, the tool will highlight *inconsistencies* between modules by painting the dependency arrows orange (see Figure 2). In the model, two module instances from the same system are consistent without further adaptation. Two modules from different systems are consistent only if some measure has been taken to ensure it, i.e., if either module have been adapted to work with the other. The actual adaptations made could in practice be of many kinds: some wrapping or bridging code as well as modifications of individual lines of code.

Another way to resolve an inconsistency is to describe adaptations to either of the inconsistent modules, in order to make them match. This is done by clicking on the incompatible link, and one of "Add …" buttons in the *Actions View*. This will require the user to enter an estimated effort for resolving this inconsistency (a number, in e.g. man-months), and a free text comment how to solve it, such as "we will modify each call to methods *x()* and *y()*, and must also introduce some new variables *z* and *w*, and do the current *v* algorithm in a different way" (on some level of detail found feasible). (As said, the tool does not do anything with the real systems automatically, but in this sense serves as a notebook during rapid explorations and discussions.) It can be noted that a module that will be newly developed would be built to fit. Nevertheless there is an additional complexity in building something to fit two systems simultaneously, which is captured by this mechanism.

There is also a third possibility to resolve an inconsistency: to let two modules for the same role live side by side, see Figure 3. Although allowing the same thing to be done in different ways is clearly a violation of the system's conceptual integrity, it could be allowed during a transition period (until the final, merged system is delivered) if the system's correct behavior can be asserted. For example, it might be allowed for some stateless fundamental libraries, but not when it is fundamentally assumed that there is only one single instance responsible for a certain functionality, e.g. for managing central resources, such as thread creation and allocation, access control to various hardware or software resources, security). The tool cannot know whether it would be feasible in the real system, this is up to the users to decide when and whether to use this possibility. The current version does not model the potential need for communication and synchronization of two modules doing same role.

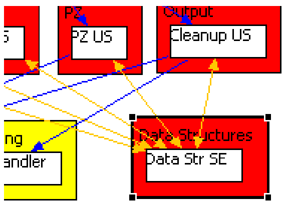


Figure 2: Example of highlighted inconsistencies.

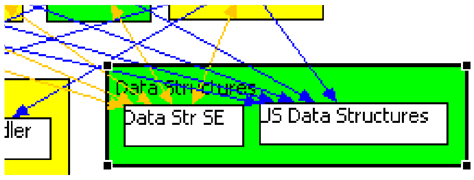


Figure 3: Two modules with same role.

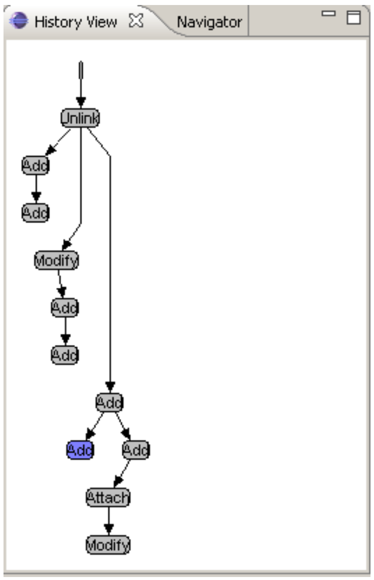
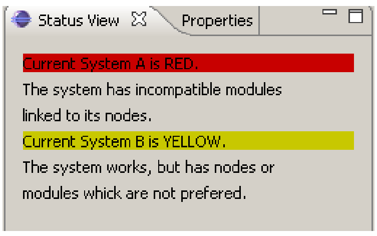


Figure 4: The History View.

**Activity E-III: Branch Scenarios.** As changes are made, the operations are added to a scenario tree in the *History View* (see Figure 4). At any time, it is possible to click any choice made earlier in the tree, and branch a new scenario from that point. The leaf of each branch represents one possible version of the system. When clicking on a node, the graphs are updated to reflect the particular decisions leading to that node. Any change to the systems (adaptations, exchanging modules, etc.) results in a new node being created; unless the currently selected node is a leaf node, this means a new branch is created. All data for adaptations entered are however shared between scenarios; this means that the second time a particular inconsistency is about to be resolved, the previous description and effort estimation will be used. As information is accumulated, the exploration will be more and more rapid.



Current System A is RED.
The system has incompatible modules linked to its nodes.
Current System B is YELLOW.
The system works, but has nodes or modules which are not prefered.

**Figure 5: The Status View.**

**Activity E-IV: Evaluate Scenarios.** The exploration is a continuous iteration between changes being made (activities E-II and E-III) and evaluation of the systems. Apart from the information of the graphs themselves, the *Status View* presents some additional information, see Figure 5. The branching mechanism thus allow the architects to try various ways of resolving inconsistencies, undo some changes (but not loosing them) and explore several alternatives in a semi-parallel fashion, abandon the least promising branches and evaluate and refine others further. The total effort for an alternative can be accessed by clicking the "History Analysis" button, which is simply the sum of all individual adaptation efforts. It also becomes possible to reason about efforts related to modifications that actually lead towards the desired future system, efforts required only to make modules fit only for the next delivery (and later discarded).

The tool's advantage over using a whiteboard lies in the possibility to switch back and forth among (temporary) decisions made during the exploration (by means of the scenario tree), make some further changes (through simple point-and-click operations), and constantly evaluate the resulting systems (by viewing the graphs, the status view, and retrieve the total effort for the scenario).

Finally, although not implemented yet, one would extract the free texts associated with the scenario into a list of implementation activities.

### 2.3 Similar Structures?

The tool (and the model) assumes that the existing systems have identical structures, i.e. the same set of module roles (e.g. one module instance each for file handling, for physics X etc.) with the same dependencies between them. This may seem a rather strong assumption, but there are three motivations for this, based on our previous multiple case study [10]. First, our previous observations strongly suggest that similar structures are a prerequisite for merge to make sense in practice. Second, we also observed that it is not so unlikely that systems in the same domain, built during the same era, are indeed similar. And third, if the structures are not very similar, it is often possible to find a higher level of abstraction where the systems are similar.

With many structural differences, *Merge* is less likely to be practically and economically feasible, and some other high-level integration strategy should be chosen (i.e. *Start from Scratch* or *Choose One*). A common type of difference, that should not pose large difficulties in practice, is if there is a set of identical module roles and dependencies, and some additional modules that are only extensions to this common architecture. (For example, in the case we could imagine one of the systems to have a module modeling one more physics model PW than the other.) However, architects need in reality not be limited by the current version: a simple workaround solution 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).

## 3. Future Research & Development

The tool is still in prototype stage and needs to be further developed. Neither the method nor the tool has been validated in a real industrial case (although their construction builds heavily on industrial experiences).

In reality there are numerous ways to make two components fit, for example as an adapter mimicking some existing interface (which requires little or no modifications of the existing code) or switches scattered through the source code (as runtime mechanisms or compile-time switches). Such choices must be considered by the architects: a high-performance application and/or a resource constrained runtime environment might not permit the extra overhead of runtime adapters, and many compile-time switches scattered throughout the code makes it difficult to understand. The method in its current version does not model these choices explicitly but has a very rough representation: the users can select which of the two inconsistent modules that should be adapted, and add a free text description and an effort estimation.

Another type of extension would be to include several structural views of the architecture, including some runtime view.

Yet another broad research direction is to extend the method and the tool to not focus so much on structure as the software architecture field usually does [2,17]. Structure is only one high-level measure of similarity between systems. Existing data models, and the technological frameworks chosen (in the sense "environment defining components") are also important additional issues to evaluate [10], and needs to be included in any merge discussions in reality, and should be included in future extensions of the merge method and the tool.

## 4. Acknowledgements

We would like to thank the interviewees and their organization for sharing their experiences and allowing us to publish them. Thanks to Mathias Alexandersson, Sebastien Bourgeois, Marko Buražin, Mladen Čikara, Lei Liu, and Marko Pecić for implementing the tool. Also thanks to Laurens Blankers, Jan Carlson, Ivica Crnkovic, and Stig Larsson for previous and current research collaborations related to this paper.

## 5. References

[1] *Eclipse.org home*, URL: www.eclipse.org, 2006.

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

[3] 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.

[4] Britton C. and Bye P., *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems* (2nd edition), ISBN 0321246942, Pearson Education, 2004.

[5] 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.

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

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

[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 ISSN 1404-3041 ISRN MDH-MRTC-196/2006-1-SE, 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] Lehman M. M. and Ramil J. F., "Software Evolution and Software Evolution Processes", In *Annals of Software Engineering*, volume 14, issue 1-4, pp. 275-309, 2002.

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

[16] Perry D. E., "Laws and principles of evolution", In *Proceedings of International Conference on Software Maintenance (ICSM)*, pp. 70-70, IEEE, 2002.

[17] 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.

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

[19] Szyperski C., *Component Software - Beyond Object-Oriented Programming* (2nd edition), ISBN 0-201-74572-0, Addison-Wesley, 2002.

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

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