# How Can We Help Software Rearchitecting Efforts? Study of an Industrial Case

Brice Govin, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Arnaud Monegier

# How Can We Help Software Rearchitecting Efforts? Study of an Industrial Case

Brice Govin* †, Nicolas Anquetil †, Anne Etien †, Stephane Ducasse †, Arnaud Monegier *

* THALES AIR SYSTEMS, Parc tertiaire SILIC, 3 Avenue Charles Lindberg 94628 Rungis Cedex, France
{brice.govin, arnaud.monegier}@thalesgroup.com
† RMod team, Inria Lille Nord Europe, University Lille 1, CRIStAL, UMR 9189, 59650 Villeneuve d'Ascq, France
{brice.govin, nicolas.anquetil, anne.etien, stephane.ducasse}@inria.fr

***Abstract*—Legacy software systems are valuable assets for organisations and are sometimes their main source of incomes. From time to time, renewing legacy software system architecture becomes necessary in order to offer them a new future. Migrating the architecture of a legacy software system is a difficult task. It involves understanding and aggregating a large set of data (the entire source code, dependencies, *etc.*); it may have a profound impact on the system's behaviour; and because it occurs very rarely in the life of a system, it is hard to gain experience in this domain. Based on the study of an industrial architecture migration case, we discuss how this essentially manual effort could be helped with automated tools and a better defined process. We identified several issues raised during the task, characterized their impact, and proposed possible solutions.**

## I. Introduction

Legacy software systems represent a significant part of companies' wealth. Maintainability of such systems has become an important goal for companies which own them. Software evolution activities can be classified in two: day to day evolution, where engineers correct bugs, implement new features, or adapt the system to new working conditions; and, much rarer, restructuring, where engineers need to completely re-think the architecture of a system because it seems no longer possible to adapt it in small steps. Because it occurs so rarely in the life of the system, restructuring software architecture is mostly a case-by-case activity. It is difficult to gain much experience in it and to improve its practice through the definition of a formal process.

In our specific case, the company we are working with wants to restructure a real-time, embedded, critical system. The system is an old ($>$ 20 years) Ada system in the defence area. The goal is to modularise the system to ease day to day evolution and take advantage of technological improvements that were introduced since the creation of the system. The wished architecture is partially specified in the sense that high level components and main architectural rules have been defined. In this restructuring, there is no immediate change in the functionalities and one goal is to reuse existing code as much as possible.

The classical process for architecture restructuring is the horseshoe "process" [1]. However, this is a very generic one that defines high level operations. As such it has been specialised in various publications (*e.g.* [2]). A line of research try to generalise the target system into a Software Product Line [2] by abstracting variation points in the functionalities, this is not our case since we are keeping the same functionalities. Others focus on migrating toward a Service-Oriented Architecture [3], which, again is not our case, since it has been decided to go for a component based architecture. The Component Recovery line of research is an old one (e.g. [4], [5]) but it typically only tries to extract the components without allowing to specify a wished architecture. Confronted to architecture restructuring in a real software architecture migration case, we found that existing literature often relies on hypotheses that were not true in our case.

In this paper we discuss a three steps process that we formalised from the activities of the software engineers in the project. We list the difficulties that arised when trying to automate some of these activities and possible solutions that we experimented. The efficacity of the different experiments are evaluated against an oracle resulting from the work of software engineers and ourselves on a part of the system.

Section II describes the problem of software architecture migration and the potential related solutions in the literature. Section III details the actions of the software engineers performing the restructuring and points out weaknesses of this project. Section IV presents a process formalising the actions taken by the software engineers and ideas to (partly) automate it. Finally, section VI draws lessons learned from this project and explains the future work.

## II. Background

As stated previously, the classical process for architecture restructuring is the Horseshoe process [1]. It is divided into three main parts: reverse engineering, re-engineering and forward engineering. The reverse engineering part aims to model and understand the current system architecture, the re-engineering part aims to restructure the architecture of the system and the forward engineering part aims to re-implement the new architecture in the source code [6]. In our project, the reverse engineering part is already handled by the Moose infrastructure [7]. This offers us a model of the system with all its software elements (packages, procedures, types, variables) and their interdependencies (use of variables, invocation of methods, etc.). We will therefore focus on the re-engineering

part. We found in the literature different approaches that can tackle this part, *e.g.* Software Architecture Transformation [8]; migration towards Service-Oriented Architecture (SOA) [3]; and Component Recovery [9].

### A. Main Lines of Research

The phrase Software Architecture Transformation is defined in [1] as the line of research that aims at transforming the current architecture of a legacy system into a new one. Transformation between old and new architecture is done thanks to a set of "recipes" transformation between elements from the two different architectures [10]. This domain focuses on monitoring the impact of architectural changes on the overall quality of the system. It also only considers small architectural transformations that can be applied automatically and guarantees the executability of the resulting source code. This line of research does not concern us since the project we are involved with will completely restructure the system around an architecture radically different from the existing one.

Migration towards Service-Oriented Architecture (SOA) aims to "move the legacy system architecture to the more flexible SOA while preserving the original system data and functionality" [11]. SOA migration approaches either address the technical perspective to expose legacy code as services, or determine the migration feasibility regarding the characteristics of the legacy system and the requirements of the target SOA system [12]. We are not interested in the technical aspects because, in our case, the current and future systems are both implemented using the same technologies and we foresee no technical difficulties there. Moreover, components and services, even though they are close, have fundamental differences that would make it very difficult to reuse SOA migration approaches for component architecture migration.

On the other hand, SOA migration work, *e.g.* [11], [12], [13], introduced interesting techniques such as Concept Slicing to associate the existing source code with the services they extract. The idea behind Concept Slicing is to identify the source code associated with the implementation of a human-oriented "concept". The human oriented concept thus becomes a service and the code associated to it is used to implement that service. Concept slicing works by combining Concept Assignment solutions, that identify *relevant* pieces of code from a concept, with Program Slicing solution, that identify an *executable* piece of code.

A last related line of research named Component Recovery, or Component Identification, automatically organises the software elements of a system into components. Component recovery corresponds to the detection of subsystems [14], it works by clustering the nodes of a dependency graph of the application [4], [5], [15] into independent components. Yet because they work on structural information (a graph of dependencies between the elements of the system), they can only extract structurally coherent components whereas they should include human and domain knowledge in the process to be able to extract business components [16], [17]. Another issue with existing Component Recovery techniques is that they do not accept a predefined target architecture, but focus on extracting what they consider the best possible components. In our case, the target architecture is already partly defined and we already know what the main components (more abstract ones) should be and even for some of them what sub-components they should contain. We are not aware of a Component Recovery approach that would suit this requirement.

### B. Extraction of components from existing source code

Although the existing Component Recovery approaches do not apply to our project, they offer some useful techniques.

All the techniques are based on the idea of representing the source code as a dependency graph where the nodes are software elements and the vertices are dependencies between them [4], [5], [15]. Any given approach will work at a specific level of abstraction and consider a small number of software element kinds, Koschke [5] calls them *Quarks*, "the smallest significant element at the architectural level". For example, Allier and Seriai [4], [15] work with classes and Koschke [5] with subprograms, user defined types and variables (in a C program). On the other hand, the example of SOA migration shows that it might be necessary to work at a very fine-grained level. This is the case when program slicing is used to extract executable services.

From the dependency graph, software elements are clustered to form components. The clustering tries to maximise dependencies within a component (known as Cohesion) and minimise dependencies between components (known as Coupling). Many different metrics have been proposed to compute Cohesion and Coupling and we will not consider them in this paper.

Finally we should add that Johnson [18] noted the necessity of an iterative process.

## III. Description of the Project

We are accompanying a large company which undertook a big software migration project where the architecture of the subject system will be completely redesigned. This section describes the project, Section III-A explains the context of the project; Section III-B details the informal process followed by the re-engineers. Section III-C points out the weaknesses of this informal process.

### A. Context of the Migration

The migration project is to restructure a real-time, embedded, critical system in the defence area. The system is old ($>$ 20 years), big ($>$ 300 KLOC) written in Ada 83. It counts around $1\,500$ packages and $15\,000$ subprograms gathered into modules. Table I summarises main characteristics of this system and Figure 1 gives a general idea of the existing software architecture. It should be noted that this existing architecture was not explicit prior to the project and has been reconstructed as part of a reverse engineering effort. This reverse engineering effort has been done by engineers inside the company and none of the authors have participated.

TABLE I
CHARACTERISTICS OF THE SOFTWARE SYSTEM

| | | |
|---|---|---|
| Technical Characteristic | Age | >20 years |
| | Size | >300 kLoC |
| | Programming Language | Ada 83 |
| | #Package | 1 537 |
| | #Subprograms | 14 650 |
| | Specificity | Embedded, Real Time, Critical |
| Design Characteristic | Architecture | Modules with single responsibility<br>Treatment based |
| | Communication System | Message exchanges |

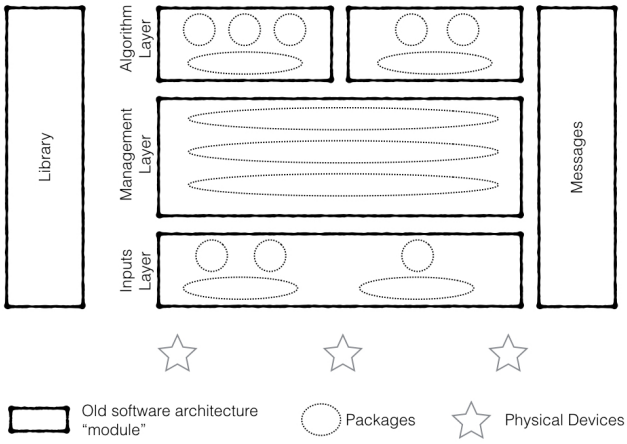


Fig. 1. Simplified representation of the existing software architecture

The problems identified with this system are:

- This is a real time system developed a long time ago with the best techniques of that time. As such, a strong emphasis was placed on the processing with little consideration for a high level data model. The company now wishes to correct this situation by defining and using high level, business oriented, data structure that could possibly be shared with other systems in the company.
- The legacy system uses a communication system, that is a proprietary middleware, based on messages exchanges. Since its installation, new communication standards have been defined, that the company wants to use. Instead of using an event based communication system (messages), interactions should be based on shared data.
- Changes in the physical environment of the system are foreseen in the near future, and there is a concern whether these changes could be easily accommodated in the existing system. For example, the Algorithm layer (Figure 1) regularly interacts directly with the physical devices. The company now wishes to have a more modular system that will constraint the impact of any of these changes to a small portion of the system.
- The legacy system adopts a layered architecture. However, communication and dependencies between layers are frequent and not only between two adjacent layers.
- Ultimately, there is a hope to be able to reuse some of the components of the target architecture in other systems of the company in the same business domain. Again, this implies having a more modular system, with better-defined components and data structures.

With these requirements in view, some high level components were defined by domain experts and structured in the target architecture sketched out in Figure 2. The target architecture is only partially specified in the sense that these high level components and main architectural rules were defined, but the exact content of the composite components (*i.e.* the small boxes within the big ones) was not clear at the start of the project. In this restructuring, all functionalities should remain the same. One goal is to reuse existing code as much as possible and what parts of the existing code should map to what components (composite or atomic) is to be decided by the restructuring project.

The planned architecture has two levels: a component can either be composite (large boxes in Figure 2), and contain other components, or atomic (smaller boxes), and be associated to current code. A few architectural rules were defined to constraint relationships between components, such as a strict three layer architecture (+ the physical device layer) where any layer can only interact with the layers directly above or below it (therefore Business layer cannot interact with Interface layer).

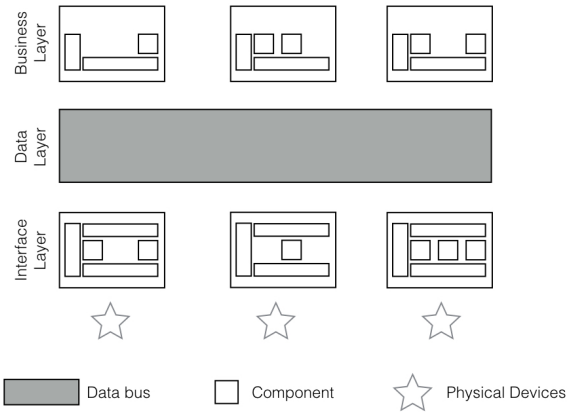


Fig. 2. Simplified representation of the target component-based architecture

### B. Existing Activities in Details

The restructuring relies on manual code analysis. A brief overview of the Ada language is given here to help the reader understand the activities taking place in this project.

The Ada entities of interest are: packages, subprograms and instructions. Packages contain entities that can be packages, subprograms or instructions. Subprograms contain only subprograms or instructions. Instructions are elementary entities (*e.g.* subprograms calls).

The software engineers study the source code to allocate software elements to one of the 16 pre-defined composite components. They do this by actually assigning software elements to potential atomic components within the composite components. Two tasks are therefore performed concurrently: assign software elements to the composite components and decomposing these into a number of atomic components. Software engineers try to allocate entire packages to a given atomic component. Yet studying it, they may consider that a part of it, maybe a subprogram, will need to be assigned to another atomic component (but always in the same composite component); they may sometimes even go down to assigning instructions (always subprogram invocations) to another atomic component.

From this informal process, we identified two interesting characteristic. First, they work at very different levels of abstraction, from coarse grained elements (packages) to very detailed ones (single instruction). To the best of our knowledge, we have not seen a proposition in the literature that allows to work with such a wide range of entities. Second, they adopted an iterative approach that tends to refine the results, working first on packages, then on subprograms and eventually on instructions.

*Allocating Packages:* Based on their understanding of the source code, the re-engineers select a set of packages that are likely to answer the responsibility of a component. The packages imported by each of these initial packages are normally allocated to the same component. Exceptions occur when an imported package has already been allocated to some other component. In that case, a deeper analysis of the package is performed, by looking at the comments in the source code, to resolve the conflict.

In case of doubt, another possibility is available: a library component gathering all the packages that could not be allocated to one single component.

*Allocating Subprograms and Subprograms Invocations:* Subprograms and subprograms invocations may be moved from one atomic component to another but only between sibling components (meaning they remain in their composite component). Re-engineers' analysis of the packages content relies on structural information such as dependencies between subprograms, and instructions in order to identify flows of execution. One must recall that we are dealing with a real time system which implies that functionalities or task within this system are first class citizens that receive a large part of the developers' attention. In comparison with more traditional information system, it seems that data received less attention during the system initial development. It is actually one of the goal of the system to correct this fact and identify high level, business, data structures that could possibly be reused in other related systems.

The analysis of the software engineers also relies on their knowledge of the system, their experience in the domain, the identification of features in the source code and the extraction of the source code associated to a feature. They are basically applying a concept slicing to determine the instructions that perform a specific feature and then allocate theses instructions to the component that should implement this feature. Respect of the mandatory / prohibited relationships between components are also used in the allocation of subprograms and subprograms invocations.

*Code Rewriting:* In accordance with the allocation, re-engineers rewrite the existing code to make the legacy software system work with its new architecture. Classic V-cycle is the basis of this part and it is done in parallel with the previous step.

At the moment, this project is ongoing, atomic components have been identified in all 16 composite components. Many packages and subprograms have been allocated to their composants, but all the work is not done yet and the final step of rewriting the code to actually build the new system did not start yet.

### C. Existing Approach Weaknesses

This informal process mostly relies on a comprehension of the legacy system by the re-engineers. A deep knowledge of each entity (package, subprogram or subprograms invocations) is required. Consequences are twofold.

First, the process is *fully manual* leading to time consumption and possible error creation. Due to the significant size of the legacy system, the full content of each package cannot be analysed in details by humans. Consequently, in the package allocating stage, the re-engineers use imported packages declarations to navigate dependencies from a given package. However, these declarations may be a super set of the actual dependencies. Unrelated packages are analysed together whereas they have no connected responsibilities and their analysis should be postponed.

Second, the process mostly *relies on the skills* of very few re-engineers who know the software enough. This deep knowledge eases their task, by enabling them to eventually skip the analysis of entities. However, their analysis strongly depends on the coding convention, making the process strongly dependent on the legacy software system. Consequently, the process is *ad hoc* and is difficult to reproduce on another legacy system or with another engineer.

## IV. A PROPOSED TOOL SUPPORTED PROCESS

To help overcome these weaknesses, we are working on formalizing a restructuring process and developing tools to automate it. Our solution is based over an existing industrial tool using the Moose technology [7]. Our hopes are: to validate the work already done and/or identify possible erroneous decisions; To speed up the remaining of the project; To prepare for envisioned future similar efforts in the company. While describing this process, we will highlight decision points ($DP$)

that we evaluated to understand which solution was the best and what consequences they could bring.

Our process consists in an iterative refinement of the allocation of software elements to components: In a first step, packages will be assigned to components, possibly resulting in conflicting assignments. We then go down to the level of subprograms to decompose the conflicting packages into coherent "sub-packages", and decompose composite components into atomic components. Finally we propose to go down to decomposing subprograms by allowing to assign some of their instructions to different atomic components. This last step assumes one would do subprogram extraction refactorings.

The process does no try to create components as often seen in past research, but relies on the knowledge of software engineers in their business domain to propose meaningful components and assign software elements to these components. This knowledge is reinforced by structural information (dependencies between elements) and the identification of special elements.

At the start of each iteration, a model of the system (or of a part of the system) is built with the software elements to be allocated (called *architectural quarks* following the convention in [5], *i.e.* the smallest significant elements) and their relationships (or dependencies). Each iteration then consists in asking the software engineers to identify *core quarks* and assign them to components. From these core quarks, we navigate the dependencies between all quarks to assign them to the components. The main decision points ($DP$) in evaluating our process and its automation will therefore be, at each iteration (or level): (i) how to choose the right core quarks; (ii) what dependencies to follow (or how to follow the dependencies as will be explained further down); and (iii) how to resolve conflicting assignments. Note that it is not mandatory to solve every conflict during a given step. Thus, at the end of an iteration, some conflicts might still exist.

Finally, we must note that our process does not yet address the code rewriting step.

### A. First iteration: Package allocation

This first iteration aims to allocate packages into composite components. At the architectural level of this iteration, the *quarks* are Ada packages. The edges of the dependency graph are the dependencies between packages as when one package imports another one.

The software engineers helped us to identify simple rules that allowed to identify the core packages of each composite components ($DP_c^1$ [*]). There can be as little as one core per component. It was easy for the software engineers to select them. Actually, the choice of the composite components of the wished architecture already implicitly relied on the identification of important part of the system that were easy to map to some packages. For the system under study, it was possible to define rules, based on naming convention and the presence of some specific software elements (an Ada procedure named `run`), allowing to automatically identify core packages for each high level component.

We experimented two alternative solutions for $DP_c^1$: the packages indicated by the software engineers (see above) and random choice of packages. Obviously, the second one will give poorer results, but it may be a specificity of this project that high level composite components and core packages seemed very clear to the software engineers.

The dependency graph is navigated from each core, following its dependencies recursively (see Figure 3). Each reached node is allocated to the same component as the core. Obviously, the same node may be reached from different cores because several packages depend on it (*e.g.* the triangle and diamond elements on the right of the figure) resulting in *conflicting* quarks (multi-core conflict). It may also occur that some quarks are never reached from the cores resulting in another type of *conflict* (isolation conflict as for the dotted elements on the left of the figure).

We experimented with three solutions for Decision Point $DP_d^1$ [†]: First the dependency relationships can be Ada *declared imports* between packages (Ada instruction `with`). Second they can be what we call *referenced imports* which are the declared imports that are actually necessary in the code to make it compile (as in many languages, the compiler does not check that a declared import is actually necessary). This is one point where an automated tool has an advantage over manual analysis in identifying the imports that are really required. The referenced imports are guaranteed to be a subset of the declared imports. Third, dependencies between packages can be the same referenced imports, but this time followed upward and downward, that is to say we look on what packages the core quarks depend (as for the two other solutions), but we also include in the component, the packages that depend on the core quarks.

Re-engineers may solve conflicts manually according to their knowledge of the application. They are helped by information about the packages *e.g.* their names, their comments or all the dependent packages in the dependency graph. In this iteration, conflicts resolution is not required since the iteration is used to set out the architecture migration process.

We experimented with two solutions for Decision Point $DP_r^1$ [‡]: first and as currently done by the software engineers, the "resolution" of multi-core conflicting packages may be to assign them to a special component called the Library. The rational is that packages that are required by more than one component need to be accessible from everywhere. In this first solution, we propose to resolve isolation conflict by following the dependencies *from* the isolated packages towards the core packages (note that the isolated packages are those that could not be reached from any core towards them). This part is similar to the third choice for $DP_d^1$.

A second option that we evaluated for $DP_r^1$ was to leave the components in conflict and evaluate them as if they were

[*] $DP_c^1$ stands for Decision Point for Core quarks at iteration 1.

[†] $DP_d^1$ stands for Decision Point for Dependency relationship at iteration 1.

[‡] $DP_r^1$ stands for Decision Point for Resolving conflicts at iteration 1.

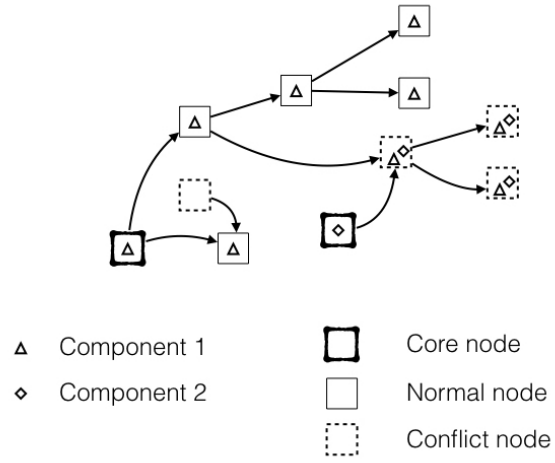


Fig. 3. Example of navigation and conflicts

actually member of both components (multi-core conflict) or of no component at all (isolation conflict).

### B. Second Iteration: Subprograms allocation

This second iteration aims to refine the allocations done in the first iteration by defining atomic components in the composite components and allocating subprograms to these atomic components. The dependency graph is build from subprograms (the quarks) and the dependencies between them are invocations. Again the iteration starts by choosing core quarks and allocating them to components following transitively the dependencies.

Allocation is always considered within a composite component, which means that subprograms in a package will always remain within the composite component of their package, but two subprograms of one package may be allocated to two different atomic components.

We experimented with only one composite component in this iteration.

The first option for $DP_c^2$ (choosing of core quarks for iteration #2) is again to follow the indications of the software engineers. (They were less clear cut this time). Several atomic components of the composite component considered for this evaluation were named by the software engineers from one package of the same composite component. All the subprograms of such packages were an obvious choice as core quarks for their respective atomic components. In other cases, we tried to rely on subprogram naming convention to choose core quarks for other atomic components.

For Decision Point $DP_d^2$, we relied on invocation between subprograms as stated above. We considered three options (not all mutually exclusive): only using downward invocation from the cores to other subprograms; using both upward and downward invocations; and/or considering what variables the subprograms use. The last option is based on the hypothesis that some variables are "indicators" of atomic components. By identifying these variables, we want to assign subprograms that use them to the components,

This iteration will also result in conflict that may be resolved by the software engineers or left for the next iteration.

### C. Third Iteration: subprograms invocations allocating

This iteration aims to resolve some of the conflicts left in the previous iteration by decomposing the subprograms and assign individual instructions (extracted in new subprograms) to atomic components. As for the manual process, the instructions considered here are invocation (subprogram calls) instructions.

We identified some conflicting subprograms that act as "dispatcher" between several atomic component: The flow of execution reaches them and they channel it toward the appropriate atomic component after a series of tests. Such dispatcher subprograms are often in conflict in that they appear to be assignable to several atomic components, usually all the components to which they may dispatch the flow of control. One idea is to try to decompose these dispatchers so that each flow of execution is concentrated in its atomic component and the dispatching done earlier in the flow of execution.

We neither had the time nor enough data yet to experiment solutions for this step.

## V. Experiment description

We presented a process to help software engineers restructure a software system. At each iteration of the process, we identified decision points and proposed solutions. We compare these solutions to an oracle corresponding to the manual work of the software engineers after possible modifications.

We first present the oracle before discussing in detail each proposed solution and see their respective efficiency and what impact they have on the result.

### A. Oracle

The manual work described in section III-B is considered the oracle. The target architecture is composed of 16 composite components. One of them is the *Library* component and is used to gather all the packages that are needed (*i.e.* imported by a package) by several composite components. The engineers allocated $1\,537$ packages on to the 16 composite components. $408$ packages, out of the $1\,537$ packages of the application, were allocated to the Library. $593$ packages have been categorised as messages and are treated by the engineers during the second iteration. This first step took about two weeks long to four engineers to complete. This first assignment of packages into composite components is the base for our oracle (evaluation of our first step solutions).

At the time of this writing, the engineers finished defining all the atomic components belonging to the composite components. However, few of these atomic components have been populated by software elements (packages, subroutines, instructions). The engineers focused on two of the 16 composite components, and their atomic components are considered

complete. These two composite components and their atomic components are used to evaluate the results of our second step solutions. This part of the oracle took some time to establish as it requires a more in depth study of the packages and their subprograms. Each of these two composite components took one person/month to complete. The software engineers established a first classification that was too low level (not sufficiently abstract). After reworking they merged some of the atomic components originally identified together.

Unfortunately, we could not establish a meaningful oracle for the evaluation of the third step solutions.

### B. Evaluation of solutions

Evaluation of all proposed solutions will be computed against the oracle. We use Precision and Recall metrics, also summarised in the F-Score metric. For any possible allocation of quarks in components:

- Precision = % of quarks allocated to a component in the evaluated solution that are in the same component in the oracle.
- Recall = % of quarks allocated to a component in the oracle that are in the same component in the evaluated solution.
- F-Score = 2*(precision*recall)/(precision + recall)

Typically, one may achieve better precision (all allocated quarks are in their right component) by lowering the recall (very few quarks are actually allocated to any component). The opposite is also true. For this reason, the F-Score metrics offers a balanced value of both metrics together.

The result we give are the arithmetic mean of the metrics values over all components.

For reference, we give the results of the first allocation of software elements into atomic components made by the engineers (Table II). They later improved this work, but considering their initial result can give some ground of comparison to rank the quality of the automated solutions. These results may be compared to what we did on the second iteration of our process (see Section V-D. The results are not perfect and show how difficult it can be for human experts to allocate software elements into components.

TABLE II
EVALUATION OF THE RESULTS OF THE FIRST VERSION OF EXPERTS ALLOCATION

| Precision | Recall | F-Score |
|---|---|---|
| 65% | 83% | 66% |

### C. Proposed Solutions For Iteration 1

$DP_c^1$: Software engineers identified 64 packages as the core quarks for the 16 composite components. As mentioned in Section V-C, we could easily define rules to automatically identify the core packages based on naming convention of the packages and the presence of some specific software elements (an Ada procedure named `run`). This is due to the fact that the composite components match high level features of the system which themselves each correspond to some easily recognizable package.

$DP_d^1$: We experimented three solutions:

- *Declared imports* between packages (Ada instruction `with`), going from core packages to the packages they import (and transitively);
- *Referenced imports* which are the declared imports backed up by some actual use of the package content, going from core packages to the packages they import (and transitively);
- *Up/Down imports* which are the referenced imports going from core packages to the packages they import (and transitively) and also from core packages to the packages that import them (and transitively). Both paths (up and down) are followed completely from cores without going back. In other terms directions are not mixed.

$DP_r^1$: We experimented two solutions:

- *With library* where packages in conflict are allocated to a special component called the library (precision and recall are computed for this special component as well as the other);
- *No library* where packages are allowed to remain in conflict and are considered allocated to several composite components and the library components is empty which already means precision and recall will be lower.

TABLE III
DECISION POINTS FOR ITERATION 1

| | Precision | Recall | F-Score |
|---|---|---|---|
| $DP_d^1$ Declared imports<br>$DP_r^1$ With library | 68% | 93% | 76% |
| $DP_d^1$ Referenced imports<br>$DP_r^1$ With library | 94% | 89% | 90% |
| $DP_d^1$ Up/Down imports<br>$DP_r^1$ With library | 25% | 3% | 6% |
| $DP_d^1$ Declared imports<br>$DP_r^1$ No library | 6% | 96% | 9% |
| $DP_d^1$ Referenced imports<br>$DP_r^1$ No library | 9% | 93% | 12% |
| $DP_d^1$ Up/Down imports<br>$DP_r^1$ No library | 1% | 93% | 11% |

The first conclusion is that precision is much better when we resolve conflict by assigning packages to the library component. This was expected as this library contains 1 001 packages in the oracle that will all hurt the precision metrics if they were not allocated to their library component. Recall is better if we do not assign conflicting packages to the library because it gives a chance to some packages that should not be in the library to be actually allocated to their component. Overall, the loss in precision largely outweighs the gain in recall as shown by the F-Score.

We also see that referenced imports are better than declared imports in terms of precision. The explanation must be that some packages are indeed imported where they are not needed

and this introduces noise. As expected this improve in precision is made at the expense of the recall but the F-Score still is better for referenced imports. This, therefore, seems to be the solution of choice.

Finally, considering referenced imports both downward and upward from the core packages significantly lowers the results. The explanation is that much more packages can be reached from any core with this solution. Some of them may be correct hits, but many more should not have been reached thus hurting precision. For line 3, recall also diminishes drastically, because with more packages reached from any core, many end up in conflict and are therefore allocated to the library component instead of their right component.

Because of its size (two third of all the packages of the application), including the library in the comparison could impact the precision, recall and F-Score. We decided then to make two comparisons for our chosen solution (line 2 of table VI): a comparison that includes the packages of the library; a comparison that does not include the packages of the library. Table IV shows that precision ad F-Score are increased without taking the library into account in the comparison while recall is stable (only a decreasing of on but it is mostly due to the fact that metrics are rounded).

TABLE IV
CHOSEN SOLUTION INCLUDING AND NOT INCLUDING LIBRARY COMPONENT IN COMPARISON

| | | Precision | Recall | F-Score |
|---|---|---|---|---|
| $DP_d^1$ | Referenced imports | 94% | 89% | 90% |
| $DP_r^1$ | With library | | | |
| | Library in Comparison | | | |
| $DP_d^1$ | Referenced imports | 98% | 88% | 93% |
| $DP_r^1$ | With library | | | |
| | No Library in Comparison | | | |

Table V gathers precision, recall and F-Score for each composite component of the application (excluding the Library) in case of a comparison like in line 2 of table IV. Name of the composite components are changed to respect the company rules on confidentiality.

Precision is stable for each composite component, except for the composite F. We took a deeper look at this composite and it appears that packages are coming from the others composite. However no general rule can be found to handle this effect. Although recall fluctuates between 70% and 100%, leading the F-Score to also fluctuates, these two metrics stay above 70%

### D. Proposed Solutions For Iteration 2

In this Iteration, quarks are subprograms that are allocated to atomic components. One decision in this iteration is that we would not go back on the previous results and leave subprograms in the composite component where their parent packages are allocated. But two subprograms in the same package may be allocated to two different atomic components (within the same composite component). For the evaluation

TABLE V
PRECISION, RECALL AND F-SCORE FOR EACH COMPOSITE COMPONENT IN ITERATION 1 WITH OUR CHOSEN SOLUTION

| | Precision | Recall | F-Score |
|---|---|---|---|
| A | 94% | 82% | 88% |
| B | 100% | 85% | 92% |
| C | 100% | 100% | 100% |
| D | 92% | 95% | 93% |
| E | 94% | 70% | 76% |
| F | 84% | 99% | 99% |
| G | 100% | 75% | 86% |
| H | 100% | 89% | 94% |
| I | 100% | 87% | 93% |
| J | 98% | 98% | 98% |
| K | 97% | 75% | 85% |
| L | 100% | 90% | 95% |
| M | 100% | 86% | 92% |
| N | 100% | 100% | 100% |
| O | 100% | 96% | 98% |

of our solution, we only consider two specific composite components that were fully treated by the software engineers.

$DP_c^2$: Based on some indications by the software engineers, rules were defined using a naming convention on the subprograms to identify core subprograms of some atomic components (the name of the atomic components can be found in the name of some subprograms). In other cases, one package was identified as central to an atomic component and all its subroutines were consequently chosen as core quarks for this component. This work was performed without the direct help or monitoring of the software engineers and therefore, we cannot ensure that our chosen cores are the best we could have defined for each atomic component. The rational was that we wanted to see how well we could do with a more limited understanding of the system. We did had to correct some of our rules after a first try when we obtained results that were not those expected. We believe, this kind of iterative refinement of the core quarks selection is normal and reflects actual working conditions when the software engineers would not have a perfect understanding of the existing system.

$DP_d^{12}$: We experimented two solutions:

- *Downward invocations* between subprograms, going from core subprograms to the subprograms they call (and transitively);
- *Up/Down invocations* going from core subprograms to the subprograms they call (and transitively) and also from core subprograms to the subprograms that call them (and transitively).

$DP_r^2$: We experimented three solutions:

- *No action* where subprograms are left in conflict. Contrarily to the previous iteration, in this case we consider that subprograms in conflict are not allocated to any atomic component. This is somehow similar to saying they are allocated to a kind of "library atomic component"

although such atomic component does not really exist. The subprograms in conflicts are left to be disambiguated in the next iteration of the process;

- *Variable uses (post)* where some variables were identified that are only used within an atomic component (we say they are *endemic* to this component). Using these variables, we try to resolve subprograms in conflict by looking if they refer to any of these variables and if so, allocating them to the atomic component related to this variable.
- *Variable uses (during)* where we worked again with variables endemic to an atomic component. But this time, we try to resolve conflicts immediately as they arise during propagation from the cores. If we could resolve the conflict, this allowed us to propagate one step further the allocation to a component. In this setting, the set of endemic variables is recomputed at each step of the propagation and gradually shrink, but it is larger at the beginning, allowing to allocate more subprograms to atomic components.

TABLE VI
DECISION POINTS FOR ITERATION 2

| | Precision | Recall | F-Score |
|---|---|---|---|
| $DP_d^2$ Downward invocations<br>$DP_r^2$ No action | 57% | 51% | 51% |
| $DP_d^2$ Downward invocations<br>$DP_r^2$ Variables Uses (post) | 60% | 49% | 51% |
| $DP_d^2$ Downward invocations<br>$DP_r^2$ Variables Uses (during) | 74% | 48% | 54% |
| $DP_d^2$ Up/Down invocations<br>$DP_r^2$ No action | 37% | 65% | 45% |
| $DP_d^2$ Up/Down invocations<br>$DP_r^2$ Variables Uses (post) | 53% | 58% | 52% |
| $DP_d^2$ Up/Down invocations<br>$DP_r^2$ Variables Uses (during) | 53% | 62% | 57% |

The first line of result is not very good due to a large number of subprograms in conflict.

Result of the second line are a slightly better, showing that resolving conflicts with endemic variables can improve precision. Unfortunately, there are actually few such variables.

The third line brings much better precision with very little impact on recall. This is due to the fact that we have a larger set of endemic variables at the beginning, when only few subprograms are allocated to an atomic component. This suggest that we might want to more permissive when computing endemic variables.

The fourth line is interesting because it brings an increase in recall when compared to the first one. This was expected as we gather more subprograms, taking the ones called by the core quarks and the ones that call the core quarks. Precision is low because, as for iteration 1 (Section V-C), a lot more nodes are reached from any core, including some that should not be.

When introducing conflict resolution with the help of endemic variables, we observe the same phenomenon as for line 2: higher precision, lower recall.

Finally, the last experiment raises agin the recall, while not affecting precision. tis gives the best F-Score over all six experiments.

Results for this iteration are encouraging, but still not satisfying. They are definitely worse than the first try of the experts. However, they were obtain which much less efforts (automated solutions) and by people with a lesser understanding of the system (we are not experts).

We are still working on this iteration to try to get more satisfying results. One direction of possible improvement would be to continuue working on the variables, either by loosening the constraint for endemic variables, or finding other interesting data. We noticed for example some "dispatching" subprograms (see also Section IV-C) that allow to distribute the flow of execution over several atomic components. They do this by looking at the value of specific variables. Identifying these variables and the values of interest could help us improve our results, somehow as endemic variables already helped us.

## VI. CONCLUSION AND FURTHER WORK

A large company called us to follow and help in its legacy software migration project. The legacy software had modularisation problems that led the company to decide for a restructuring of the system architecture. The project started, before we stepped in, by the definition of a wished, target architecture at a rather high abstraction level. Then, the engineers and architects performed an informal software architecture migration process from which we pointed out weaknesses that could prevent its reproduction in another migration.

We decided to propose a tool-supported process that remedies these weaknesses. We based the definition of our process on the informal one that we witnessed in the project. Our tool supported solution is an iterative process that combines dependency graph extraction, data uses analysis, and domain knowledge of the re-engineers to select core elements that serve as seed for components. While other research only work with a fixed kind of software elements (typically classes or procedures), our process works with different kinds (packages and subprograms for now).

We identified several points of decision in our process and proposed possible choices at each point. These choices were then evaluated to see which one made more sense.

The next steps of our work will be to improve our results in the second iteration and/or concentrate on the third step that is intended to resolve the conflicts left in the second one. Our main hopes for now will be to analyze data usage in the subprograms to either allocate them to their correct component or understand how to split them into different subprograms each going more clearly into one atomic component.

## ACKNOWLEDGEMENT

We wish to express all our gratitude to people at Thales for helping us understanding their system and evaluating our results.

## REFERENCES


[1] R. Kazman, S. Woods, and S. Carriére, "Requirements for integrating software architecture and reengineering models: Corum ii," in *Proceedings of WCRE '98*. IEEE Computer Society, 1998, pp. 154–163, iSBN: 0-8186-89-67-6.

[2] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. d. P. Leite, "Reverse engineering goal models from legacy code," in *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*. IEEE, 2005, pp. 363–372.

[3] R. Khadka, A. Saeidi, S. Jansen, and J. Hage, "A structured legacy to soa migration process and its evaluation in practice," in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2013 IEEE 7th International Symposium on the*. IEEE, 2013, pp. 2–11.

[4] S. Allier, H. A. Sahraoui, S. Sadou, and S. Vaucher, "Restructuring object-oriented-applications into component-oriented applications by using consistency with execution traces," in *Component-Based Software Engineering*. Springer, 2010, pp. 216–231.

[5] R. Koschke, "Atomic architectural component recovery for program understanding and evolution," Ph.D. dissertation, Universität Stuttgart, 2000. [Online]. Available: http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/papers/koschke.thesis.2000.html

[6] C. Gerardo and D. P. Massimiliano, "New frontiers of reverse engineering," in *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 326–341.

[7] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," in *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, ser. CoSET '00, Jun. 2000. [Online]. Available: http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf

[8] O. Barais, A. F. Le Meur, L. Duchien, and J. Lawall, "Software architecture evolution," in *Software Evolution*. Springer, 2008, pp. 233–262.

[9] R. Koschke, "An incremental semi-automatic method for component recovery," in *Working Conference on Reverse Engineering*, 1999, pp. 256–. [Online]. Available: http://citeseer.nj.nec.com/koschke99incremental.html

[10] S. J. Carrière, S. Woods, and R. Kazman, "Software architectural transformation," in *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*. IEEE, 1999, pp. 13–23.

[11] A. A. Almonaies, J. R. Cordy, and T. R. Dean, "Legacy system evolution towards service-oriented architecture," in *International Workshop on SOA Migration and Evolution*, 2010, pp. 53–62.

[12] R. Khadka, G. Reijnders, A. Saeidi, S. Jansen, and J. Hage, "A method engineering based legacy to soa migration method," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 163–172.

[13] R. K. A. Idu, J. Hage, and S. Jansen, "Legacy to soa evolution: A systematic literature review," *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments: Challenges in Service Oriented Architecture and Cloud Computing Environments*, p. 40, 2012.

[14] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *Proceedings of the 13th International Conference on Software Engineering*, May 1991, pp. 83–92.

[15] A. Seriai, S. Sadou, and H. A. Sahraoui, "Enactment of components extracted from an object-oriented application," in *Software Architecture*. Springer, 2014, pp. 234–249.

[16] A. L. Gear, "Component reconn-exion," 2006.

[17] Z. Wang, X. Xu, and D. Zhan, "A survey of business component identification methods and related techniques," *International Journal of Information Technology*, vol. 2, no. 4, pp. 229–238, 2005.

[18] P. Johnson, "Mining legacy systems for business components: An architecture for an integrated toolkit," in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*. IEEE, 2002, pp. 563–571.