Equipping the Reflexion Method with Automated Clustering

Andreas Christl
University of Stuttgart, Germany
christas@droste.informatik.uni-stuttgart.de

Rainer Koschke
University of Bremen, Germany
koschke@informatik.uni-bremen.de

Margaret-Anne Storey
University of Victoria, Canada
mstorey@csr.uvic.ca

Abstract

A significant aspect in applying the Reflexion Method is the mapping of components found in the source code onto the conceptual components defined in the hypothesized architecture. To date, this mapping is established manually, which requires a lot of work for large software systems. In this paper, we present a new approach, in which clustering techniques are applied to support the user in the mapping activity. The result is a semi-automated mapping technique that accommodates the automatic clustering of the source model with the user's hypothesized knowledge about the system's architecture.

This paper describes also a case study in which our semi-automated mapping technique has been applied successfully to extend a partial map of a real-world software application.

1   Introduction

Software architecture is described by many views. The most popular view addressed in research is the module view [15]. The module view describes the modules of a system, their layering and composition into subsystems, and the provided and required interfaces of these elements. The module view is required for many purposes such as allocating working packages to teams, global change impact analysis, evaluating the maintainability of the system, and more.

Far too often, the module view that was initially designed does not reflect the real implementation due to changes made in the source without updating the documented module view. Murphy and colleagues [28] have developed the reflexion model technique to reconstruct the mapping from the specified or hypothe-sized to the concrete module view. The basic idea of the reflexion model is to create a hypothesized view from existing documentation or interviews with architects. Then source entities are extracted from a system (global variables, routines, types, classes, interfaces, packages, files, subdirectories, etc.) along with their respective dependencies forming the concrete module view. These elements are then mapped to the hypothesized view. A tool then computes resemblances and differences between the two views. Iteratively, the hypothesized and concrete views and/or the mapping are refined based on the findings.

The technique was successfully used in several case studies. The most interesting case study—reported by Murphy and Notkin [27]—is the analysis of Microsoft Excel, which consists of about 1.2 MLOC of C code. Later, we extended the original reflexion model so that hypothesized modules can be hierarchical and applied it to two different compilers [18].

The most challenging part of the reflexion method is to determine the mapping of concrete source entities onto the hypothesized entities of the hypothesized model. The original reflexion method does not provide any support for that – although sometimes naming conventions may be leveraged. Yet, they often do not exist or are used inconsistently.

The key point of the reflexion method is to start with an initial hypothesis on the expected module view and then to validate the hypothesis against the implementation. Inversely, software clustering techniques group source entities together – typically based on some notion of coupling and cohesion – to form hypothesized entities. The advantage of clustering techniques is that they can be completely automated. Yet, these techniques are not directed towards the expectations of the analyst and often fail to find the components a human would find [17].

Contributions. This paper combines the reflexion method with automatic clustering techniques. The automatic clustering techniques are used to create additional candidate mappings based on a partial mapping. For this to work, the clustering techniques are adjusted so that they consider an existing mapping and the targeted hypothesized model. A case study explores the influence of degree of completeness of the existing mapping to make reliable suggestions.

2 Related Research

This section describes related research. We start with a detailed description of the reflexion technique and introduce concepts later used in the description of our extension. Then we summarize research in the wider area of software clustering.

2.1 Reflexion Method

The reflexion method is based on three things. The **hypothesized view (HV)** models the user's viewpoint on the system design. It consists of hypothesized entities and their expected dependencies. Hypothesized entities represent "coherent conceptual modules" [18], and thus provide the entire or partial system decomposition that the user expects to exist.

In this paper, the hypothesized view HV is modelled as a directed graph containing the hypothesized entities N_H connected by hypothesized dependencies E_H.

$HV = (N_H, E_H)$

$N_H = \{h_1, h_2, ... h_n\}, n \in \mathbb{N}$ set of hypothesized entities

$E_H \subseteq (N_H \times N_H)$ set of hypothesized dependencies

The **concrete view (CV)** models the software system from the viewpoint of its implementation. The concrete view consists of the concrete entities and their dependencies found in the code. Analogously to the hypothesized view, the concrete view is modelled as a directed graph containing the concrete entities N_C connected by concrete relationships E_C.

$CV = (N_C, E_C)$

$N_C = \{c_1, c_2, ... c_n\}, n \in \mathbb{N}$ set of concrete entities

$E_C \subseteq (N_C \times N_C)$ set of concrete dependencies

The hierarchical nature of concrete entities is modelled using the function $partof$, which maps concrete entities to their containing concrete entities, for instance, methods onto classes or classes onto packages:

$$partof : \quad N_C \longrightarrow N_C$$

Finally, the (potentially partial) **mapping** $maps\text{-}to\colon N_C \rightarrow N_H$ connects the two views CV and HV. A concrete entity may either be mapped directly by the user (denoted by $dmaps\text{-}to$ or be mapped indirectly by way of its closest containing component that is directly mapped:

$maps\text{-}to(c) =$
$$\begin{cases} dmaps\text{-}to(c) & : & c \in \mathtt{dom}\ dmaps\text{-}to \\ maps\text{-}to(c') & : & c \notin \mathtt{dom}\ dmaps\text{-}to \wedge part\text{-}of(c) = c' \\ undefined & : & \text{otherwise} \end{cases}$$

The dependencies in both CV and HV may be typed: function $type$ assigns a type to each dependency edge both in CV and HV. One edge type may be a subtype of another edge type, forming an is-a type hierarchy.

The result of the reflexion analysis is a **reflexion model (RM)**, summarizing the differences and the agreements of hypothesized and concrete view on the basis of the mapping. The result is a multigraph $\mathfrak{R} = (N_C, E_{\mathfrak{R}})$ where the edge types in $E_{\mathfrak{R}}$ are as follows (let $h_i, h_j \in N_C$):

Two hypothesized entities are connected by a **convergence** edge, if an expected relationship defined in the hypothesized view could be found between concrete entities mapped to them.

$$convergence(h_i, h_j) \in E_{\mathfrak{R}} :\Leftrightarrow$$
$$\exists [c_k, c_r] \in E_C, [h_i, h_j] \in E_H : \text{maps-to}(c_k) = h_i \wedge$$
$$\text{maps-to}(c_r) = h_j \ \wedge \ type([c_k, c_r]) \text{ is-a } type([h_i, h_j])$$

Two hypothesized entities are connected by an **absence** edge, if an expected relationship could not be found in the concrete view.

$$absence(h_i, h_j) \in E_{\mathfrak{R}} :\Leftrightarrow$$
$$\nexists [c_k, c_r] \in E_C\ \exists [h_i, h_j] \in E_H : \text{maps-to}(c_k) = h_i \wedge$$
$$\text{maps-to}(c_r) = h_j \ \wedge \ type([c_k, c_r]) \text{ is-a } type([h_i, h_j])$$

Two hypothesized entities are connected by a **divergence** edge, if a dependency was found between their mapped concrete entity that was not expected by the user.

$$divergence(h_i, h_j) \in E_{\mathfrak{R}} :\Leftrightarrow$$
$$\exists [c_k, c_r] \in E_C\ \nexists [h_i, h_j] \in E_H : \text{maps-to}(c_k) = h_i \wedge$$
$$\text{maps-to}(c_r) = h_j \ \wedge \ type([c_k, c_r]) \text{ is-a } type([h_i, h_j])$$

2.2 Clustering Techniques

Clustering techniques aim at deriving the structural decomposition of a system into modules. Logically related declarations (global variables, routines, data types, and classes) or whole packages are recovered and described by the part-of relationship. The results are flat or hierarchical modules. Flat modules are, for instance, abstract data types or objects [7, 6, 8, 16, 20, 21, 29, 30, 32, 40, 42, 45]. Hierarchical modules are often referred to as subsystems [10, 13, 35, 23].

Typically, static dependencies such as calls or accesses to variables or fields are leveraged for the grouping. Sometimes, similarities between identifiers in the presence of naming conventions are considered [3, 14] or an explicit modeling of the application domain is taken into account [12]. Only one approach (to our best knowledge) uses dynamic information gathered by way of use cases [5].

Several types of techniques are used to determine modules. Software clustering – as one of the most popular ones – is based on known clustering techniques that have been developed and used in other domains such as biology [24, 1, 17, 35, 14]. Generally, a hierarchical agglomerative approach is used. The approaches differ largely in their definition of similarity to compare two individual entities and whole groups to be grouped.

Other techniques view module recovery as a partitioning problem that aims at minimizing coupling and maximizing cohesion between modules. To find a good partition (finding an optimal one is NP hard) genetic algorithms [23] and other more traditional search techniques, such as hill climbing, are used [22, 23].

Methods from graph theory are also used, for instance, cycle detection to find strongly connected entities or dominance analysis to find entities local to other entities [10, 13, 26], or graph pattern matching [34].

Other techniques are based on formal concept analysis that allows one to analyze binary relations [5, 41, 19, 37, 38, 32]. The resulting concept lattice describes a hierarchy of concepts, which can then be used to identify modules. The approaches here differ in the binary relation and how the concept lattice is used. Data mining [11, 33] and spectral analysis [36] is used as well.

Often, these completely automatic techniques are integrated into an interactive process [26, 17] because the factors for a sensible grouping are not exactly defined (at least not in an operational manner). For interactive methods incremental techniques play an important role, in which not yet clustered entities are assigned to existing groups [39, 17].

3 Integration of RM and Clustering Techniques

3.1 Manual Mapping

It is helpful to view the Software Reflexion Model technique from a clustering perspective, in order to identify the potential of cluster analysis to support the mapping activity. From our perspective, the Software Reflexion Model technique can be described as an *incremental*, *human-controlled*, *non-hierarchical* clustering technique, in which concrete entities that are mapped to the same hypothesized entity form a cluster. In other words, each hypothesized entity in the hypothesized view represents one cluster and the function *maps-to* associates concrete entities with these clusters.

The goal of the mapping activity is to assign concrete entities to their *correct* hypothesized entity. The *correct* hypothesized entity is the hypothesized entity to which a concrete entity belongs when considering the existing architecture of the system under investigation. The Software Reflexion Model technique is an *incremental* clustering technique because concrete entities are mapped to hypothesized entities in multiple iterations (described by Murphy *et al.* [28]). The term *human-controlled* signifies that cluster decisions are exclusively made by the user.

3.2 Semi-Automated Mapping

Most incremental clustering techniques cluster all free entities in one iteration [39, 25, 17]. If directly applied to automated mapping, these techniques would yield a complete map in one step, which the user could either accept, reject, or validate manually. This behavior would therefore interfere with the human control and the iterative nature of the Software Reflexion Technique. In order to preserve these attributes, semi-automated mapping should not replace the manual mapping process but should assist and support the user to achieve a correct map faster. The following ideas allow a smoother integration:

1. The cluster analysis should identify concrete entities for which a mapping decision is "easy enough" in order to be made automatically. Only these concrete entities are mapped automatically by the integrated clustering algorithm.

2. For those concrete entities for which an automatic mapping decision is not possible, the cluster analysis should support the user in the manual mapping decision by detecting hypothesized entities that are likely to be the correct hypothesized entity.

Furthermore, the introduction of semi-automated mapping must not jeopardize the main success factors of the Software Reflexion Model technique namely its lightweight nature, scalability, and approximity [28].

3.3 Clustering Algorithm

To address these requirements, we have developed an *incremental, supportive, partial* clustering algorithm for semi-automated mapping that incorporates assets of existing clustering techniques. In the first phase of our algorithm, unmapped concrete entities are filtered if a meaningful mapping decision is not possible, since they are not connected or only slightly connected with mapped concrete entities. Next, an attraction matrix is calculated containing attraction values for each pair of unmapped concrete and hypothesized entities. The attraction values are based on source dependencies between the unmapped concrete entities and already mapped concrete entities and represent the likeliness that a hypothesized entity is the matching partner for a concrete entity. During our research, we developed two attraction functions for the calculation of the attraction matrix. Both attraction functions are derived from existing clustering approaches. Based on the attraction matrix, potentially matching hypothesized entities are then detected for each concrete entity. The actual clustering (i.e., automated mapping) takes place in the last phase of the algorithm. All concrete entities are automatically mapped for which only a single candidate could be detected. All others are presented to the user ranked by their attraction. The user then decides.

3.3.1 Partial Clustering

As discussed earlier, the attraction values are based on source dependencies between the unmapped concrete entities and already mapped concrete entities. Source relationships between two unmapped concrete entities have no effect on the attraction values. This approach, however, is problematic if unmapped concrete entities share many source relationships with unmapped concrete entities and only a few with mapped ones. In this case, chances are high that the calculated attraction values are not representative for the concrete entity and the correct hypothesized entity might not have a significant attraction.

In order to prevent poor cluster decisions due to a high degree of ignored source dependencies, a filter function δ is applied on the set of unmapped concrete entities (to simplify the description, let $\{c_i, c_j\}$ denote

the undirected edges $[c_i, c_j]$ or $[c_j, c_i]$) .

$$\delta(S \subset N_C) = \{c_i \in S \mid$$
$$\frac{|\{\{c_j, c_i\} \in E_C \mid c_j \in dom(\text{maps-to}) \ \wedge \ c_j \neq c_i\}|}{|\{\{c_j, c_i\} \in E_C \mid c_j \neq c_i\}|}$$
$$\geq \omega\}$$

The filter function δ filters concrete entities whose ratio of mapped neighbors does not allow a meaningful attraction calculation. Hence, concrete entities that have a high degree of source dependencies to unmapped concrete entities are not considered for the automated mapping. The threshold that defines whether a concrete entity is filtered and whether a concrete entity is considered for the automated mapping is specified by the parameter $\omega \in]0, 1]$. An ω value close to 1 will pass through only those free concrete entities that are mostly connected with mapped concrete entities. Hence, the cluster decision for those entities is based on the majority of their dependencies and is not likely to be strongly affected by ignored source dependencies. On the other hand, an ω value close to 0 is likely to result in more concrete entities to be considered for automated mapping. The accuracy of the attraction calculation might be misled due to a high number of ignored source dependencies of the entity with unmapped entities. For the presented case study, an ω value of 0.5 was applied that we determined experimentally [9].

3.3.2 Attraction Calculation

The core of a clustering algorithm is a similarity function which determines the closeness of entities [43]. In the context of our semi-automated mapping, the term *similarity* is appropriate; however, it might be misleading, since concrete entities are assigned to hypothesized entities and not to similar concrete entities. To avoid confusion when we talk about our semi-automated mapping, we use the term *attraction* to define the "closeness" of concrete entities to hypothesized entities.

$$attraction : \quad N_C \times N_H \longrightarrow \mathbb{R}_0^+$$

We considered similarity functions from two existing clustering techniques and tailored them towards our semi-automated mapping. The attraction functions *CountAttract* and *MQAttract* follow different approaches to calculate the attraction between a hypothesized entity and a concrete entity. The *CountAttract* focusses on the minimization of coupling between hypothesized entities, while the *MQAttract* attraction function follows the approach of the Bunch clustering based on coupling and cohesion [25]. Both attrac-

tion functions *CountAttract* and *MQAttract* implement the generic function *attraction*. In the following case study, the attraction functions are compared in terms of their value for automated mapping.

Both attraction function base their calculations on the source dependencies between mapped and unmapped concrete entities. Similarly to other clustering techniques, both attraction functions apply a weighting function on source dependency types to reflect their semantic impact on the clustering [31, 25].

$$\lambda \ : \quad T \longrightarrow \mathbb{R}^+ \qquad T = \text{set of dependency types}$$

Generally, the occurring dependency types vary depending on the programming language of the investigated system and on the scope of the Software Reflexion Model. This flexibility of the Software Reflexion Model technique, however, makes it difficult to provide a fixed set of dependency weights. Moreover, different reengineering tasks might require different weights. More research is required to identify weight sets that are optimal for the different scopes of the technique.

CountAttract

The attraction function *CountAttract* is derived from Koschke's connection based approach [17, algorithm 8-10]. In his proposed clustering algorithm, a free entity is assigned to the cluster with the highest similarity value; in other words, the highest number of entities that are already part of the cluster and share a source dependency with the unmapped entity. Thus, each entity is assigned to the cluster so that the lowest number of inter-cluster dependencies is produced. Note that this approach does not lead to an absolute minimal number of edges between clusters after all entities have been clustered, and a different order in which the entities are clustered might result in different clusters. In Koschke's clustering technique, however, the similarity function does not facilitate simple source dependencies. Instead, two entities share a relationship if they are related to each other according to a certain clustering heuristic. The same principle can also be applied to simple source dependencies. In this case, two entities are connected if they share a source dependency from the concrete view. The Orphan Adoption algorithm proposed by Tzerpos *et al.* [39] implements a similar type of connection-based similarity function that works on basic source dependencies.

The *CountAttract* attraction function follows this coupling-focussed approach by Tzerpos and Koschke. The attraction of a hypothesized entity towards a concrete entity equals the weighted sum of all source dependencies between the concrete entity and all concrete

entities mapped to the hypothesized entity.

$$CountAttract(c_i, h_k) \ = \ \sum_{\forall \{c_i, c_j\} \in E_C \, : \, \text{maps-to}(c_j) = h_k} \lambda(type(\{c_i, c_j\}))$$

In addition to the information from the source code, we integrate more user information into the attraction calculation in order to improve the quality of our clustering results. In the hypothesized view, the user defines hypothesized dependencies that are allowed to exist in the system architecture. Hence we can distinguish two types of source relationships that cause coupling when a concrete entity is mapped. First, source relationships that cause allowed coupling and second, source relationships that result in unwanted or not anticipated coupling. It is intuitive that allowed source relationships should have less of a negative impact on the attraction than unanticipated source relationships. In order to integrate this user knowledge into the calculation, the *CountAttract* function is transformed and extended.

$CountAttract(c_i, h_k)$ counts the number of source dependencies between a free concrete entity c_i and concrete entities mapped to hypothesized entity h_k. In order to integrate hypothesized dependencies, the initial attraction function is transformed into $CountAttract'$.

$$CountAttract'(c_i, h_k) = overall(c_i) - toOthers(c_i, h_k)$$

$$overall(c_i) = \sum_{\forall \{c_i, c_j\} \in E_C \, : \, c_j \in dom(\text{maps-to})} \lambda(type(\{c_i, c_j\}))$$

$$toOthers(c_i, h_k) = \sum_{\substack{\forall \{c_i, c_j\} \in E_C \, : \\ c_j \in dom(\text{maps-to}) \, \wedge \, \text{maps-to}(c_j) \neq h_k}} \lambda(type(\{c_i, c_j\}))$$

After the transformation, the number of source dependencies from concrete entity c_i to concrete entities mapped to hypothesized entities h_k is counted indirectly. The number of source dependencies that c_i shares with concrete entities of all other hypothesized entities ($toOthers(c_i, h_k)$) is subtracted from the total amount of source dependencies of c_i with mapped concrete entities ($overall(c_i)$).

The transformation does not modify the result of the function but makes it easier to see that *CountAttract* favors the hypothesized entity with the largest number of connecting source dependencies. If the concrete

entity c_i is clustered to a hypothesized entity h_k, the source dependencies counted in $toOthers(c_i, h_k)$ result in coupling. The larger $toOthers(c_i, h_k)$ is, the lower the attraction to this hypothesized entity will be.

At this point, each counted source dependency has the same influence on the attraction, regardless of whether the resulting coupling is allowed or not. To take this conceptual information into account, the function $toOthers(c_i, h_k)$ is modified.

$$toOthers(c_i, h_k) =$$
$$\sum_{\substack{\forall \{c_i, c_j\} \in E_C : \\ c_j \in dom(\text{maps-to}) \,\wedge\, \text{maps-to}(c_j) \neq h_k}} \gamma(\{c_i, c_j\})$$

$$\gamma([c_i, c_j]) = \begin{cases} \lambda(type([c_i, c_j])) \times \phi & \exists[\text{maps-to}(c_i), \\ & \text{maps-to}(c_j)] \in E_H \\ \lambda(type([c_i, c_j])) & \end{cases}$$

In case a source dependency causes unwanted coupling, the function returns its dependency weight without any modification. Source dependencies that result in allowed coupling, however, are multiplied with the parameter $\phi \in [0, 1]$ (recall that $toOthers$ is subtracted). When ϕ is set to its upper bound of 1, the conceptual information is ignored. The modified function $CountAttract'(c_i, h_k)$ is then equivalent to its initial version. A higher value of ϕ is not reasonable. With ϕ exceeding its upper bound, a source dependency that causes allowed coupling would decrease the attraction more than a source dependency that causes unwanted coupling. If ϕ is set to its lower bound of 0, source dependencies creating unwanted coupling have no negative effect on the attraction value. In the case study presented in this paper, we have applied different ϕ values to examine their effect on the clustering quality.

MQAttract

The clustering approach of the Bunch Tool [25], which follows the idea of maximal coherency and minimal coupling, applies various optimization clustering algorithms until the modularization quality of the cluster set is maximal [23]. One feature of the Bunch clustering technique is the later assignment of free entities into an existing set of clusters [25]. In this incremental clustering approach, a free entity is sequentially assigned to each cluster. From there the overall modularization quality is calculated. Finally, the free entity is assigned to the cluster for which the modularization quality is best. Hence, the modularization quality function serves as a similarity function for the incremental clustering algorithm of Bunch. For an efficient calculation of the modularization quality for a cluster set, $TurboMQ$ has been proposed [25].

For the task of semi-automated mapping, we have transformed $TurboMQ$ into a second attraction function. The attraction of a concrete entity c_i and a hypothesized entity h_j is defined by the $MQAttract$ function under the assumption that c_i is assigned to h_j.

$$MQAttract(c_i, h_j) = \sum_{k=1}^{|N_H|} CF_k \quad and \; \text{maps-to}(c_i) = h_j$$

$$CF_k = \begin{cases} 0 & \mu_k = 0 \\ \dfrac{2\mu_k}{2\mu_k + \sum\limits_{\substack{j=1 \\ j \neq k}}^{|N_H|} (\epsilon_{k,j} + \epsilon_{j,k})} & \text{otherwise} \end{cases}$$

To complete the attraction function $MQAttract$, the parameters μ, describing the internal dependencies and ϵ, describing external edges have to be translated into the terminology of the Software Reflexion Model technique.

$$\mu_k = \sum_{\substack{\forall [c_l, c_m]: \\ \text{maps-to}(c_l) = \text{maps-to}(c_m) = h_k}} \lambda(type([c_l, c_m]))$$

$$\epsilon_{i,j} = \sum_{\substack{\forall [c_l, c_m]: \\ \text{maps-to}(c_l) = h_i \wedge \text{maps-to}(c_m) = h_j}} \lambda(type([c_l, c_m]))$$

The integration of conceptual information into the $MQAttract$ function follows the same pattern as the one in the $CountAttract$ attraction. The adapted attraction function $MQAttract'$ equals the summation of the modified cluster factor CF' of each hypothesized entity in the case that c_i is mapped to h_j.

$$MQAttract'(c_i, h_j) = \sum_{k=1}^{|N_H|} CF'_k \quad and \; \text{maps-to}(c_i) = h_j$$

$$CF'_k = \begin{cases} 0 & \mu_k = 0 \\ \dfrac{2\mu_k}{2\mu_k + \sum\limits_{\substack{j=1 \\ j \neq i}}^{|N_H|} (\varphi_{k,j} + \varphi_{j,k})} & \text{otherwise} \end{cases}$$

In the initial cluster factor CF, the parameter ϵ represents the inter-dependencies between hypothesized entities. In the calculation of the modified cluster factor CF', inter-dependencies are represented by the parameter φ including conceptual information.

$$\varphi_{i,j} = \begin{cases} \lambda(type([c_i, c_j])) \times \phi & \exists[\text{maps-to}(c_i), \\ & \text{maps-to}(c_j)] \in E_H \\ \lambda(type([c_i, c_j])) & \text{otherwise} \end{cases}$$

As in the $CountAttract'$ function, the parameter $\phi \in [0, 1]$ controls the impact of the conceptual information on the attraction. If ϕ is set to the upper bound of 1, conceptual information is ignored and CF' equals CF. Source dependencies that are confirmed by a conceptual dependency do not decrease the similarity if ϕ is set to 0. Similar to the $CountAttract'$ attraction, exceeding these boundaries is not helpful. A ϕ value larger than 1 might result in a negative fraction and thus lead to a negative attraction value. A ϕ value lower than 0 results in a situation where source dependencies that cause allowed coupling penalize the attraction more than source dependencies that result in forbidden coupling.

3.4 Candidate Detection

Clustering techniques such as [25, 17, 2, 1, 41] assign an entity to the cluster that shares the highest similarity with it. In the case of multiple maximal similarity values, a cluster is often picked arbitrarily. These arbitrary decisions are likely to have ripple effects in the course of clustering. To mitigate these effects, our clustering algorithm maps only those entities which feature a significantly high affinity for a single hypothesized entity. The *candidate detection technique*, as part of the clustering algorithm, detects hypothesized entities whose attraction values "stick out" positively and marks them as *candidate hypothesized entities* using statistical analysis. The function *cand* results in a candidate set that contains all candidate hypothesized entities for each free concrete entity. The result of this function equals one of two different temporary sets, namely $candSet1_i$ and $candSet2_i$. Both of these sets contain candidates for the concrete entity c_i, however, they apply a different threshold for the attraction value. The threshold of the temporary candidate set $candSet1_i$ is hereby more restrictive than the threshold applied for the calculation of $candSet2_i$.

$$cand(c_i) = \begin{cases} candSet1_i & |candSet1_i| > 0 \\ candSet2_i & \text{otherwise} \end{cases}$$

The temporary candidate set $candSet2_i$ contains all hypothesized entities that have an attraction toward concrete entity c_i equal to or larger than the arithmetic mean over all attraction values of c_i. Therefore, the temporary candidate set $candSet2_i$ is defined as follows:

$$candSet2_i = \{h \in N_H \mid attraction(c_i, h) \geq \bar{x}_i\}$$

$$\bar{x}_i = \frac{1}{|N_H|} \sum_{j=1}^{|N_H|} attraction(c_i, h_j)$$

The candidate set $candSet2_i$ is likely to contain a high number of candidate hypothesized entities. The intention of the candidate detection process is, however, to reduce the number of candidates as much as possible. In order to filter more concrete entities, the calculation of the set $candSet1_i$ applies a more restrictive approach. The standard deviation sd_i describes how much a typical attraction value varies from the arithmetic mean.

$$sd_i = \sqrt{\frac{1}{|N_H|} \sum_{j=1}^{|N_H|} (attraction(c_i, h_j) - \bar{x}_i)^2}$$

In other words, the standard deviation defines a margin around the arithmetic mean, in which most attraction values are located. Attraction values that are located outside this band, are considered mavericks. Hypothesized entities that possess a positive maverick to c_i are a member of $candSet1_i$.

$$candSet1_i = \{h \in N_H \mid attraction(c_i, h) \geq \bar{x}_i + sd_i\}$$

In some cases, $candSet1_i$ may not contain any candidates due to the dispersion of the attraction values. The temporary candidate set $candSet2_i$, however, always contains at least one candidate. Considering the definitions of the temporary candidate sets $candSet1_i$ and $candSet2_i$, it is trivial to prove that $candSet1_i$ is a subset of $candSet2_i$.

$$N_H \supseteq candSet2_i \supseteq candSet1_i$$

Regardless of which temporary candidate set serves as the final candidate set, the search space for the user is always reduced. Only if all hypothesized entities hold the same attraction to a concrete entity, all hypothesized entities are considered candidates.

3.5 Automated Mapping

The actual clustering (i.e., automated mapping) takes place in the last phase of the algorithm. After the candidate sets have been calculated for each unmapped concrete entity, all concrete entities are automatically mapped for which only a single candidate could be detected. In other words, only those mapping decisions for which a decision is easy enough to be made without consulting the user. In all other cases, the candidate sets are presented to the user for a manual clustering decision.

4 Case Study

This section reports on a case study to evaluate the semi-automated mapping and the underlying clustering

alternatives. The system analyzed in this case study is the core of SHriMP [44]. SHriMP is a Java application to visualize graph-based information. The core system that was examined comprises roughly 300 classes and interfaces which share about 7000 source dependencies. The hypothesized view of SHriMP was created by a developer who had several years experience in the development and maintenance of the SHriMP system. The created hypothesized view contained 10 hypothesized entities and 13 hypothesized dependencies. The concrete view on SHriMP was extracted from its source code using the extraction tool for Java code from the Bauhaus toolkit[1]. After the concrete view and the hypothesized view had been created, the developer who had previously modeled the architecture of the system, mapped all concrete entities to their corresponding subsystem. This complete and correct map later served as a reference for the automatic clustering decisions of our algorithm.

In preparation for the application of the clustering algorithm, the mapping relationships of 218 concrete entities were deleted randomly. Thus, only 28 percent of the previously 302 mapped concrete entities remained mapped to their correct hypothesized entity. The concrete entities which were "unmapped" from their correct hypothesized entity were chosen arbitrarily considering only one constraint: each hypothesized entity should have at least one concrete entity mapped to it. Then our clustering algorithm was applied in order to automatically recreate the deleted mappings. Both attraction functions were applied in multiple experiments using different ϕ values. After each experiment, the initial partial map was reset so that the results of the experiments would be comparable. The set of dependency weights that was applied in the experiments was taken from the work of Rayside *et al.* [31]. Note that investigating the effect of different edge-weight sets on the clustering was not part of the case study.

Table 4 and 4 summarizes the results. Several observations can be made. The most important observation is that both attraction functions can be applied successfully to automatically map concrete entities with a high mapping quality. Although the MQAttract' achieved the highest mapping quality of 100 percent in one experiment, we consider the CountAttract' attraction function to be more suitable for semi-automated mapping for several reasons:

The CountAttract' attraction function resulted in an acceptably high cluster quality in each of the experiments. Especially when conceptual information was considered; about 92 percent of the automated map-

ping decisions were correct. The MQAttract' function, on the other hand, produced a high cluster quality only when conceptual information had a maximal impact on the calculation. In the case where conceptual information was ignored, the clustering quality was unacceptable and even lower than a random mapping decision. Moreover, if the conceptual information at hand is inaccurate, or the conceptual dependencies are unknown to the user and thus not included in the hypothesized view, the mapping decisions when MQAttract' is applied are questionable.

In the experiments, the quality of the technique was highly dependent on the applied attraction function. In the case of the MQAttract' attraction function, the support for the user was negligible. Although the correct cluster was considered a candidate in most of the cases, the reduction of the search space was not effective. For most concrete entities the search space was reduced to only about 6 or 7 of the 10 hypothesized entities. The supportive aspect, however, worked well when the CountAttract' attraction function provided the attraction values. The search space was reduced in most cases down to 2 or 3 candidates. In addition, in most of the cases the correct hypothesized entity was among the detected candidates.

The third benefit of the CountAttract' function is that its calculated values allow for an easier human interpretation. The generated attraction values showed high variation, which allowed for an easier comprehension of the derivation of attraction values. In contrast, the MQAttract' function always produced values between 9.9 and 10. For most concrete entities, the attraction values differed only after the third decimal. For human judgement, it was very difficult to separate the hypothesized entities with high attraction from those with a low affinity to the concrete entity. Finally, the complex definition of the MQAttract' function makes it very hard for the user to verify the calculated attraction value. The values produced by the CountAttract' function, however, were much easier to verify manually.

In general, the parameter ϕ (i.e., the impact of conceptual information) has a significant impact on the quality and quantity of the automated mapping decisions. It was observed that the number of automatic mapping decisions decreased as the influence of the conceptual information on the attraction value increased. The quality of these mapping decisions, however, also improved the more influence the conceptual information gained.

---

[1]http://www.bauhaus-stuttgart.de

	CountAttract'			MQAttract'		
	$\phi = 1$	$\phi = 0.5$	$\phi = 0$	$\phi = 1$	$\phi = 0.5$	$\phi = 0$
concrete entities passed filter	53	53	53	53	53	53
entities mapped	49	39	25	30	28	14
correctly mapped	33	31	23	2	5	13
percentage correctly mapped	67%	79%	92%	7%	17%	93%
not mapped	4	14	28	23	25	39
correct con. is candidate	2	12	27	23	25	39
average no. of candidates	2	2.1	2.6	6.7	6.6	5.2
percentage correct support	50%	86%	96%	100%	100%	100%

Table 1. Mapping Results

5 Restrictions

Both attraction functions derive the attraction values from source relationships between concrete entities and hypothesized dependencies between hypothesized entities. This approach shares the same drawbacks as other clustering techniques based on source dependencies. Similar to those techniques, our clustering algorithm yields hypothesized entities featuring high cohesion and low coupling. As stated by Andritsos *et al.* [2], this approach is problematic when the developers of the system did not follow the principle of low coupling and high cohesion. Especially in a system that needs to be maintained, the intended architecture is likely to be repeatedly violated. Moreover, subsystems such as libraries naturally do not follow the concept of high cohesion and low coupling and would be therefore hard to detect. In order to mitigate this problem, we suggest the approach by Bauer *et al.* [4]. In their semi-automated clustering technique, a separate library detection is applied prior to the actual clustering. The detected libraries would then not be considered for a later automatic mapping.

6 Conclusions

Our case study demonstrates the support of clustering techniques for establishing the reflexion mapping. The clustering technique was able to achieve a mapping quality where more than 90 percent of the automatic mapping decisions turned out to be correct. Moreover, the experiments indicate that the coupling-based attraction function (CountAttract') is more suitable for semi-automated mapping than the approach based on coupling and cohesion (MQAttract'). Especially in combination with the CountAttract' attraction function, the candidate detection technique performed remarkably well on both tasks of semi-automated mapping. In addition to the high quality of the automated mapping decisions, the search space for the user's search for the correct entity was reduced significantly.

References

[1] F. Abreu, G. Pereira, and P. Sousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *CSMR*. IEEE Press, 2000.

[2] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *WCRE*, pages 334–343. IEEE Press, Nov. 2003.

[3] N. Anquetil and T. Lethbridge. Extracting concepts from file names: a new file clustering criterion. In *ICSE*, pages 84–93. ACM Press, 1998.

[4] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. In *CSMR*, pages 3–12. IEEE Press, Mar. 2004.

[5] D. Bojic and D. Velasevic. A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *CSMR*. IEEE Press, 2000.

[6] G. Canfora, A. Cimitile, A. D. Lucia, and G. A. D. Lucca. A case study of applying an eclectic approach to identify objects in code. In *IWPC*, pages 136–143. IEEE Press, 1999.

[7] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Journal of Software Practice and Experience*, 26(1):25–48, Jan. 1996.

[8] S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, Jan. 1990.

[9] A. Christl. Semi-automated mapping for the reflexion method. Diploma thesis no. 2160, University of Stuttgart, Computer Science, Jan. 2005.

[10] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems and Software*, 28:117–127, 1995.

[11] C. M. de Oca and D. L. Carver. A visual representation model for software subsystem decomposition. In *WCRE*. IEEE Press, 1998.

[12] H. Gall and R. Klösch. Finding objects in procedural programs: an alternative approach. In *WCRE*, pages 208–217. IEEE Press, July 1995.

[13] J.-F. Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *ICSM*. IEEE Press, 1997.

[14] J.-F. Girard, R. Koschke, and G. Schied. A metric-based approach to detect abstract data types and state encapsulations. *Journal Automated Software Engineering*, 6(4), 1999.

[15] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Object Technology Series. Addison Wesley, 2000.

[16] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE TSE*, 11(8):749–757, Aug. 1985.

[17] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. Ph.d. thesis, University of Stuttgart, `http://www.iste.uni-stuttgart.de/ps/rainer/thesis`, Oct. 1999.

[18] R. Koschke and D. Simon. Hierarchical reflexion models. In *WCRE*. IEEE Press, 2003.

[19] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *ICSE*, pages 349–359. IEEE Press, 1997.

[20] S. S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *ICSM*, pages 266–271. IEEE Press, Nov. 1990.

[21] P. Livadas and T. Johnson. A new approach to finding objects in programs. *Journal Software Maintenance and Evolution*, 6:249–260, 1994.

[22] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *ICSM*, pages 315–324. IEEE Press, Sept. 2003.

[23] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC*. IEEE Press, 1998.

[24] O. Maqbool and H. A. Babri. The weighted combined algorithm: A linkage algorithm for software clustering. In *CSMR*, pages 15–24. IEEE Press, Mar. 2004.

[25] B. S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. Dissertation, Drexel University, Philadelphia, PA, United States, 2002.

[26] H. A. Müller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software development environments*, pages 88–98. ACM Press, Dec. 1992.

[27] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 30(8):29–36, Aug. 1997. Reprinted in Nikkei Computer, 19, January 1998, p. 161-169.

[28] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. of the Third ACM SIG-SOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, 1995. ACM Press.

[29] R. M. Ogando, S. S. Yau, and N. Wilde. An object finder for program structure understanding in software maintenance. *Journal Software Maintenance and Evolution*, 6(5):261–283, September-October 1994.

[30] S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *ICSE*, pages 38–48. ACM Press, 1992.

[31] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In *IWPC*. IEEE Press, 2000.

[32] H. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying concept formation methods to object identfication in procedural code. In *ASE*, pages 210–218. IEEE Press, Nov. 1997.

[33] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In *IWPC*, pages 115–117. IEEE Press, May 2001.

[34] K. Sartipi and K. Kontogiannis. A graph pattern matching approach to software architecture recovery. In *ICSM*, pages 408–417. IEEE Press, Nov. 2001.

[35] R. W. Schwanke and S. J. Hanson. Using neural networks to modularize software. *Machine Learning*, 15:136–168, 1994.

[36] A. Shokoufandeh, S. Mancoridis, and M. Maycock. Applying spectral methods to software clustering. In *WCRE*, pages 3–12. IEEE Press, Oct. 2002.

[37] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE TSE*, 25(6):749–768, November/December 1999.

[38] P. Tonella. Concept analysis for module restructuring. *IEEE TSE*, 27(4):351–363, Apr. 2001.

[39] V. Tzerpos. The orphan adoption problem in architecture maintenance. In *WCRE*. IEEE Press, 1997.

[40] R. R. Valasareddi and D. L. Carver. A graph-based object identification process for procedural programs. In *WCRE*, pages 50–58. IEEE Press, Oct. 1998.

[41] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE*, pages 246–255. IEEE Press, 1999.

[42] J. Weidl and H. Gall. Binding object models to source code: An approach to object-oriented re-architecturing. In *Proc. of the 22nd Computer Software and Applications Conference*. IEEE Press, 1998.

[43] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *WCRE*, pages 33–43. IEEE Press, 1997.

[44] J. Wu and M.-A. D. Storey. A multi-perspective software visualization environment. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 15. IBM Press, 2000.

[45] A. Yeh, D. H. D., and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *WCRE*, pages 227–236. IEEE Press, 1995.