Comparing Attract Functions for the Reflexion Analysis Regarding the Usage of Dependencies and Words

Leon Ehrhardt
University of Bremen, Germany
leeeon333@gmail.com

Rainer Koschke
University of Bremen, Germany
https://orcid.org/0000-0003-4094-3444

***Abstract*—In this paper, we propose a new attraction function to be used to make mapping suggestions in the context of the reflexion analysis. The reflexion analysis allows developers to reconstruct or verify their architecture with regards to its implementation. To be able to do that, the reflexion analysis requires a mapping of source-code elements onto architecture components. This mapping can be a labor-intensive manual task. The attraction functions may ease this task by recommending mappings.**

Our method draws on existing methods and combines dependency information with term similarity aggregated by way of dependencies. We describe an evaluation that compares our new method against three other existing methods, namely, Count Attract and NB-Attract. All these attraction functions are applied within the framework of the HugMe method. The computed suggestions for all methods are compared against oracle mappings provided by other researchers. As measures for the comparison, we use recall and precision relative to an oracle. Our results show that our new method can be applied successfully and performs similar to NB-Attract. Furthermore, whether the usage of dependencies or terms is more suitable for high-quality mappings seems rather system dependent, even though the attract functions using terms are performing better more often. Our method, as a combination of dependency and term similarity, could be particularly well suited in situations in which one does not want to rely on only dependencies or only term similarity.

***Index Terms*—reflexion modeling, semi-automatic mapping, heterogenous data, orphan adoption problem, SEE, attract functions, architecture recovery, HugMe-Method.**

I. Introduction and Motivation

The alignment between a software system's implementation and its architectural description is a recognized practice to enhance the quality of software development. It enables verification of the system's structure, facilitates comprehensive documentation, and aids in understanding the design. However, due to the extended lifecycle of software systems, it is not uncommon for architectural descriptions to be incomplete, outdated, or entirely absent. The concept of reflexion modeling [1] can be used to maintain and re-establish consistency between dependencies within the implementation and rules imposed by an architectural description where both architecture and implementation are represented as dependency graphs. Based on a mapping of implementation modules onto architecture components, the consistency of dependencies within a software system can be checked.

The mapping of implementation modules onto architecture components is typically a manual and labor intensive task [2]. To address this, automated techniques employing *Attract Functions* have been proposed [3]–[8]. These functions calculate a value expressing how well an architecture component attracts a not yet mapped implementation component. They can be used to suggest plausible mappings based on predefined criteria, allowing for automated mapping suggestions when matches surpass certain thresholds of reliability.

Most existing Attract Functions typically leverage either structural information derived from the dependeny graphs or textual information extracted from the source code, although there are also hybrid approaches [6]. Yet, approaches that effectively combine both types of information have been underexplored. In particular, hybrid methods aggregating textual similiarity over dependency edges have not been investigated yet. Therefore this work presents a novel hybrid approach for computing mapping suggestions and compares it to existing structural, text-based, and hybrid methods.

II. Related Work

In the following we present related research with a focus on attract functions in the context of the mapping step as part of the reflexion model. We discuss related research on reflexion modeling in general and also on how the automated mapping suggestions can be integrated into an interactive approach in another paper [9].

A. Reflexion Modeling and the Orphan Adoption Problem

Reflexion modeling was initially applied in a case study using Microsoft Excel to demonstrate its applicability to complex systems [10]. In this study, over 400 modules were manually mapped onto architectural components, showcasing the challenge of large-scale system analysis. Concurrently, the orphan adoption problem was introduced [11], focusing on the automatic assignment of individual implementation modules to existing architecture components. This problem centers on reducing *coupling* between newly assigned modules and their surrounding modules, providing a selection criterion

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – KO 2342/8-1.

for module assignment. Unlike approaches that assign multiple modules simultaneously, Orphan Adoption emphasizes incremental individual assignments, enabling iterative semi-automatic clustering through suggested mappings.

B. Dependency-Based Approaches

One approach to address the Orphan Adoption problem in the context of reflexion modeling is the *HugMe* method [3]. This approach uses attraction values between unassigned implementation elements and existing architectural components. Using statistical thresholds, the method identifies architectural candidates for individual implementation elements. If only one candidate is selected, the assignment can proceed automatically.

Attraction values in *HugMe* are calculated through *Attract functions*, which depend on various selection criteria. For instance, the authors revisited coupling minimization in the *Count-Attract* function, considering desired couplings specified via reflexion graphs, scaled by a parameter [3]. Another significant contribution was *MQ-Attract*, inspired by the TurboMQ algorithm [12], which optimizes module cohesion while minimizing coupling. Beyond these approaches, supervised learning algorithms such as *naive Bayes*, *k-nearest neighbors*, and *neural networks* have been employed to guide candidate selection based on weighted dependencies [4]. These techniques underline the versatility of dependency data in architecture recovery tasks.

C. Text-Based and Hybrid Approaches

In addition to structural dependencies, textual information extracted from source code, such as identifiers, comments, and file names, has been explored for clustering. Information retrieval (IR)-based approaches, including *IR-Attract* and *LSI-Attract*, leverage term frequencies to evaluate similarity between candidates and clusters [5]. While IR-Attract uses cosine similarity, LSI-Attract additionally employs latent semantic indexing to reduce term dimensionality before clustering. More recently, *NB-Attract*, a multinomial naive Bayes classifier, has demonstrated improved performance by incorporating textual features into candidate selection [6]. When combined with the method *Concrete Dependency Abstraction (CDA)*, which generates additional terms from structural dependencies, NB-Attract integrates dependency and textual data to enhance the mapping quality. However, as these functions depend upon an initial set of assignments, they may not always be suitable for early-stage clustering.

Methods such as *InMap* [7] and *RELAX* [8] extend text-based clustering by indexing source code terms and associating modules with thematic topics. Despite their strengths, these approaches are not tied to reflexion graphs and require additional text annotations, limiting their applicability to automated processes if no such textual annotations are available Hybrid methods that combine text and dependencies often concatenate Attract functions or apply CDA to boost the F1-score [13]. Nonetheless, a direct integration of reflexion graphs into hybrid Attract functions remains unexplored.

D. Summary

The NB-Attract function currently outperforms alternatives, followed by Count-Attract, which often surpasses MQ-Attract and IR-based functions in previous studies [5] [6]. Other hybrid methods, InMap and RELAX, depend on external annotations, making them less suitable for reflexion modeling if no annotations are available. Our work builds upon NB-Attract and Count-Attract, aiming to introduce a novel approach that integrates both dependency and textual data to resolve the orphan adoption problem more effectively. While alternative classifiers could replace naïve Bayes, studies suggest minimal performance differences within this context [14].

III. Theoretical foundations

In this section, we formalize different existing attract functions and our new variant. We will first lay the formal foundation for reflexion modeling.

A. Basic definitions

The input to reflexion analysis are two directed typed and hierarchical dependency graphs and a mapping in between them.

The first graph, $G_I = (N_I, E_I \subseteq N_I \times N_I)$, describes the implementation, where N_I consists of declarations of methods, fields, classes, packages and the like in the source code. Which types of nodes occur, depends on the programming language. The directed edges E_I represent dependencies among these, such as calls or field accesses. This graph is typically generated by a static program analysis.

Likewise, the architecture model is also described as a $G_A = (N_A, E_A \subseteq N_A \times N_A)$ whose nodes represent architectural components (modules, layers, etc.) and the edges expected dependencies among these. This graph is typically crafted manually by software architects.

In the following, we use the convention that symbols c_i will be used for nodes in N_I and symbols a_i for nodes in N_A. If we talk about a node that can be in either of the two graphs, we simply use symbols n_i. Similarly, we use E, which can be E_I or E_A, if the distinction is irrelevant.

The nodes and edges in both graphs are typed according to an underlying type hierarchy, as we want to distinguish between classes and methods, for instance. We are using the function *type-of* to denote the type of a given node or edge, respectively. The type hierarchy of node and edge types is defined by a predicate t_1 *is-a* t_2 denoting that type t_1 is a (transitive) subtype of t_2. For instance, a Java dispatching call is a special kind of call, and a static call is a special kind of call, too. We assume a general edge type *dependency* which all other edges type derive from. The type hierarchy allows an architect to be specific or general if needed. For instance, the architect could allow a call between modules but no field accesses. If the exact dependency does not matter, the architect can simply use *dependency* for the edge type.

Some attract functions use weights for the importance of edge types, which we model through a function $\lambda : E \rightarrow \mathbb{R}_0^+$:

$$\lambda(e) = \mathit{weight}(\mathit{type\text{-}of}(e))$$

based on a user-provided function *weight* weighing the importance of an edge type. For instance, a user might find a call be more important than an import dependency.

All edges within the graphs are directed. Whenever we want to abstract from the direction, we will use the notation $\{n, n'\}$ in the following; that is $\{n, n'\}$ can refer to (n, n'), (n', n) or both.

Both graphs can be hierarchical, that is, their nodes may contain other nodes, which we model by way of a partial function *parent-of* that yields the parent node of a given child node. The function is partial because a root node has no parent. Based on this node hierarchy, we define the function *descendants-of* that returns, for a given node n, the set of all transitive child nodes within the subtree rooted by n including n itself.

The function *edges-of* describes the set of all incoming and outgoing edges of the descendants of n. One can think of the function as a way to lift edges of descendants up to n to consider all adjacent dependencies within the hierarchy.

$$\textit{edges-of}(n) = \{\{n', n''\} \in E \mid n' \in \textit{descendants-of}(n)\}$$

To compute the reflexion model, nodes of the implementation, N_I, need to be mapped onto nodes of the architecture, N_A. For that, we assume a set $M \subseteq N_I \times N_A$, where every element $(c, a) \in M$ expresses that c is mapped onto a. We are using a set instead of a partial function just because that eases the addition of new mappings in the description later on. This mapping is to be provided by developers and architects. The goal of the attract functions is to propose for an unmapped $c \in N_I$ additions (c, a) to M. Unmapped means that $\nexists(c, a') \in M$ yet (for any a', be it a or different from it).

Set M establishes an explicit partial mapping. In case of hierarchical nodes in N_I, a mapping can be implicit, too [15]. For instance, if a user maps a class, all its methods and fields can be considered mapped implicitly, too, if they are not yet mapped elsewhere themselves. To handle that, we define a function *maps-to* to describe the complete mapping relative to M, which considers the node hierarchy of the implementation graph (where $\perp$ is used to denote *undefined*):

$$\textit{maps-to}(c, M) =$$

$$\begin{cases} a : \exists(c, a) \in M \\ \textit{maps-to}(\textit{parent-of}(c)) : \nexists(c, a) \in M \\ \perp : \text{otherwise} \end{cases}$$

Any implementation component, c, where $\textit{maps-to}(c, M) = \perp$ holds, is called an *orphan*. Based on this mapping, the outcome of the reflexion analysis can be defined. The analysis expresses its results in terms of a function $\textit{state-of} : E \to \{\textit{convergent}, \textit{absent}, \textit{divergent}, \textit{allowed}, \perp\}$. The states *divergent* and *allowed* are reserved for edges in E_I to express whether an implementation dependency, $(c, c') \in E_I$, is allowed according to the architecture or not:

$$\textit{state-of}((c, c'), M) =$$

$$\begin{cases} \textit{allowed} : & \begin{array}{l} \exists(a, a') \in E_A : \\ \textit{maps-to}(c, M) = a \\ \wedge\, \textit{maps-to}(c', M) = a' \\ \wedge\, \textit{type-of}((c, c'))\ \textit{is-a}\ \textit{type-of}((a, a')) \end{array} \\[2ex] \textit{divergent} : & \begin{array}{l} \nexists(a, a') \in E_A : \\ \textit{maps-to}(c, M) = a \\ \wedge\, \textit{maps-to}(c', M) = a' \\ \wedge\, \textit{type-of}((c, c'))\ \textit{is-a}\ \textit{type-of}((a, a')) \end{array} \\[2ex] \perp : & \textit{maps-to}(c, M) = \perp \vee \textit{maps-to}(c', M) = \perp \end{cases}$$

At this point it should be noted that a user of reflexion modeling would generally assume an implicit self-loop for all architecture nodes, which means that all implementation components are allowed to depend upon other implementation components mapped onto the same architecture component. This allowance is not covered in our definition above to keep the description simple.

The state of an architecture dependency, $(a, a') \in E_A$, can be either *convergent* or *absent*. An architecture dependency is considered *convergent* if there is an allowed implementation dependency for it; otherwise it is considered *absent*. The exact formalization is irrelevant for the purpose of this paper. An interested reader is referred to our paper where we introduce hierarchical reflexion modeling [15].

Putting it altogether, a reflexion analysis has the input (G_I, G_A, M) and implements the function *state-of*.

In Figure 1, an example of a reflexion model is presented, illustrating the definitions so far. The blue circles represent implementation components and the yellow squares architecture components. The edge (c_2, c_4) is in the state *allowed*, due to the allowing edge (a_1, a_2). The implicit self-dependency (a_1, a_1) allows the edge (c_1, c_2). The mapping of c_2 to a_1 is transferred to c_3 via the function *parent-of*. Consequently, the edge (c_3, c_4) is also allowed. The edges (c_5, c_4) and (c_5, c_3) are in the state *divergent*, as no allowing edges (a_3, a_1) and (a_3, a_2) exist. The edge (a_2, a_3) is in the state *absent*, as no expected implementation edge exists. This could be resolved by adding a dependency between (c_4, c_5).

Next we will introduce the attract functions, with the codomain $\mathbb{R}_0^+$, that calculates the attraction of an orphaned implementation component relative to an architecture component. These function are used to select the mapping candidates.

B. Count-Attract

The definition of Count-Attract aims to minimize coupling during the process of mapping, but also considers a scaling factor to consider coupling prescribed by the reflexion model. Our definitions in the following are based on our prior work [3].

We first define the *overall* coupling of a given orphan, $c \in N_I$, to its neighbors by summing up the edge weights for all

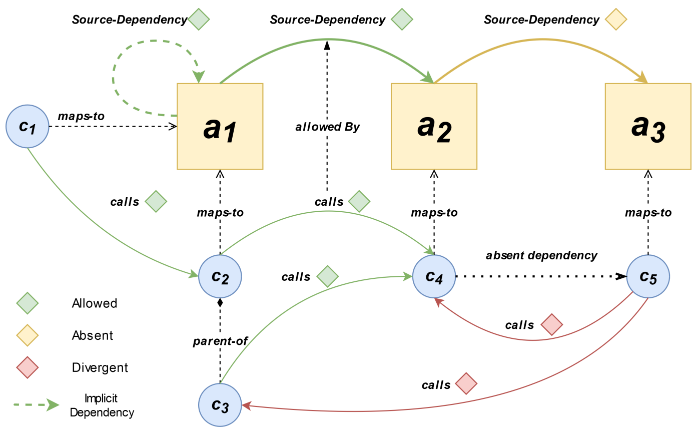

Fig. 1. Example of a reflexion model

its direct neighbors that are already mapped:

$$overall(c, M) = \sum_{\substack{\forall\{c,c'\}\in \text{edges-of}(c):\\ maps\text{-}to(c',M)\neq\bot}} \lambda(\{c, c'\})$$

Next we will calculate coupling under the assumption that orphan, c, were mapped onto an architecture node, a. Here we want to ignore all incoming and outgoing implementation dependencies that are allowed according to the architecture since they are an expected dependency. This can be accommodated by a function *toOthers* that ignores coupling to neighbors mapped onto a different architecture component (dependencies to implementation components mapped to the same architecture component are allowed by default):

$$toOthers(c, a, M) =$$

$$\sum_{\substack{\forall\{c,c'\}\in \text{edges-of}(c):\\ maps\text{-}to(c',M)\neq\bot \wedge maps\text{-}to(c',M)\neq a}} scale(\{c, c'\}, M \cup (c, a))$$

The logic whether and how dependencies are weighed is specified by function *scale* in the equation above. If the edge passed as parameter is allowed under the condition *maps-to*$(c) = a$, a scaling factor, ϕ is applied. This parameter can vary between 0 and 1 and is determined by the user. If set to 0, the allowed dependencies are ignored completely; if set to 1, they have the same weight as divergent dependencies. Users would select a value closer to 0 (see below).

$$scale((c, c'), M) = \begin{cases} \lambda((c, c')) \times \phi : & state((c, c'), M) = \\ & allowed \\ \lambda((c, c')) : & \text{otherwise} \end{cases}$$

Function *toOthers*—taking into account desired coupling represented in allowed dependencies—can be used to reduce the *overall* coupling. Thus, *Count-Attract* can be formulated as the difference between the two functions;

$$Count\text{-}Attract(c, a, M) = overall(c, M) - toOthers(c, a, M)$$

Because *toOthers* is subtracted from *overall*, implementation dependencies creating expected coupling have no negative effect on the attraction value if $\phi = 0$.

Count-Attract can be used to rank the target architecture components for a given orphan. The best candidate would be the one with the highest *Count-Attract* value.

C. *NB-Attract*

The function *NB-Attract* is based on the works of Olsson et al. [6]. The *NB-Attract* function is a naïve multinominal Bayesian classifier and uses the oberservation of words and mappings to calculate the attraction between an implementation node and an architecture node. The function treats a given architecture node as a class and calculates a probability that a given implementation node belongs to it. The function is composed of the prior probability an architectural node is chosen—calculated based on the current mapping—and the likelihood an implementation node belongs to an architectural component—based on the usage of words. Like other naïve Bayesian classifiers, *NB-Attract* assumes that the probabilities to observe words are independent of each other. Even though this assumption will generally not hold in our context, naïve Bayesian classifiers still often perform well in practice ignoring this assumption [16]. Originating from Bayes' theorem and the assumption that the occurrence of evidence for all classes has the same constant probability, we can assume the following proportionality $\propto$:

$$NB\text{-}Attract(c, a, M) = P(a \mid c, M) \propto P(a, M) \times P(c \mid a)$$

The function *terms-of* returns a vector representing the frequencies of words associated with a given node. *NB-Attract* considers only whether a word is present, but does not consider its frequency. For a given architecture node, the function accumulates the words of all implementation nodes mapped to the architecture node, regarding the frequency of words, if they appear across implementation nodes. We define $\mathbb{W}$ as a set containing all words present in the system, thus *terms-of* is a function from nodes onto a $|\mathbb{W}|$ dimensional vector space:

$$terms\text{-}of : N \mapsto \mathbb{R}^{|\mathbb{W}|}$$

The prior probability $P(a, M)$ is based on the probability an architecture node is chosen for an implementation node regarding all mapped implementation nodes:

$$P(a, M) = \frac{|\{c \in N_I \mid maps\text{-}to(c, M) = a\}|}{|\{c \in N_I \mid maps\text{-}to(c, M) \neq \bot\}|}$$

The likelihood of an implementation node given an architecture node is the product of the probabilities of each word associated with the implementation node appearing in the given class. The probability for a word per class is calculated by the occurrences of the word within the class divided by the number of all words occurred within the class. To compensate words unknown by the class, alpha smoothing with $\alpha = 1$ is applied (*terms-of*$(c)[w]$ yields the value for w in the vector *terms-of*(c)):

$$P(c \mid a) =$$

$$\prod_{\forall w \in \mathbb{W}: \textit{terms-of}(c)[w]>0} \textit{terms-of}(c)[w] * \frac{\textit{terms-of}(a)[w] + \alpha}{|\textit{terms-of}(a)| + \alpha * |\mathbb{W}|}$$

D. Combining Text and Dependencies

In traditional text-based assignment methods, similarities between candidates are typically determined by comparing their words. This approach assumes that the frequency and type of words used provide insights into the affiliation and similarity of candidates. Commonly, the words are assigned to architectural components based on which implementation components are mapped to them. However, when comparing implementation nodes for selection of an architecture node, words shared across architectural nodes may disrupt the comparison.

Originating from this idea, the desired coupling defined by the reflexion model shall be incorporated into the process. Instead of assigning words to architectural nodes, they are associated with architectural edges. Specifically, those words shall be linked to the architectural edge that are typical for the coupling of that part of the system. To achieve this, we want to aggregate all words within an architectural edge which are shared across implementation edges that are allowed by the architectural edge. The aggregation of such words within architectural edges is referred to here as *Abstract Dependency Concretization (ADC)*.

This approach enables a comparison between the neighboring edges of an implementation node and those of an architectural node, guided by the content of the reflexion model. It is important to note the implicit dependencies in the architecture, which would group words that contribute to cohesion. Consequently, the text from modules, desired couplings and cohesion are all considered during the assignment process.

We want to define ADC formally. First, let the function *allowed-by* return the allowing architecture edge for a given implementation edge and a current mapping M if the edge is allowed:

$$\textit{allowed-by}((c, c'), M) =$$

$$\begin{cases} (\textit{maps-to}(c, M), \textit{maps-to}(c', M)) : & \textit{state-of}((c, c'), M) = \\ & \textit{allowed} \\ \bot : otherwise & \end{cases}$$

We extend the function *terms-of* to describe the terms associated with an implementation edge. These terms are the intersection of the terms of both connected nodes, while summing up the frequency of remaining words.

$$\textit{terms-of}((c, c')) = v$$

where $v[w] =$

$$\begin{cases} \textit{terms-of}(c)[w] + \textit{terms-of}(c')[w] : & \textit{terms-of}(c)[w] > 0 \\ & \wedge\, \textit{terms-of}(c')[w] > 0 \\ 0 : otherwise & \end{cases}$$

Using both functions, the terms associated with an architecture edge can be defined. The terms of an architecture edge are the union of all terms associated with each implementation edge allowed by the architecture edge, by summing up their frequencies.

$$\textit{terms-of}((a, a'), M) =$$

$$\sum_{\substack{\forall (c,c') \in E_I: \\ \textit{allowed-by}((c,c'),M)=(a,a')}} \textit{terms-of}((c, c'))$$

Continuing, the *ADC-Attract* function can now be defined. The function compares the words of all allowed adjacent implementation edges of the candidate with the words of the corresponding allowing architecture edges using a distance function. For the mentioned implementation edges, the mapping $\textit{maps-to}(c) = a$ is assumed, because the state of edges adjacent to the candidate is required for the calculation.

$$\textit{ADC-Attract}(c, a, M) =$$

$$\sum_{\substack{\forall \{c,c'\} \in \text{edges-of}(c): \\ state(\{c,c'\}, M \cup (c,a)) = \\ allowed}} \lambda(\{c, c'\}) \times \textit{ADC-Attract}(\{c, c'\}, M)$$

where

$$\textit{ADC-Attract}((c, c'), M) =$$

$$overlap(\textit{terms-of}((c, c')), \textit{terms-of}((\textit{allowed-by}((c, c'), M), M))$$

For the distance function, the above overlap coefficient of two sets was chosen, as it showed better results in preliminary experiments compared to Cosine similarity, Euclidean distance, and Jaccard similarity [17]. In our case, the function *overlap* treats words represented by a vector as sets, regarding only the presence of words.

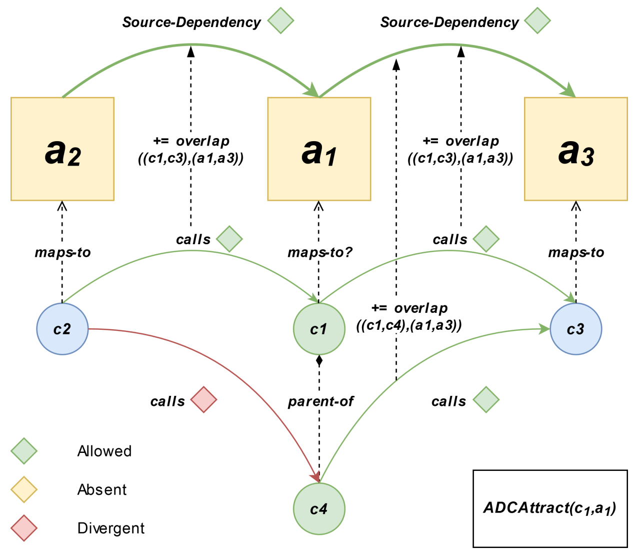

Fig. 2. Principal of *ADC-Attract*

As illustrated in Figure 2, *ADC-Attract* calculates the attraction between $c1$ and $a1$ by comparing the adjacent edges of $c1$ and its subtree with the allowing architectural edges. The overlap of the words of each allowed implementation edge with those words associated with the allowing architectural edges is summed up. We note that the states of the adjacent edges of $c1$ are determined, but their associated words have not yet been added to the allowing architectural edges during this computation.

E. HugMe Method

The *HugMe* method [3] is utilized to identify sets of architecture nodes for a given implementation node based on outstanding attraction, which can be used for automatic mappings or recommendations. The following functions are operating on a matrix representing all attraction values between every implementation node and every architecture node.

Mathematically, the method operates as follows:

$$\textit{HugMeSet}(c) = \begin{cases} \textit{HugMeSet1}(c) & \text{if } |\textit{HugMeSet1}(c)| > 0 \\ \textit{HugMeSet2}(c) & \text{otherwise} \end{cases}$$

where *HugMeSet1*(c) selects architecture nodes with attraction values sticking out statistically, that is, exceeding the sum of the mean $\overline{x}_c$ and standard deviation sd_c regarding all architecture nodes:

$$\textit{HugMeSet1}(c) = \{a \in N_A \mid \textit{attract}(c,a) \geq \overline{x}_c + sd_c\}$$

$$\overline{x}_c = \frac{1}{|N_A|}\sum_{i=1}^{|N_A|} \textit{attract}(c, a_i)$$

$$sd_c = \sqrt{\frac{1}{|N_A|}\sum_{i=1}^{|N_A|}(\overline{x}_c - \textit{attract}(c, a_i))^2}$$

The combination of means and standard deviation is to identify statistical outliers. If *HugMeSet1*(c) is the empty set, *HugMeSet2*(c) is consulted, which uses only the mean as the threshold:

$$\textit{HugMeSet2}(c) = \{a \in N_A \mid \textit{attract}(c,a) \geq \overline{x}_c\}.$$

An automatic mapping is proposed for c if the suggestion is unique, that is, if $|\textit{HugMe}(c)| = 1$.

IV. Evaluation

This section presents our evaluation of the different *attract* functions introduced in the previous section by comparing them against an oracle mapping for different software systems. We chose software systems with an available oracle that have been used by other researchers before [6]. The oracle mapping M_O for each system serves as the reference for evaluation, but also generate an initial mapping required to calculate the dependency-based *attract* functions. In the evaluation, we will frame the mapping process as a classification problem.

Our evaluation will be somewhat more restrictive than those in other studies as we will accept an automated suggestion only if $|\textit{HugMe}(c)| = 1$, that is, if the suggestion is unambiguous. Other studies have often accepted the mapping suggestion with the highest attraction if there are more. In addition, we will neither use different edge weights ($\textit{weight}(t) = 1$ for every edge type t) nor apply the filtering originally proposed by Christl et al. [3]. A filtering threshold lets the user control which orphans are considered for the mapping at all. If orphans share many dependencies with other orphans and only a few with mapped ones, chances are high that the calculated attraction values are not representative. Note that dependencies between two orphans are not taken into account by *HugMe*. The aim of the original filter is to limit uncertainty due to too many orphaned neighbors. Our focus is on the influence of dependencies and words themselves.

Our implementation is publicly available on GitHub[1] and our research data at Zenodo[2].

A. Procedure

Four groups of attract functions were evaluated: *CA_0, CA_1, ADC, and NB*. *CA_0* is *Count-Attract* incorporating desirable coupling ($\phi = 0$), whereas *CA_1* is *Count-Attract* disregarding it ($\phi = 1$). *ADC* refers to *ADC-Attract* balancing coupling and cohesion based on word occurrences, while *NB* denotes *NB-Attract* solely relying on word distribution.

The dependency-based *attract* functions depend on an initial mapping to be able to come up with meaningful suggestions. To investigate the influence of the size of the initial mapping, we will use different proportions, θ, of the oracle M_O.

An experiment is conducted for each *attract* variant as follows:

1) Generate a random initial mapping of size θ based on M_O.
2) Calculate the attraction matrix using the *attract* function under investigation.
3) Identify implementation nodes whose *HugMeSet* has exactly one element.
4) If there is no such node, proceed to step 6.
5) Otherwise map the node to the identified architecture node and return to step 2.
6) Save results based on current M.

The initial mapping is constructed iteratively in step 1 while the fraction of mapped nodes remains below the threshold θ and unmapped nodes are still available. At each step, an implementation node is randomly selected from the set of yet unmapped nodes and mapped to its corresponding architecture component according to the oracle M_O.

Multiple stages of the reflexion modeling process are simulated by initializing mappings with different values for θ.

B. Measures

Our evaluation views the mapping recommendation as a information-retrieval problem by comparing the suggested mappings, M, against an oracle mapping, M_O. Thus, the

[1]https://github.com/uni-bremen-agst/SEE

[2]https://doi.org/10.5281/zenodo.14726965

measures of success are based on true positives (TP), false positives (FP), and false negatives (FN):

$$\begin{aligned} TP &= |\{(c,a) \in M \mid (c,a) \in M_O\}|, \\ FP &= |\{(c,a) \in M \mid (c,a) \notin M_O\}|, \\ FN &= |\{c \in N_I \mid \forall a \in A : (c,a) \notin M\}| \end{aligned}$$

where we exclude all nodes $c \in N_I$ already present in the initial mapping.

Because we frame the *HugMe* method as a classification problem, we use precision, recall and F-score as quality metrics. Precision and recall are calculated as follows:

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

The F-Score combines precision and recall to provide a single measure of performance:

$$F\text{-}Score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

C. System Preparation

The following Java systems are used for comparing the attract functions, chosen for their comparability to related works and the availability of complete oracle mappings:

- *SweetHome3D (SHD)* is an open-source 3D design tool for interior design.
- *CommonsImaging (CI)* is a library for image file editing and analysis.
- *JabRef (JR)* is an open-source tool for managing BibTeX databases and references.
- *TeamMates (TM)* is a web-based tool for managing and evaluating team projects in education.
- *Lucene (LC)* is an open-source library for full-text search and indexing.
- *Ant* is a build management tool for automating build processes.

For all systems, a dependency graph was created using the *Java2Rfg* tool from the Axivion Suite[3]. Dependencies on system libraries such as `Java.util` were removed to avoid high coupling. Only nodes of type *class* are considered as candidates for automatic mappings.

The oracle mapping for each system was derived from the *s4rdm3x* project[4]. It has to be noted that the architecture components of the oracles are not hierarchical. Moreover, we ignored all implementation components gathered by *Java2Rfg* that did not have a mapping in the oracle.

The systems are listed in Table I, showing architecture nodes (A) and implementation nodes (C) for the mappings. Also shown are the numbers of explicitly allowed (*ea*), divergent (*d*), and implicitly allowed edges (*ia*) for a complete oracle-based mapping. Be reminded that implicitly allowed edges are dependencies within an architecture component, whereas allowed and divergent dependencies are those between two different architecture components. Thus, the proportion of explicitly allowed and divergent edges over all dependency edges, $cr = (|ea| + |d|)/(|ea| + |d| + |ia|)$, represents a degree of coupling (cf. Table I).

[3]https://www.axivion.com

[4]https://github.com/h0bb3/s4rdm3x

System	Version	$\|A\|$	$\|C\|$	$\|ea\|$	$\|ia\|$	$\|d\|$	cr
SweetHome3D	r002382	9	366	2803	5541	0	0.34
CommonsImaging	v1.0a2	21	371	278	3659	272	0.13
TeamMates	v5.11	14	527	5215	4125	188	0.57
Lucene	r1075001	7	786	2259	9531	281	0.21
Ant	r584500	15	792	5884	6436	181	0.49
JabRef	v3.7	6	1226	4430	11995	53	0.27

TABLE I
DATA OF SELECTED SYSTEMS

D. Term Preparation

Terms for each implementation node were prepared using the following steps:

- Source code is tokenized using the ANTLR Java lexer.
- Keywords are removed.
- Identifiers are split by whitespace, camelCase, snake_case, and kebab-case conventions.
- Words are stemmed using the Porter-Stemming algorithim implemented within the Accord.NET library.
- Terms with three or fewer characters are filtered out.
- Namespace components are added to node terms.

V. Results

The results are shown in Figure 3. There is one diagram for each pair of system and attract function investigated.

The procedure outlined in Section IV-A depends on the fraction, θ, of the oracle mapping M_O as a prerequisite for calculating the attract functions meaningfully, where the concrete subset of M_O is chosen randomly. To investigate the influence of θ, we ran this procedure for different values of θ, specifically for of 0.3, 0.5, 0.7, and 0.9 (shown on the x axis of the diagrams in Figure 3). To control a potential bias of randomization in the selection of the subset, the procedure was repeated $n = 100$ times with different random samples for the same value of θ. Thus, instead of reporting only one recall and one precision value for a particular θ, we report the average of the whole range of 100 runs. A dot in each diagram represents the average, while the colored area indicates the standard deviation.

In case of the F-Score, we also report the minimal and maximal value beyond the average only. In addition to recall, precision, and F-Score, the charts show also the mapping rate, which represents the ratio of candidates mapped during the mapping process in relation to the size of the complete oracle mapping; more precisely, it is defined as $(TP + FP)/|M_O|$. This number gives an idea on how many suggestions were made relative to the oracle. Necessarily, the higher θ, the lower the mapping rate since all elements of the initial mapping are excluded from the suggestions. Yet, in addition to these, orphans may also be excluded because *HugMe* did not find an unambiguous mapping (did not find any mapping or more than one mapping candidate).

A. Observations

a) CA_1 (ignoring desired coupling): Precision of CA_1 is consistently the lowest across three systems (TM, SH3D, JR) for all θ values compared to other functions. Also for the system Ant, it is relatively low compared to CA_0 and ADC. Its recall values are leading on four systems (TM, LC, CI, JR), but the function did not achieve equally high precision for those. In most cases, it is the least precise function on the chosen systems. An exception to this trend is observed for the CI system, where the function performed best among all groups. A performance comparable to *ADC-Attract* and *NB-Attract* is noted in terms of F-score for two systems (JR, LC), particularly for higher θ values. However, the function could not achieve equal precision for smaller θ values.

b) CA_0 (considering desired coupling): The precision of CA_0 is very high even for smaller θ values for five out of six systems. System CI is the exception. Its recall values are lower or at least equal compared to other groups for small θ values, resulting in lower F-scores. Recall is never exceeding 0.6. This function performed best for Ant. For SH3D and Lucene, where the highest F-scores were observed for this group, *ADC-Attract* and *NB-Attract* achieve equal or higher F-scores for the same θ values due to better recall within the other groups. For CI, no mappings were found for θ values greater than 0.5.

c) ADC: The observed maximum and average F-score values for *ADC-Attract* are near or equal to those of *NB-Attract*. *ADC-Attract* outperforms *NB-Attract* in precision for $\theta \geq 0.5$ for SH3D and performs weakest when it cannot find any mappings, although it does still tend to achieve a relative high precision then. It performs worse than CA_0 and CA_1 for Ant with respect to very low recall values.

d) NB: NB-Attract achieves the best F-scores for four systems (TM, SH3D, LC, JR), with values near 0.8 across all θ values, except for TM. For Ant and CI, no automatic mappings are found regardless of θ. When mappings are found, the F-Score values are consistently higher or near to those of other functions for all θ values. *NB-Attract* shows minimal dependency on θ for SH3D, LC, and JR and demonstrates robustness to high standard deviations, which appear system-dependent rather than function-dependent. It is also more resilient to these variations for TM.

B. Discussion

The performance of each attract function appears to be primarily system-dependent. Whether the usage of structure or words is more suitable for identifying high-quality mappings also seems to be system-dependent rather than adhering to a general rule. Nevertheless, functions operating on words tend to perform better in many cases.

The results demonstrate that the principle of ADC can be successfully employed to formulate an attract function that performs comparably to *NB-Attract* on four systems. *ADC-Attract* may serve as a solid compromise when it is unclear which source of information can be trusted, words or dependencies. The comparison of the performance of *NB-Attract* and *ADC-Attract* on the systems *Ant* and *CI* supports this conjecture.

Method	θ	ρ Precision vs. *cr*	ρ Recall vs. *cr*	ρ F-Score vs *cr*
ADC	0.3	-0.73 (0.06)	-0.33 (0.47)	-0.60 (0.14)
CA_1	0.3	-0.73 (0.06)	-0.47 (0.27)	-0.87 (0.02)
CA_0	0.3	-0.07 (1.00)	-0.20 (0.72)	-0.20 (0.72)
NB	0.3	-0.28 (0.44)	-0.14 (0.70)	-0.28 (0.44)
ADC	0.5	-0.73 (0.06)	-0.33 (0.47)	-0.33 (0.47)
CA_1	0.5	-0.73 (0.06)	-0.60 (0.14)	-0.87 (0.02)
CA_0	0.5	0.07 (1.00)	-0.07 (1.00)	-0.07 (1.00)
NB	0.5	-0.14 (0.70)	-0.14 (0.70)	-0.28 (0.44)
ADC	0.7	-0.47 (0.27)	-0.20 (0.72)	-0.20 (0.72)
CA_1	0.7	-0.73 (0.06)	-0.60 (0.14)	-0.73 (0.06)
CA_0	0.7	0.41 (0.25)	-0.07 (1.00)	-0.07 (1.00)
NB	0.7	0.00 (1.00)	-0.14 (0.70)	-0.28 (0.44)
ADC	0.9	0.47 (0.27)	-0.20 (0.72)	-0.20 (0.72)
CA_1	0.9	-0.60 (0.14)	-0.60 (0.14)	-0.73 (0.06)
CA_0	0.9	0.28 (0.44)	-0.07 (1.00)	-0.07 (1.00)
NB	0.9	0.14 (0.70)	-0.14 (0.70)	-0.28 (0.44)

TABLE II
KENDALL-TAU CORRELATION BETWEEN PRECISION, RECALL, F-SCORE AND RATIO OF COUPLING EDGES *cr* FOR EACH GROUP AND THETA VALUE (P-VALUES IN PARENTHESES).

Regarding the performance in preliminary experiments [17], the successful application of the overlap coefficient as the distance function for *ADC-Attract* indicates that the presence of words in modules is more critical for identifying high-quality mappings than their frequency. *ADC-Attract* shares this property with *NB-Attract*, suggesting an underlying principle.

The consideration of desired coupling within the reflexion model leads to very high precision, as seen in the results for CA_0. However, this can also result in reduced recall values. Although *ADC-Attract* also considers desired coupling calculated by word usage, it achieves much higher recall values while maintaining solid precision. Ignoring desired coupling seems to increase recall for group CA_1 across four systems (*TM*, *LC*, *JR*, *CI*), consequently enhancing the F-score.

The correlation data presented in Table II shows the relationship between the measured performance and the coupling rate *cr* listed in Table I. Coupling negatively correlates with *ADC* at low theta values but this relation weakens or turns positive as theta increases, suggesting that coupling may hinder performance with lower mapping rates (θ) but might enhance performance for larger initial mappings. Method CA_1 shows a negative correlation with coupling, which is expected, as this method does not take coupling edges into account. When *Count-Attract* incorporates architectural rules, its performance becomes more stable against high coupling. *NB* shows low correlation with coupling, indicating coupling does not significantly impact term distribution. Due to the limited selection of systems and high p-values, these values must be interpreted with caution; but they provide at least some context for the results.

C. Threats to Validity

A significant limitation is the restricted system selection. The choice of only six systems may not be representative and constrains the generalizability of the findings. Additionally, the details of the dependency graphs influence the results. The tool *java2rfg* plays a crucial role in determining the actual graphs. Moreover, many of the systems analyzed are programmed in Java. Language-specific conventions, such as file naming, class naming, and namespace conventions, as well as the selection of the AntLR lexer may impact both *NB-Attract* and *ADC-Attract*.

The influence of desired coupling imposed by the reflexion model depends highly on the architectural view and its level of abstraction. We cannot make definitive statements about how the complexity or abstraction level of an architectural description would affect the performance of the attract functions. Another aspect affecting the validity of the results is the exclusion of edge weights. Incorporating this factor could further enhance the structure-based *attract* functions and potentially lead to different results. Furthermore, the correctness of the oracle mappings plays a significant role in the quality of the experiments. As most edges appear to be allowed based on the system data shown in Table I, we can assume that the oracle mappings reflect the architecture well. The generation of the initial mapping was balanced in the sense that each architectural layer was roughly assigned an equal number of implementation nodes. However, this balance may not transfer to real-world scenarios where correct mappings are unknown, influencing the *attract* functions designed to minimize coupling. The performance comparison of the *attract* functions is strongly influenced by our implementation of the *HugMe* method. It is possible that alternative selection procedures for automatic assignments could lead to different outcomes. Human correction or intervention in the semi-automatic mapping process cannot be fully simulated through automated experiments. This limitation highlights the need for further investigation into the interplay between human expertise and automated tools in mapping experiments.

VI. Conclusions

We introduced a new variant of an *attract* function, named Abstract Dependency Concretization (ADC), that combines dependencies and the presence of words to provide mapping suggestions for reflexion modeling and compared it against two existing methods, *Count-Attract* and *NB-Attract*. In our evaluation, ADC has demonstrated the ability to effectively combine dependencies and words, offering a versatile approach to formulating attract functions comparable to the Bayesian approach. The suitability of dependencies or words for mappings appears to be system-dependent, yet in many cases the text-based *attract* functions perform better. While the usage of desired coupling leads to high precision values for *Count-Attract*, it is accompanied by a disproportionate decrease in recall. On the other hand, ignoring the desired coupling can lead to poor precision on small mapping sizes. Further research is necessary to explore the integration of additional information retrieval techniques and text classification approaches with ADC. For example, extending *ADC-Attract* with Latent Semantic Indexing (LSI) could potentially generalize the matching process and improve recall. Similarly, combining Naïve Bayes text classification with ADC represents a promising area for investigation. The advantage of the possibility to parameterize *ADC-Attract* with edge weights could lead to further improvement, which an approach purely based on words could not provide. Additionally, the evaluation of a broader range of systems is crucial for better understanding the general applicability of the investigated *attract* functions. In a preliminary study, we assessed how automated mappings can be integrated in an interactive process with architects [9]. More research is needed along this line and it might also be worthwhile for future research to investigate how to leverage feedback by architects on the automated techniques to propose even better mappings.

References

[1] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995, pp. 18–28.

[2] R. Koschke, "Industrial experience on code clean-up using architectural conformance checking," in *Proceedings of the 12th European Conference on Software Architecture (ECSA): Companion Proceedings*, 2018.

[3] A. Christl, R. Koschke, and M.-A. Storey, "Automated clustering to support the reflexion method," *Information and Software Technology*, vol. 49, no. 3, pp. 255–274, 2007, 12th Working Conference on Reverse Engineering.

[4] M. Bibi, O. Maqbool, and J. Kanwal, "Supervised learning for orphan adoption problem in software architecture recovery," *Malaysian Journal of Computer Science*, vol. 29, pp. 287–313, 12 2016.

[5] R. A. Bittencourt, G. J. d. Santos, D. D. S. Guerrero, and G. C. Murphy, "Improving automated mapping in reflexion models using information retrieval techniques," in *2010 17th Working Conference on Reverse Engineering*, 2010, pp. 163–172.

[6] T. Olsson, M. Ericsson, and A. Wingkvist, "To automatically map source code entities to architectural modules with naïve Bayes," *Journal of Systems and Software*, vol. 183, p. 111095, 2022.

[7] Z. T. Sinkala and S. Herold, "Inmap: Automated interactive code-to-architecture mapping recommendations," in *2021 IEEE 18th International Conference on Software Architecture (ICSA)*, 2021, pp. 173–183.

[8] D. Link, P. Behnamghader, R. Moazeni, and B. Boehm, "Recover and relax: Concern-oriented software architecture recovery for systems development and maintenance," in *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, 2019, pp. 64–73.

[9] L. Erhardt and R. Koschke, "A controlled experiment on the usability of automated reflexion mapping suggestions integrated in code cities," in *Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon)*, ser. ICSA 2025 Companion Proceedings, 2025.

[10] G. Murphy and D. Notkin, "Reengineering with reflexion models: a case study," *Computer*, vol. 30, no. 8, pp. 29–36, Aug 1997.

[11] V. Tzerpos and R. Holt, "The orphan adoption problem in architecture maintenance," in *Proceedings of the Fourth Working Conference on Reverse Engineering*, 1997, pp. 76–82.

[12] B. Mitchell, "A heuristic approach to solving the software clustering problem," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, 2003, pp. 285–288.

[13] S. M. Naim, K. Damevski, and M. S. Hossain, "Reconstructing and evolving software architectures using a coordinated clustering framework," *Automated Software Engineering*, vol. 24, pp. 543–572, 2017.

[14] A. Florean, L. Jalal, Z. T. Sinkala, and S. Herold, "A comparison of machine learning-based text classifiers for mapping source code to architectural modules," in *European Conference on Software Architecture*, 2021.

[15] R. Koschke and D. Simon, "Hierarchical reflexion models," in *IEEE Working Conference on Reverse Engineering*, Nov. 2003, pp. 36–45.

[16] H. Zhang, "The optimality of naïve Bayes," in *International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2004.

[17] L. Erhardt, "Ein Vergleich von Attract-Funktionen für die Reflexionsanalyse hinsichtlich Struktur, Code-Termen und interaktive Nutzung," Master's Thesis, Department for Mathematics and Informatics, University of Bremen, Germany, Aug. 2024.

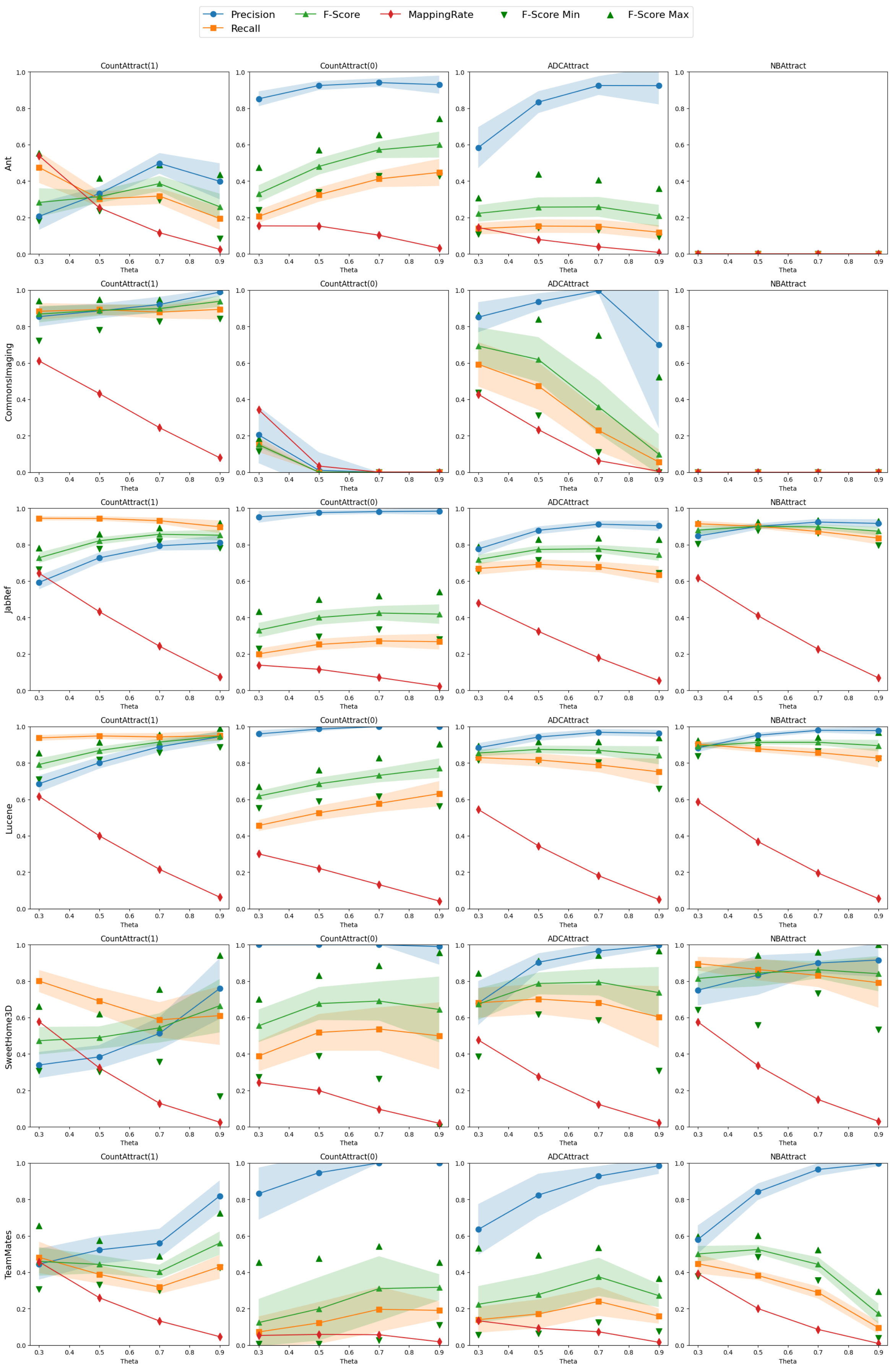

Fig. 3. Results of experiments