Mining Cross-Task Artifact Dependencies from Developer Interactions

Usman Ashraf, Christoph Mayr-Dorn, and Alexander Egyed
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
{usman.ashraf,christoph.mayr-dorn,alexander.egyed}@jku.at

***Abstract*—Implementing a change is a challenging task in complex, safety-critical, or long-living software systems. Developers need to identify which artifacts are affected to correctly and completely implement a change. Changes often require editing artifacts across the software system to the extent that several developers need to be involved. Crucially, a developer needs to know which artifacts under someone else's control have impact on her work task and, in turn, how her changes cascade to other artifacts, again, under someone else's control. These cross-task dependencies are especially important as they are a common cause of incomplete and incorrect change propagation and require explicit coordination. Along these lines the core research question in this paper is: how can we automatically detect cross-task dependencies and use them to assist the developer? We introduce an approach for mining such dependencies from past developer interactions with engineering artifacts as the basis for live recommending artifacts during change implementation. We show that our approach lists 67% of the correctly recommended artifacts within the top-10 results with real interaction data and tasks from the Mylyn project. The results demonstrate we are able to successfully find not only cross-task dependencies but also provide them to developers in a useful manner.**

***Index Terms*—cross-task dependencies, change impact assessment, developer interactions, software artifacts recommendation, Mylyn, Bugzilla.**

I. Introduction

Implementing a change such as fixing a bug, introducing a new feature, or removing outdated functionality is a challenging task. Correctly and completely implement a change requires the developers to identify all relevant artifacts. In non-trivial systems, a change often requires editing artifacts that are maintained by different developers or different teams. Aside from small, localized bug-fixes changes are rarely described and managed by a single task but rather a set of tasks worked on by different developers.

Crucially, a developer needs to know which artifacts under someone else's control have impact on her underlying work task: changes to such artifacts then may induce additional changes, might restrict how to implement a change, and when to do so. In turn, a developer's artifact changes cascade to other artifacts under someone else's control. We characterize the situation when changes to artifacts in one task influence (potential) changes to other artifacts in another task as a cross-task dependency. Simple, illustrative examples include fixing a bug in business logic and updating integration tests accordingly, introducing a new field in the database and displaying it on the user interface, or introducing a feature toggle and adding the toggle trigger to the configuration database. From the authors' experience, in many organizations often different teams are responsible for these tasks due to prescribed development processes, required expertise, or organizational structure.

Awareness of cross-task dependencies is especially important as lack thereof is a common cause of incomplete and incorrect change propagation. Developers thus need support in becoming aware of these dependencies, respectively the involved artifacts for simpler coordination of change propagation (i.e., forward and backward impact assessment [6]). Manually maintaining these dependencies is tedious.

Along these lines the core research question in this paper is: how can we automatically detect cross-task dependencies and use them to assist the developer?

We introduce an approach for mining cross-task dependencies from developer interactions with engineering artifacts as captured in the IDE. Interactions describe which artifacts a developer has accessed and edited within the scope of a task. From task pairs, we extract re-occurring artifact sequences: the cross-task dependencies. We further observe the accessed artifacts during live editing in the IDE and apply the dependencies to recommend which other artifacts are potentially affected by the ongoing work.

This paper provides the following two contributions: First, we developed a technique for extracting cross task dependencies from interaction data. To the best of our knowledge, this is the first attempt to obtain artifact dependencies across tasks. Contemporary approaches derive dependencies from within-task data or data from aggregated tasks of very close temporal proximity. Second, we provide a recommender prototype that applies developer interactions in order to suggest a ranked list of affected artifacts.

We obtained real developer interaction data and tasks from the Mylyn project and mined cross-task artifact dependencies in a sliding window over the duration of several years. Our approach lists 50% of correctly recommended artifacts within the top-5 results. The results demonstrate we are able to successfully find not only cross-task dependencies but also provide them to developers in a useful manner.

This work was supported in part by the Austrian Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria; and the FFG, Contract No. 854184.

The paper is structured as follows. Section II provides a detailed motivating scenario to further outline the importance of this research followed by section III that outlines the technical details of our approach. Section IV then lists the specific research questions this paper investigates. Section V reports the study design with results available in Section VI and Section VII. We discuss the findings in Section VIII. Section IX compares our approach to state of the art before Section X completes the paper with conclusions and an outlook on future work.

II. Motivating Scenario

We motivate our research based on a real example from the open source Mylyn project. Mylyn [15] allows a developer to connect to a task management tool (such as Bugzilla) for selecting tasks to work on and captures all developer read and write events within the Eclipse IDE. DevMM is assigned to new task *152211*[1] *- Allow local tasks to be promoted to repository tasks*, a feature in Mylyn's *Bugzilla* sub component. DevMM obtains further details such as *Provide facility to copy task from local repository to remote one* from the task description. Furthermore, he also examines comments from other developers on the same task which suggests their familiarity with the task. From the interaction history attached to the task we learn that DevMM browses through different artifacts; a usual activity to locate relevant source code artifacts.

Ideally, an impact analysis tool supports DevMM during this activity by suggesting potentially affected artifacts based on, for example, his browsing activities. In a developer's experience, implementing a change often has impact on artifacts maintained by other developers. A developer will then assess to what extent his intended (or already implemented) changes will affect someone else's code (forward impact assessment). That typically implies inspecting the potentially affected artifacts, and hence requires being aware of them in the first place. Similar, other developers' artifacts (and their recent changes) may influence how a change can be implemented (backward impact assessment). Again, this involves being aware of such artifacts and inspecting them for relevance. Support is particularity important when these artifacts reside *far away* from the developers current work context, e.g., in a different (java) project or subcomponent.

In this particular example, our approach could support DevMM by recommending to inspect *...trac.core/.../internal/trac/core/TracTaskDataHandler.java*.[2] While task 152211 belongs to the `Bugzilla` subcomponent, this artifact belongs to the `Trac` subcomponent; a subcomponent primarily maintained by DevSP.[3] DevMM would investigate, for example, whether the change for the Bugzilla subcomponent can be replicated in the Trac subcomponent.

Our approach is able to provide such a recommendation as it analyses past developer interactions and extracts which artifact dependencies exist across task boundaries. In this case, as soon as DevMM accesses *...tasks.ui/.../tasks/ui/editors/AbstractRepositoryTaskEditor.java* in the IDE we observe this event and search for matching cross-task dependencies. In the months prior to DevMMs ongoing development efforts on task 152211, the Mylyn project exhibited six task pairs where *...trac.core/.../internal/trac/core/TracTaskDataHandler.java* was accessed in one task and *...tasks.ui/.../tasks/ui/editors/AbstractRepositoryTaskEditor.java* in another one. Listing 1 provides the corresponding cross-task artifact dependency excerpt as represented in JSON.

```
{
  "id": "7ec6f597a31b6623e0207eebba23f199",
  "miningStart" : "2007-07-06",
  "miningEnd" : "2008-01-02",
  "sourceArtifacts": [
    "org.eclipse.mylyn.tasks.ui/src/org/eclipse/
        mylyn/tasks/ui/editors/
        AbstractRepositoryTaskEditor.java"
  ],
  "destinationArtifacts": [
    "org.eclipse.mylyn.trac.core/src/org/eclipse/
        mylyn/internal/trac/core/TracTaskDataHandler
        .java"
  ],
  "TaskPairs": [
    [ "Id-196643", "Id-196622" ],
    [ "Id-196643", "Id-196585" ],
    [ "Id-196700", "Id-196622" ],
    [ "Id-196700", "Id-196585" ],
    [ "Id-196700", "Id-196643" ],
    [ "Id-196643", "Id-196700" ]
  ]
}
```

Listing 1: JSON representation of a cross-task artifact dependency

III. Mining and Applying Cross-Task Artifact Dependencies

Our approach to mining and applying cross-task artifact dependencies consists of three phases as depicted in Figure 1: (i) software development data gathering (1), (ii) dependency mining (2,3), and (iii) artifact recommendation (4,5,6,7). Artifact recommendation is one of many potential uses of cross-task artifact dependencies selected for demonstration purposes.

A. Software Development Observation

The basic data set from which we extract cross-task dependencies are developer interactions with engineering artifacts such as requirements, models, source code, documentation, test reports etc. (see Figure 1 (1)). In this paper, we focus on source code artifacts.[4] A developer interaction describes how the developer interacted with the IDE such as opening, editing, closing artifacts, navigating the artifact structure, or executing commands. Various preexisting approaches such as Mylyn

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=152211

[2] We abbreviated all artifact names in this section for sake of readability.

[3] Bugzilla and Trac are two different bug tracking tools.

[4] As the developer interactions from the evaluation system under study contain almost exclusively references to source code we cannot demonstrate that the approach would be successful in discovering cross-task dependencies among other artifact types.

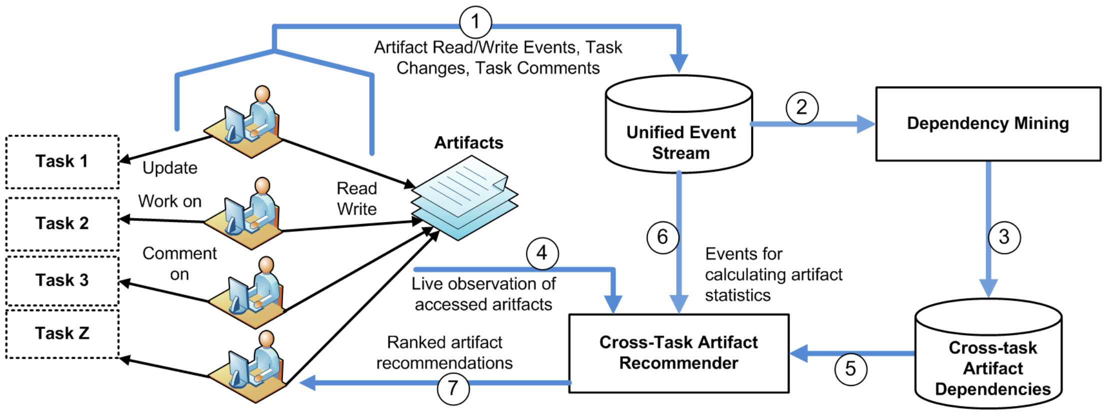
1
Artifact Read/Write Events, Task Changes, Task Comments
Task 1
Task 2
Task 3
Task Z
Update
Work on
Comment on
Artifacts
Read
Write
Unified Event Stream
2
Dependency Mining
3
6
Events for calculating artifact statistics
4
Live observation of accessed aritfacts
Ranked artifact recommendations
7
Cross-Task Artifact Recommender
5
Cross-task Artifact Dependencies

Fig. 1: Approach.

capture such interactions. The key requirements for interaction data to be useful for our approach is providing following properties per interaction: who conducted the interaction (i.e., which developer), when did the interaction occur (we don't need duration), what artifact (fragment) was accessed (read or write), and in the scope of which task (what bug report, issue, or story) did the interaction happen. Additionally, we require task details on assigned developers, task description, etc. As with interaction events, we store this information as task update events in the Unified Event Stream. We explain their use for measuring similarity among tasks in the following subsection.

We store all interaction events in the Unified Event Stream, thereby abstracting away which tool provided the interaction data. Ultimately, the Unified Event Stream contains the interaction events of all engineers working on the various task over the complete life-cycle of a software system.

B. *Cross-task Artifact Dependency Mining*

For dependency extraction, we decide on a time window (e.g., 6 months) and select tasks having being worked on actively during that interval (Figure 1 (2)). The window size depends on work intensity and the typical task durations. An overly wide window increases the risks of detecting primarily noise, i.e., false positive dependencies, while a too narrow window risks missing dependencies as it fails to include related tasks. We define a task as being actively worked on when a minimum number of interaction events have occurred (e.g., 10). Again, such a threshold depends on the typical number of artifacts accessed within the scope of a task. A high threshold implies that only tasks with a lot of events respectively involved artifacts will be considered for mining (thus potentially missing dependencies), while a too low threshold selects also task with little progress that are likely not part of a dependency (yet) but increase the mining effort.

In short, the preliminary input to the extraction algorithm consists of the active tasks, and for each task the set of artifacts that were accessed (read or write) within the time window according to the interaction events. Interaction events for an active task that occurred outside that time window are not considered.

The next challenge is generating meaningful task pairs from all active tasks in the observed time-window. Taking all possible permutations of tasks (i.e., the cartesian product) is not an option for two reasons: first, the number of pairs grows exponentially with the number of tasks and would require extensive computing resources for running the mining algorithm. Second, with all permutations the likelihood for finding noise increases. Our approach aims to identify task pairs that are more likely to exhibit cross-task dependencies.

Figure 2 depicts our task pair generation approach. Assume that out of tasks A to Z, querying the Unified Event Steam for active tasks in a given time-window t returns the tasks A, C, F, and X (1). We then calculate pair-wise similarity among tasks using properties such as common commenting developers or textual similarity in task description and comments (2). We use cosine similarity on the two sets of developers having commented on the respective tasks, which yields a similarity value of 0 for completely different commenting developers, and 1 for two tasks with the same set of commenting developers. With respect to textual similarity, we extract terms, apply stemming, and apply cosine similarity on the resulting term vectors. Overall similarity is simply a linear combination of these two metrics, giving each equal weight.

The rationale behind using developer and textual similarity as an indicator for cross-task dependencies is that the tasks are likely to share a common vocabulary (e.g., a database field update with impact on the user interface, very likely has the respective user interface artifacts use similar terms) and that relevant developers—even when not explicitly assigned to the task—share their expertise in comments. We deliberately ignore temporal distance among tasks as a similarity metric as, on the one hand, there are often large temporal gaps between related tasks, and on the other hand, developers try to work on unrelated tasks concurrently or in close temporal proximity as these require no or very little coordination.

From the resulting similarity matrix, we chose for each task the top-k most similar tasks (3) and collect them in a list. From each list, we generate the set of pairs (4). We then filter out

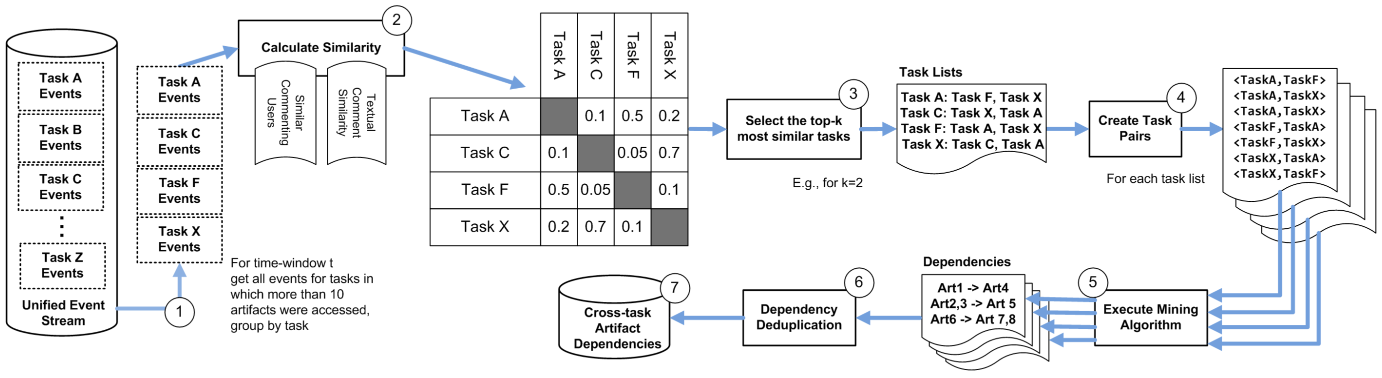

Fig. 2: Generation of TaskPairs.

all artifacts that occur insufficiently often (i.e., $< k * 2 + 1$) to become part of a pattern. Filtering artifacts may lead to an empty set, we then remove the respective sequence from the set. We abort mining once the number of sequences falls below the support threshold s. We introduce s to obtain some initial confidence in a cross-task dependency. We store only those dependencies which the mining algorithm detects among at least s distinct task pairs. The lower s the more patterns we are likely to find, but also the more likely these represent noise. The higher s the fewer patterns we can expect to find.

The actual problem of mining cross-task dependencies can be mapped to a simplified sequential pattern mining problem for which several algorithms exist [9]. Sequential pattern mining (SPM) takes a set of sequences, each sequence in turn consisting of item sets. SPM then searches for reoccurring patterns where one or more items in one item set are followed by items in a subsequent item set (within the same sequence). We use an algorithm by Fournier-Viger et al. [10] from the SPMF library.[5]

In our heuristic a sequence consists of a source tasks and a destination task (i.e., a task pair). The artifacts accessed within these tasks make up the respective item sets. We are interested which changes in one task (i.e., the source task) tend to be followed by (other) artifact writes and reads in another task (i.e., the destination task). Specifically, we include only write events in the source task, and include write and read events in the destination task. We do this to capture situations where a developer inspects the changed artifact to assess its relevance, respectively, where a change in one task results in artifacts automatically being (re)generated (e.g., model-to-code transformations) that are not changed again but rather read only.

Note that the temporal order of interaction events as well as the temporal order of tasks is irrelevant for dependency extraction as we assume cross-task dependencies to be time-insensitive. For example, it does not matter whether first one task updates a test, and then another task updates the corresponding functionality, or whether it is the other way around, or whether this happens simultaneously, the dependency exists nevertheless. We, thus, place every task pair twice into the set of sequences—once in temporal order, once in inverse temporal order—to detect dependencies independent of time. This has no negative effect on detecting dependencies that happen to be time-sensitive.

In our approach, we run the mining algorithm (5) at most n times (once of each active task), which a maximum of $k * (k - 1)$ sequences per mining run—the mining effort thus grows linearly in n. In contrast, without generating task pairs, we would run the mining algorithm once but with $n * (n - 1)$ sequences. Listing 1 provides an example cross-task artifact dependency output. A dependency consists of the source artifact set, destination artifact set, the set of task pairs among which the mining algorithm detected the pattern, and the time-window. Specifically in Listing 1, editing `AbstractRepositoryTaskEditor.java`, coincided with accessing of the destination artifact `TracTaskDataHandler.java` among six task pairs.

After running the mining algorithm (6), we de-duplicate the found dependencies: for an identical pair of source and destination artifacts we simply merge the two task pair sets. Ultimately, we store the cross-task artifact dependencies for later use (7).

C. Artifact Recommendation

We expect cross-task artifact dependencies to be useful for multiple use cases. One such use case is supporting developers live during development by informing them about artifacts to inspect for impact on their underlying work (backward impact assessment) as well as what artifacts to inspect for impact of their work (forward impact assessment). Our recommendation prototype listens to artifact read and write events during a developer's work in the IDE (i.e., the same type of events as stored in the Unified Event Stream, Figure 1 (4)).

For every few events, the recommender checks if it has used the accessed artifact as a trigger for a recommendation before. If this is not the case, it queries the cross-task artifact dependency database for any dependency that lists the artifact among the source artifacts (5). From all matching dependencies, the recommender then retrieves all artifacts from the destination set, filtering out those that have been recommended

[5]http://www.philippe-fournier-viger.com/spmf

once before. We recommend each artifact at most once to avoid annoying the developer. The recommender then ranks the remaining artifact candidates along the following criteria to rank the most relevant ones first.

We apply the following criteria (6):

Occurrence counts across all matched dependencies in how many tasks the candidate artifact was accessed. We assume a higher count implies more relevance.

Distance measures the package hops from the triggering artifact to the candidate. Similar to [7], we count how many packages up the hierarchy and down again does a developer has to navigate to reach one artifact from the other. For example, the artifact `org/eclipse/tasks/Class1.java` is three hops away from `org/eclipse/internal/sandbox/Class2.java` as we need to traverse `tasks` up and `internal/sandbox` down to reach `Class2.java` from `Class1.java`. We assume a more distant candidate artifact to be more relevant than a closer one as a developer may be less aware of the change impact than on an artifact that is close to his/her current work context.

Access Frequency counts the number of tasks (in the time-window used for mining dependencies) in which any developer accessed the candidate artifact. In contrast to Occurrence, this metric evaluates the popularity of an artifact. We assume, a frequently changed artifact has more impact than an infrequently changed one.

Personal Access Count determines in how many tasks the currently active developer (the one about to receive a recommendation) has accessed the artifact candidate before.

We normalize each score to the range $[0, 1]$ for comparability and multiplying each score by its respective weight, where $w_i \in [0, 1]$ and $\sum_i w_i = 1$. The sum of weighted scores produces the overall score used for ranking all artifact candidates. The recommender returns the ranked list of artifacts (i.e., a recommendation instance, Figure 1(7)) and internally stores for each recommended artifact the ranking metric results and applied ranking weights. This enables dynamically adapting the weights upon observing what artifact the developer eventually accesses. Such analysis and self-tuning, however, is out of scope of this paper.

IV. Research Questions

In our paper, we split the research questions (RQ) into two coarse grained groups: (1) what are the characteristics of detected cross-task dependencies? and (2) are the detected cross-task dependencies indeed useful?

RQ1a: Do we find cross-task dependencies with our proposed heuristic? Answering this question informs us whether we are able to detect reoccurring development situations where accessing an artifact in one task (i.e., the source artifacts) tends to coincide with accesses to an artifact in another task (i.e., the destination artifacts). At this stage, however, we remain unsure whether the found relations are true dependencies or just noise.

RQ1b: How much are the source and destination artifacts overlapping. Put inversely: to what extent are artifacts in the source set disjunct from artifacts in the destination set?
A large overlap would imply that the same artifact are changed together in multiple tasks and hence that contemporary approaches based on logical coupling, for example, would be equally able to find these dependencies.

RQ1c: What is the ratio of unique artifacts in the dependencies compared to all unique artifacts?
A high ratio might indicate that our heuristic finds a lot of noise as we wouldn't expect the majority of artifacts to be part of a cross-task dependency. A high ratio points to poor cohesion and tight coupling of artifacts (and teams!) across the system.

RQ1d: Are the task pairs—among which we find task dependencies—linked in the issue tracking system?
If so, then our heuristic could focus on analysing linked task pairs only. Previous work has shown that linked task pairs not necessarily exhibit access of the same artifacts [20]. It didn't investigate, however, the presence of cross-task dependencies.

RQ2a: Are we able to predict based on cross-task dependencies whether developers need to become aware of a particular artifact given their current work task context?
If so, the task dependencies can be applied in a recommender to inform a developer about artifacts that might have been changed in another task context before, respectively what artifacts might have to be changed by another developer, thereby assisting forward and backward change propagation. If we can't predict artifacts then the possibility exists that our heuristic detects mere noise in the interaction data.

RQ2b: Are we able to predict artifacts in a meaningful manner?
If we are able to predict artifacts but these are buried among a large number of false positives then the developer is burdened with evaluating the correctness of the recommendation and—over time—will likely ignore the recommendations. Thus we achieve little benefit.

V. Study Design

A. Data Gathering

We use developer interactions captured via the Eclipse IDE Mylyn Plugin[6] during the development of the Mylyn project [15]. Mylyn tracks the development context of an interaction which provides us with details on who was working on which bug report, the type of interaction (e.g., selection, write, navigation) and the timestamp of the interaction. We have public access to these interaction events in the form of attachments to bug reports on the Eclipse Bugzilla website.[7] We interpret a bug report as a task in our approach. We retrieved all bug reports including their change history, comments, and interaction attachments from the Mylyn project that listed at least one

[6]http://www.eclipse.org/mylyn/
[7]https://www.bugzilla.eclipse.org/bugs/query.cgi

Mylyn interaction attachment via Bugzilla's JSONRPC API.[8] For each interaction attachment, we extracted all read events (i.e., selection) and write events (i.e., edit) at the file level and stored them in the Unified Event Stream (ignoring all other events as they don't impact this work). We additionally added events for each task change and each comment as needed for task similarity calculation. In total, we processed 4,477 tasks with interaction attachments, resulting in over 417,000 events that cover the interval from 15th November 2005 to 4th April 2017. We observed the majority of interaction events in the interval between 2006 and 2013.

B. Dependency Mining

We applied a sliding window approach for mining dependencies and then evaluating their usefulness.

Within a training window, we selected all tasks with at least 10 artifact access events in the Unified Event Stream as the `training tasks`. This selects only those tasks that have their main work effort fall into the training window. We generated task pairs with $k = 4$ as outlined in Section 1 and executed the mining algorithm with a support threshold of $s = 5$, a trade-off between computation time and likelihood to detect a dependency.

Inspecting the Mylyn data set, we noticed that related tasks (i.e., tasks that likely exhibit cross-task dependencies) often have significant temporal gaps due to the nature of open source development. We therefore determined a 180 day training window to be large enough to include related tasks and small enough to detect dependency changes across time.

C. Dependency Evaluation

We take the time (i.e., evaluation) window immediately following the training window for evaluating the detected dependencies. We chose a 30 day evaluation window size as this selects tasks that are not too far away from the mining window and long enough to select the majority of work going on in the task. Within the evaluation window, we selected all tasks with at least 10 artifact access events in the unified event stream as the `evaluation tasks`. This selects tasks for evaluation only during intervals of their main work effort; not at the task's beginning when there are too few events to use as input for recommendation, nor towards the task's end when recommendations wont be useful.

We simulated developer work on a particular task by replaying all interaction events for that task within the evaluation window in temporal order. For every five events—the first five are the seed—we trigger the recommender with replayed events and obtain its artifact recommendations (see Figure 3 (1)). Hence, we obtain for each task a set of recommendation instances, each containing a list of one or more ranked artifacts (4).

For each recommended artifact, we mark it as a true positive when the interaction event sequence yet-to-be-replayed contains an event accessing that particular artifact.[9] We mark the artifact recommendation as a false positive when no such event occurred. Given that we recommend multiple times, we further record its recommendation-specific rank (i.e., its position within the current set of artifact recommendations), and its task-specific rank (i.e., its position among all recommendations for a task).

At the end of each task replay, we mark all artifacts as false positives which appeared in the Unified Event Stream in any prior training interval and that were not part of the seed (7). We excluded any new artifacts from the false negative set as we cannot predict an artifact we haven't encountered before.

After mining dependencies from the 180 day window, and evaluating them in the subsequent 30 day evaluation window, we would then shift the training and evaluation window by 30 days and repeat the procedure. Given the lack of active tasks (i.e., insufficient developer interactions per task) during the first few months, we set the first training window to 12th May 2006 to 8th November 2006. Similar, we set the end of the last training window to 6th May 2013 for a total of 80 training, respectively evaluation, iterations. Events after March 2013 were either too few to consider a task as active, or consisted merely of task updates and comments.

The following section reports results from the quantitative analysis.

VI. Dependency Mining Quantitative Analysis

This section reports on the quantitative analysis results required for answering research questions RQ1a-d.

Figure 4 displays dependency statistics for 60 training window iterations starting May 2006 to Nov 2011. We select between 2 and 6 tasks for mining until May 2013. These are, however, too few for our mining algorithms to find patterns and hence we don't display iterations 61 to 80. Our Supporting Online Material[10] includes the data set with all iterations, though.

Answering ***RQ1a:*** *Do we find cross-task dependencies with our proposed heuristic?*
Figure 4 (top) reports how many unique artifacts developers accessed in events across all tasks within each iteration ($\bigtriangledown$), respectively how many unique artifacts exist in the code base from the first iteration until the end of the ith iteration ($\bigtriangleup$).

Figure 4 (middle) visualizes the number of training tasks, the amount of unique dependencies found the number of unique artifacts across all source sets, respectively destination sets; again based on events from a single iteration.

Our approach detects the first dependencies in iteration 3, the last ones in iteration 50, with a total of 1,813 dependencies. In iteration 13 we find the most dependencies: 235 dependencies, respectively 372 normalized dependencies, made up of 72 unique artifacts among the source artifacts, respectively 124 unique artifacts among the destination artifacts.

Overall, these results show that our mining heuristic finds a significant number of cross-task artifact dependencies. The

[8]https://www.bugzilla.org/docs/4.4/en/html/api/Bugzilla/WebService/Server/JSONRPC.html

[9]Recall that we don't recommend already accessed artifacts.

[10]https://figshare.com/s/cf9b7c8b6a46f52b3ad0

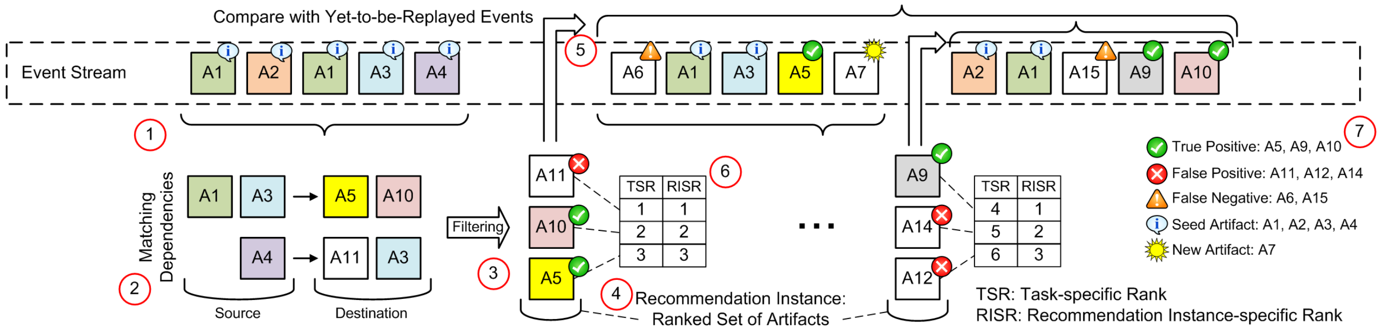

Fig. 3: Recommendation evaluation procedure based on replaying developer interactions, resulting in two recommendation instances.

bulk of these dependencies occur during the initial growth phase of the project. We note a sudden spike in unique artifacts in iteration 17 which sees a sudden increase of almost 5,000 new artifacts. This spike interrupts the gradual slowdown of newly introduced artifacts which we observe before and after that iteration. We assume that this is the result of a major restructuring of the code base and/or integration of sub components previously not covered in the interaction traces. Our approach would require artifact, respectively source code commit, inspection capabilities to detect artifact renaming, respectively, artifact relocation to another package. At the moment, our approach treats the artifact before and after renaming as two distinct artifacts. Consequently, our mining heuristic is less likely to find dependencies involving the renamed artifacts, and existing dependencies are no longer relevant during recommendation and result in false positives. This is one explanation why the number of detected dependencies is low after iteration 17 (see Figure 4 middle) even though the number of unique artifacts accessed per iteration remains high (see Figure 4 top).

Up to iteration 8 and beyond iteration 25, a lack of accessed artifacts explains the low number of detected dependencies.

Answering ***RQ1b:*** *How much are the source and destination artifacts overlapping. [...]*
Figure 4 (bottom, red x) reports the median Jaccard similarity coefficient (i.e., set intersection over union), measuring the overlap of artifacts in the source set and destination set for detected dependencies. A Jaccard coefficient close to 1 describes sets with almost identical members, while a value close to 0 describes two sets that share virtually no members.

The median Jaccard coefficient rarely rises above zero. Only at later intervals where we hardly find dependencies (compare with Figure 4 middle - *Dependencies Detected*) do we experience non-negligible overlap. This implies that for at least half of all dependencies there is no overlap between source and destination artifacts. This is a strong indicator that we are able to find dependencies that cannot be detected with logical coupling techniques. Logical coupling requires the coupled artifacts to repeatedly appear together in the same commit. If cross-task dependencies were just logically coupled artifacts, then we would find similar source and destination sets (i.e., the same artifacts appearing in source and destination) consequently yielding high Jaccard similarity. Figure 4 (bottom, red x) shows that this is not the case.

Answering ***RQ1c:*** *What is the ratio of unique artifacts in the dependencies compared to all unique artifacts?*
Figure 4 (bottom, green □ and blue ◇) depicts the ratio of artifacts in the source set, respectively destination set, with respect to all accessed artifacts in that iteration.

There we observe how only for a single iteration in the beginning of the project the source artifacts consist of $\sim$13% of all accessed artifacts. This number quickly drops to single digits, and eventually fluctuates around 1%. Destination artifacts show an even earlier drop and remain similarly low. These numbers indicate that only a handful of artifacts make up a cross-task artifact dependency, and hence explain why recall is not a suitable metric for evaluating the benefit of our approach.

Answering ***RQ1d:*** *Are the task pairs—among which we find task dependencies—linked in the issue tracking system?*
Explicit links between task pairs of a dependency (e.g., Bugzilla's *blocks* and *depends on* relations) occurred extremely rarely (hence not shown). In total, we found only in 13 out of 1,813 dependencies where at least one task pair exhibited an explicit link. We encountered these dependencies in iterations 10, 11, 12, 13, 17, and 30.

Observing an explicit link among task pairs in less than 1% of all dependencies is a strong signal that explicit links cannot be used as a filter/predictor among which task pairs we are likely to find a cross-task artifact dependency.

VII. Recommendation Quantitative Analysis

Figure 5 (top) displays for each iteration the number of tasks that received no recommendation (□), no correct recommendation (x), and tasks with at least one correct recommendation (●). Evaluation iteration i identifies the 30-day window following the ith training iteration.

Over the 50 iterations, we replayed a total of 2,086 evaluation tasks. We stop at iteration 50, the last iteration where we detected dependencies.

Answering ***RQ2a:*** *Are we able to predict based on cross-task dependencies whether developers need to become aware*

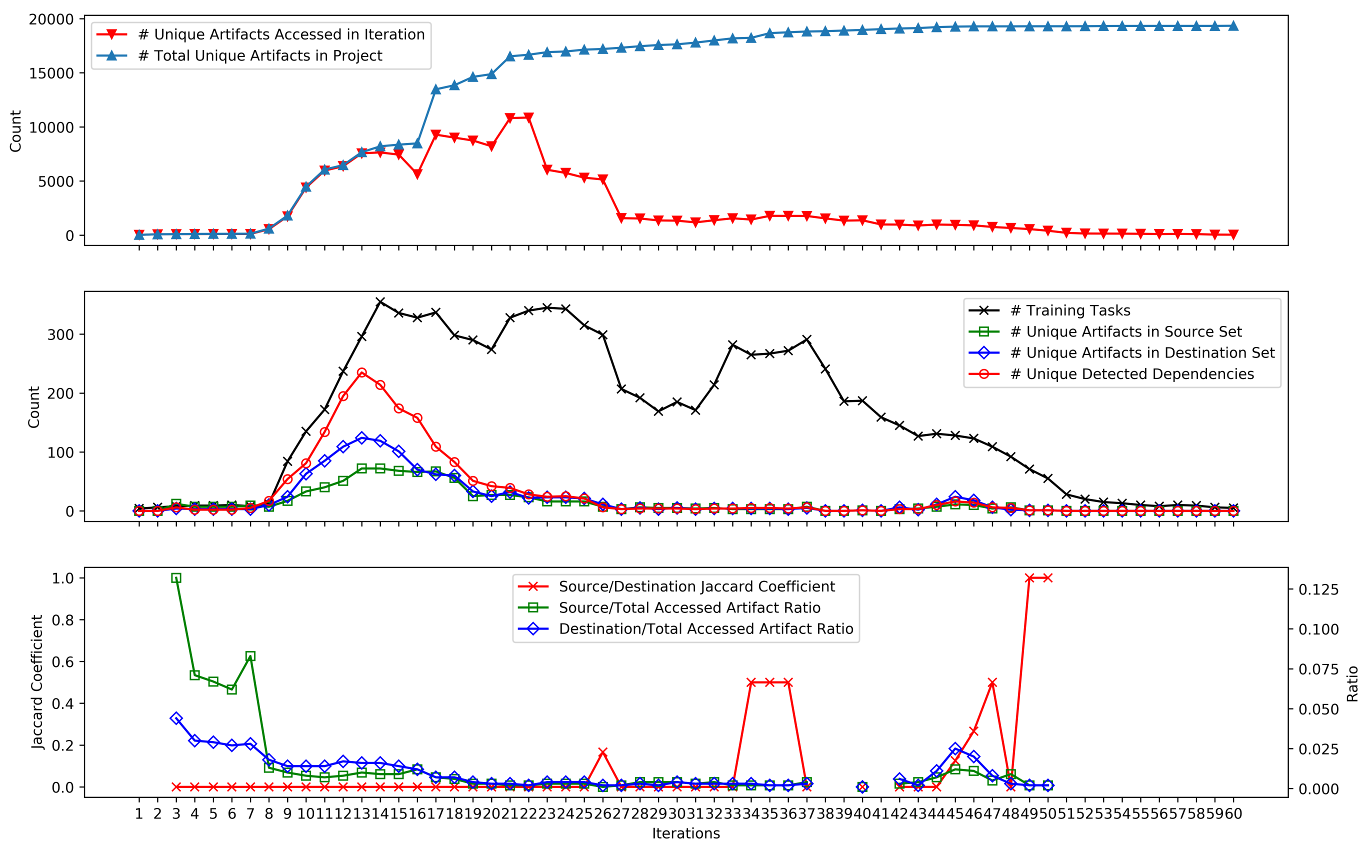

Fig. 4: Dependency Statistics

of a particular artifact given their current work task context? We provide recommendations (i.e., recommend at least one artifact) for 798 out of the 2,086 evaluation tasks, a task coverage rate of 38%. Out of these, 229 contain one or more correct recommendations. On the task level, we thus achieve a precision rate of 29%.

A task coverage rate of 38% and task precision rate of 29% is expectedly low as a well-designed software system enables the majority of tasks to be worked on independently, i.e., without affecting other tasks and hence involving no artifacts which are part of a cross-task dependency. This fact is also apparent in low recall values (see Figure Figure 5 bottom, ○).

Low coverage and recall is not an issue as our goal is NOT to recommend every artifact a developer should access but rather aim to focus recommendations to those that have potential cross-task impact and thus might need dedicated coordination.

Answering ***RQ2b:*** *Are we able to predict artifacts in a meaningful manner?* We compute the mean reciprocal rank (MMR) using task-specific rank (TSR) across tasks for every interval and report in Figure5 (bottom): once for all tasks with a recommendation (+) and once only for tasks with at least one correct recommendation (×). The overall MMR for tasks with at least one correct artifact recommendation is 0.60; and 0.18 across all tasks.

Out of the 229 tasks, we are able to provide in 214 instances at least one recommendation instance that has a correct artifact within the top 10 results (i.e., 93%). Overall, the median recommendation instance-specific rank (RISR) of a correct artifact recommendation (i.e., independent of tasks) is 5, with 67% of artifacts having an RISR <= 10. This implies that in 67% of recommendation instances, whenever there is at least one correctly recommended artifact, the developer will find it within the top 10 results. Hence, a developer needs not look far down the list of artifact recommendations to obtain at least one useful recommendation. This is in range of the typical search distance when browsing search engine results on the web.

VIII. Result Discussion

In this section, we discuss the implications and limitations of our approach and its evaluation.

A. Implications

Overall, the results paint a promising picture that developer interaction events are suitable for detecting cross-task artifact dependencies. We refrain from the claim, however, that such events should be the only input to dependency detection. The quantitative evaluation of recommendations highlighted that additional efforts need to be put into determining when a cross-task dependency is relevant as the precision rates leave room

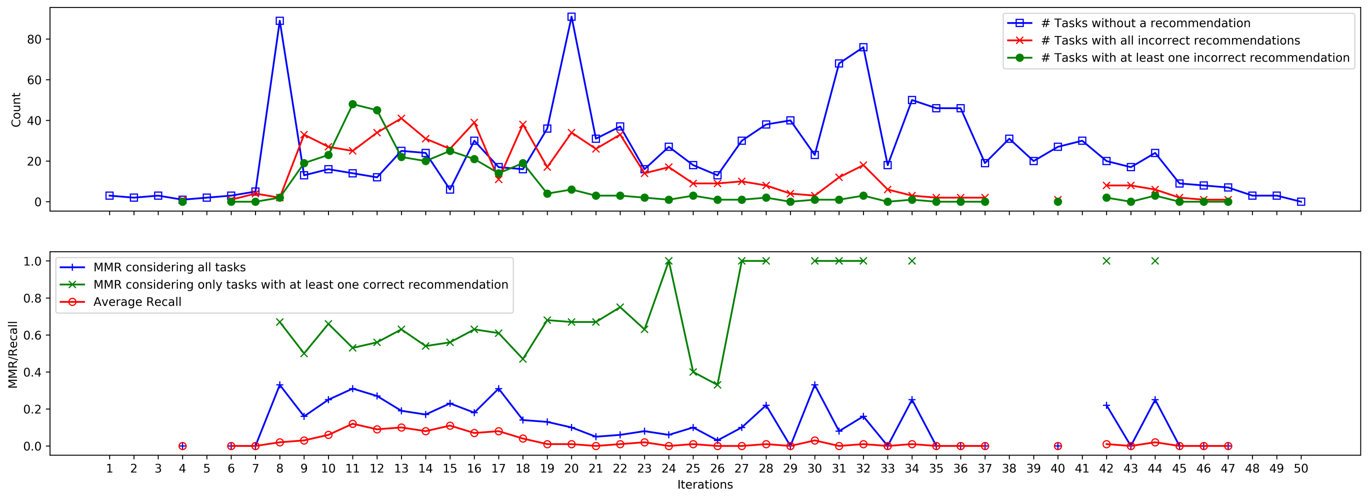

Fig. 5: Tasks with correct/without correct/without any recommendations (top); Recommendation Recall and MRR (bottom).

for improvement. Determining dependencies based on artifact access events below file level might allow for more accurate dependencies and recommendations.

While live recommendations to developers are one way to make use of the detected dependencies, potential alternative uses include providing insights during explicit impact assessment activities, code reviews, task scheduling, or software system architecture inspection. Our results motivate the evaluation of these application scenarios as part of future work.

B. Limitations

We evaluated out approach with artifacts at the file level as we hypothesize that cross-task dependencies among Java classes are likely to exist at the file level rather than at the method level or below. While our approach is generic enough to work on any artifact granularity level, extra effort and evaluation is needed to confirm dependencies detection among, for example, model elements in UML and source code, or among source code and documentation.

Another limitation is support in the presence of newly created artifacts. We can only provide recommendations for artifacts that developers accessed in previous tasks that where subsequently mined and hence might not be aware of the most recent cross task dependencies. In this respect our approach has the same limitations with regard to new artifacts as logical coupling techniques. We argue, however, that we generate recommendations as soon as the developer accesses an existing artifact (e.g., to check how something was implemented so far). Additionally, we don't expect an immediate impact when only new artifacts are introduced.

C. Threats to validity

Internal Validity We address researcher bias by analysing data from an open source system rather than conducting controlled experiments. The analysis focused on artifacts and tasks and was not specifically tailored to Java development in general or the Mylyn project in particular. With respect to the data set quality, we noticed that not all tasks in the Bugzilla issue tracking system provided an interaction attachment. As Mylyn interaction attachment upload is neither automated nor mandatory, we were limited to a subset of all tasks. Hence, we very likely were unable to detect several dependencies, respectively couldn't evaluate them on tasks that might have benefitted from them. Overall, the Mylyn dataset is rich enough for a sufficiently long, continuous interval to allow successful mining and recommendation.

External Validity We analysed only a single data set as we are not aware of other real world projects aside from Mylyn that make a significant amount of task-centric interaction events available. Mylyn interaction data upload capabilities are not available by default and thus not widely used beyond the Mylyn project. Hence, we are careful to generalize our findings beyond the scope of the Mylyn project. Our analysis, however, demonstrated that it is indeed possible to detect cross-task dependencies from interactions and motivates further research in this direction. As outlined in future work (Section X), alternative data sources such as commit information might possibly allow for cross-task mining. Commit data, however, lacks (i) read-only events which reduces the detection rate and (ii) temporal information which precludes interaction replaying for evaluation purposes.

Construct Validity The replay approach is a proxy of usefulness as we consider only those recommendations as successful where the developer eventually accessed the recommended artifact. We might have recommended artifacts that are indeed relevant but the developer at that time wasn't aware of these artifacts, hence didn't access them, and we therefore regarded them as false positives. We, therefore, cannot assume that all false positives were indeed inaccurate. We thus can primarily claim that our recommender is helpful for remembering which artifacts to assess, which in non-trivial systems is important nevertheless. We refrained from interviewing developers from the Mylyn project as the time frame with sufficient interaction data to detect dependencies and evaluate them is almost

10 years ago. We would not expect feedback to provide meaningful insights after such a long time.

IX. Related Work

Investigations into change impact among code artifacts studied the logical coupling between artifacts, i.e., which artifacts tend to co-evolve [27], [29], [30]. These approaches observe which artifacts frequently occur in the same commit (or in temporal proximity) independent of the task that the changed happened in.

Few approaches analyse control and data flow among code artifacts [25], mine association rules from software revision histories [17], [23] , or utilize a variability model to detect the impact within product families [1].

Several researchers consider developers' interaction histories to augment traces among logically coupled source code artifacts [2], [3], [16]. Kostadin et al. [5] use low-level IDE interaction to detect hidden behavior of developers. Bantelay et al. [2] combine interaction histories and commit data to improve the detection of evolutionary coupling between artifacts. Sebastian et al. [22] use developer activities in the IDE with context information, such as source-code snapshots for change events to study developer behavior. These approaches aim to find traces among code artifacts without considering contextual information such as the task the developer is working on. However, Wiese et al. [28] apply contextual information collected from tasks, developers' communication, and commit data to capture the change patterns of artifacts. They use this contextual information to improve the artifacts co-change prediction. We focus specifically on cross-task dependencies for identifying dependencies with dedicated coordination needs.

Several task-centric approaches consider fine-grained developer interactions but restricts analysis to interactions within a task without considering the relations to other tasks. Kersten and Murphy [15] introduce the Mylyn tool for determining which code artifacts are relevant for a particular development task. Their analysis, however, is limited to a single developer within a single task.

Hipikat [4] supports the developer in retrieving relevant artifacts from the project's overall history. It considers documents, tasks, commits, messages, and artifact changes but not the detailed engineering interaction history. A tool that capture the interaction occurred in a particular file is HeatMap [24]. Another tool, Wolf [8] extracts artifact ownership and changes from source code repositories and generates traces between artifacts and engineers. The tool provides an organizational view for managers and an individual view for developers to support impact analysis activities. However, such tools focus on the relation between artifacts and tasks but not necessarily on the dependencies of artifact across tasks.

Multiple authors investigate inter-task relationships. Thompson et al. [26] study how software developers use relationship between tasks based on their titles to breakdown project work. Their study indicates that finding relationship between tasks can improve software development techniques. Mayr-Dorn et al. [20] investigated if the propagation of artifact changes across tasks reflect work dependencies among them. They observed that task links are useful for recommending artifacts to monitor for changes and these links can also potentially be used to recommend cross task dependencies. However, their focus was on whether the same artifact is changed in two tasks, while we investigate whether distinct artifacts are changed.

Related work with respect to horizontal traces (i.e., dependencies among code and non-code artifacts) falls into two categories: (semi)-auto-matically establishing traces [12] and maintaining traces under system evolution. Examples include Guo et al. [13] who apply domain specific knowledge to generate traces between requirements and code. Ghabi and Egyed [11] identify likely incorrect or missing traces between requirement and code by comparing trace patterns and source code calling relationships. Mahmoud and Niu [19] suggest refactoring techniques to improve and re-establish traceability between requirement documents and source code.

Examples for trace maintenance approaches include Mäder et al. [18] who use UML model changes to trigger automatic traceability maintenance rules. Jiang et al. [14] apply incremental latent semantic indexing to automatically manage traceability links between code and documentation. Nejati et al. [21] demonstrate the use of natural language processing to automatically identify the impact of requirements changes on system design.

Approaches to supporting the management of horizontal trace represent orthogonal approaches to our work. Combining horizontal traces approaches with cross-task artifact mining could potentially identify which horizontal traces require the most coordination effort. Alternatively, our approach could identify implicit horizontal traces that have not been explicitly modeled.

X. conclusions and Future Work

We presented an approach for mining cross-task artifact dependencies from developer interaction events. We described a heuristic for pairing up tasks from which to mine dependencies using a state-of-the-art sequential rule mining algorithm. Detected dependencies allow the recommendation of artifacts to be inspected for change impact analysis. We evaluated our approach on the Mylyn data set and demonstrated that we are able to detect dependencies over a considerable project duration (given the available data) that also resulted in usable recommendations to the developers. We provided correct recommendations in $\sim$30% of all tasks where we had matching dependencies and ranked 50% of all artifact recommendations within the top 5 results.

Our future work consists of two activities. On the one hand, we will evaluate the impact of exluding read-only data and compare the results obtained from the Mylyn data set with other data sets that consist of commit data only. Publicly accessible issue trackers such as Eclipse Bugzilla, Apache Jira, and Github host various projects that provide task and commit data. On the other hand, we will focus on evaluating other uses of dependencies such as task dependency analysis, task similarity analysis, or developer network analysis.

References

[1] Florian Angerer, Andreas Grimmer, Herbert Prahofer, and Paul Grunbacher. Configuration-aware change impact analysis (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 385–395. IEEE, 2015.

[2] Fasil Bantelay, Motahareh Bahrami Zanjani, and Huzefa Kagdi. Comparing and combining evolutionary couplings from interactions and commits. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 311–320. IEEE, 2013.

[3] Kelly Blincoe, Giuseppe Valetto, and Daniela Damian. Facilitating coordination between software developers: A study and techniques for timely and efficient recommendations. *IEEE Transactions on Software Engineering*, 41(10):969–985, 2015.

[4] Davor Cubranic, Gail C Murphy, Janice Singer, and Kellogg S Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[5] Kostadin Damevski, Hui Chen, David Shepherd, and Lori Pollock. Interactive exploration of developer interaction traces using a hidden markov model. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 126–136. ACM, 2016.

[6] Cleidson de Souza and David Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering, ICSE'08*, pages 241–250. IEEE, 2008.

[7] Dario Di Nucci, Fabio Palomba, Sandro Siravo, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. On the role of developer's scattered changes in bug prediction. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 241–250. IEEE, 2015.

[8] Mayara C Figueiredo and Cleidson RB de Souza. Wolf: Supporting impact analysis activities in distributed software development. In *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*, pages 40–46. IEEE Press, 2012.

[9] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.

[10] Philippe Fournier-Viger, Roger Nkambou, and Engelbert Mephu Nguifo. A knowledge discovery framework for learning task models from user interactions in intelligent tutoring systems. In *Mexican International Conference on Artificial Intelligence*, pages 765–778. Springer, 2008.

[11] Achraf Ghabi and Alexander Egyed. Code patterns for automatically validating requirements-to-code traces. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 200–209. IEEE, 2012.

[12] Marek Gibiec, Adam Czauderna, and Jane Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 245–254. ACM, 2010.

[13] Jin Guo, Natawut Monaikul, Cody Plepel, and Jane Cleland-Huang. Towards an intelligent domain-specific traceability solution. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 755–766. ACM, 2014.

[14] Hsin-Yi Jiang, Tien N Nguyen, Xiang Chen, Hojun Jaygarl, and Carl K Chang. Incremental latent semantic indexing for automatic traceability link evolution management. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 59–68. IEEE Computer Society, 2008.

[15] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11. ACM, 2006.

[16] Martin Konôpka and Mária Bieliková. Software developer activity as a source for identifying hidden source code dependencies. In *Proceedings of the International Conference on Current Trends in Theory and Practice of Informatics*, pages 449–462. Springer, 2015.

[17] Seonah Lee, Sungwon Kang, Sunghun Kim, and Matt Staats. The impact of view histories on edit recommendations. *IEEE Transactions on Software Engineering*, (1):1–1, 2015.

[18] Patrick Mader, Orlena Gotel, and Ilka Philippow. Enabling automated traceability maintenance by recognizing development activities applied to models. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 49–58. IEEE Computer Society, 2008.

[19] Anas Mahmoud and Nan Niu. Supporting requirements to code traceability through refactoring. *Requirements Engineering*, 19(3):309–329, 2014.

[20] Christoph Mayr-Dorn and Alexander Egyed. Does the propagation of artifact changes across tasks reflect work dependencies? In *Proceedings of the 40th International Conference on Software Engineering, ICSE'18*, pages 397–407. ACM, 2018.

[21] Shiva Nejati, Mehrdad Sabetzadeh, Chetan Arora, Lionel C Briand, and Felix Mandoux. Automated change impact analysis between sysml models of requirements and design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 242–253. ACM, 2016.

[22] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. Enriching in-ide process information with fine-grained source code history. In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 250–260. IEEE, 2017.

[23] Thomas Rolfsnes, Leon Moonen, and David Binkley. Predicting relevance of change recommendations. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 694–705. IEEE, 2017.

[24] David Rothlisberger, Oscar Nierstrasz, Stéphane Ducasse, Damien Pollet, and Romain Robbes. Supporting task-oriented navigation in ides with configurable heatmaps. In *Proceedings of the IEEE 17th International Conference on Program Comprehension, ICPC'09*, pages 253–257. IEEE, 2009.

[25] Neha Rungta, Suzette Person, and Joshua Branchaud. A change impact analysis to characterize evolving program behaviors. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 109–118. IEEE, 2012.

[26] C Albert Thompson, Gail C Murphy, Marc Palyart, and Marko Gašparič. How software developers use work breakdown relationships in issue repositories. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 281–285. ACM, 2016.

[27] László Vidács and Martin Pinzger. Co-evolution analysis of production and test code by learning association rules of changes. In *Proceedings of the IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 31–36. IEEE, 2018.

[28] Igor Scaliante Wiese, Reginaldo Ré, Igor Steinmacher, Rodrigo Takashi Kuroda, Gustavo Ansaldi Oliva, Christoph Treude, and Marco Aurélio Gerosa. Using contextual information to predict co-changes. *Journal of Systems and Software*, 128:220–235, 2017.

[29] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.

[30] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.