

Tracking and Analyzing Cross-Cutting Activities in Developers' Daily Work

Lingfeng Bao^{1,2}, Zhenchang Xing², Xinyu Wang¹, and Bo Zhou¹

¹College of Computer Science, Zhejiang University, Hangzhou, China

²School of Computer Engineering, Nanyang Technological University, Singapore
{lingfengbao, wangxinyu, bzhou}@zju.edu.cn; {zcxing}@ntu.edu.sg;

Abstract—Developers use many software applications to process large amounts of diverse information in their daily work. The information is usually meaningful beyond the context of an application that manages it. However, as different applications function independently, developers have to manually track, correlate and re-find cross-cutting information across separate applications. We refer to this difficulty as *information fragmentation problem*. In this paper, we present *ActivitySpace*, an interapplication activity tracking and analysis framework for tackling information fragmentation problem in software development. *ActivitySpace* can monitor the developer's activity in many applications at a low enough level to obviate application-specific support while accounting for the ways by which low-level activity information can be effectively aggregated to reflect the developer's activity at higher-level of abstraction. A system prototype has been implemented on Microsoft Windows. Our preliminary user study showed that the *ActivitySpace* system is promising in supporting interapplication information needs in developers' daily work.

I. INTRODUCTION

Software developers are a typical example of knowledge workers [1]. Their work produces, consumes and communicates large amounts of diverse information. Integrated Development Environments (IDEs) have been designed to maximize developer productivity by integrating separate software development tools, such as editor, build tool, debugger, version control, and project management. However, today's software development involves not only software development tools but also many other software applications specializing in different tasks. For example, developers use web browsers to search the Web, ask or answer questions in Q&A sites, or communicate with other developers on social media.

The information associated with the work of developers is usually meaningful beyond the context of an application that manages it. For example, the developer encounters an unexpected error in the IDE, and then searches the Web for solutions. He reads some web pages and finds a solution on Stack Overflow. This solution contains a code example shared on jsfiddle.net. He tests the code example on jsfiddle and finds the code useful for his error. He finally integrates the code example in his code. In this example, source code the developer works on, program error he encounters, search queries he uses, web pages he visits, and code example on jsfiddle are all correlated, but they are managed in various desktop and web applications, including IDE, web browser, Stack Overflow, jsfiddle, respectively.

These applications function independently of one another. This independence creates a problem of *information fragmen-*

tation, forcing the developer to manually correlate and re-find cross-cutting information across separate applications. For example, during a task, the developer may be interrupted by another task. Once he is done with the other task, he needs to resume his working context across several applications for the previous task, including not only source files but also Stack Overflow posts and code examples on jsfiddle. As another example, two week later, the developer's colleague encounters a similar error. The developer believes that one of the Stack Overflow posts he read before would be useful for his colleague. Web browsing history does not help much because there are too many browsing activities on Stack Overflow that are irrelevant to that particular error. He recalls the source file involved in his previous task. However, the information fragmentation problem makes it difficult for him to re-find the Stack Overflow posts that he visited while he worked on that source file two weeks ago.

Today's software development practice calls for innovations to support *interapplication information needs* in software development, going beyond tool integration (e.g., IBM Jazz [2], SeaHawk [3]) and information management within separate applications (e.g., Mylyn [4], Context Web History [5]). In this work, we aim to develop mechanisms that unobtrusively track and effectively aggregate cross-cutting activities in developer's daily work, to ease the process of re-finding cross-cutting information across applications.

The challenge is to monitor the developer's activity in many applications at a low enough level to obviate application-specific support while accounting for the ways in which information at this low level fails to accurately reflect the developer's activity at higher-level of abstraction. To tackle this challenge, we design *ActivitySpace* system that supports interapplication activity tracking, aggregation, search, and exploration. We have implemented a *ActivitySpace* prototype on Microsoft Windows. We conducted a user study of the *ActivitySpace* prototype, in which we collected 417 hours activity data from 8 participants. Using this data, we analyzed the information fragmentation problem in software development and showed that *ActivitySpace* is promising in supporting interapplication information needs in developers' daily work.

II. THE *ActivitySpace* SYSTEM

Fig. 1 shows the architecture of our *ActivitySpace* system. *ActivitySpace* is composed of three components: 1) *Activity Tracker* supports real-time recording of the developer's activity in different applications using operating-system (OS) level

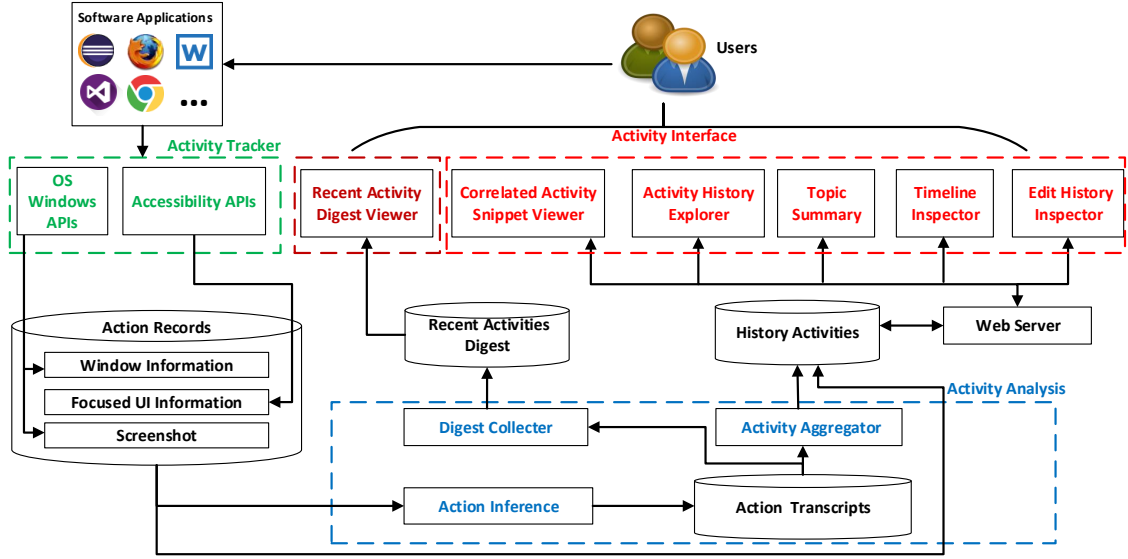


Fig. 1: The Architecture of the ActivitySpace System

instrumentation and computer vision techniques; 2) *Activity Analysis* identifies cross-cutting activities and documents likely to be relevant based on *temporal locality* [6]; and 3) *Activity Interface* provides *episodic and semantic User Interfaces (UIs)* to help the developer search and browse cross-cutting activities and documents from various perspectives.

A. Activity Tracker

The *ActivityTracker* component monitors the developer's actions in the applications being used. It generates a sequence of time-ordered *action records* stored in the action records database. Action records collect the low-level Human-Computer Interaction (HCI) data, including window information, focused UI information and screenshot.

Three factors must be taken into account when designing the *ActivityTracker*: *generality* (i.e., easy to deploy to track the developer's actions in a wide range of commonly used software applications), *unobtrusiveness* (i.e., do not disturb the developers' normal work flow), and *efficiency* (i.e., real-time activity tracking and data collection). At the same time, the collected action data should be easy to aggregate to infer the developer's activity at higher-level of abstraction.

To satisfy these design requirements, the *ActivityTracker* component combines OS level instrumentation and screen capture technique. As the developer is using a software application, OS level instrumentation collects window and focused UI information that applications expose to operating system, while screen-capture technique records the screenshots which can be analyzed using computer vision techniques to infer the developer's actions. These two techniques do not require application-specific support, and thus can be easily deployed in many applications and operating systems. Furthermore, they can provide real-time activity tracking with very little data collection overhead, and will not disturb the developer's work.

1) *OS Level Instrumentation*: *ActivityTracker* uses a mouse hook to monitor mouse click actions. The user can specify which application(s) he wants or does not want *ActivityTracker* to monitor. Once a mouse click occurs in a to-be-monitored

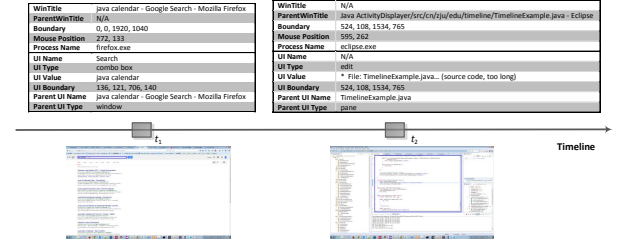


Fig. 2: An Example of Action Records

application, *ActivityTracker* uses OS window APIs and accessibility APIs to extract two kinds of information: window information and focused UI information. The extracted information will be time-stamped and stored as an action record in the action records database.

Using OS window APIs, *ActivityTracker* extracts the following window information: the *WinTitle* and *Boundary* of the focused application window in which the mouse click occurs, the *Position* of mouse click, the *ParentWinTitle* of the root parent window of this application window, and the *Process Name* of the application. Using accessibility APIs, *ActivityTracker* extracts the following information of the focused UI component: *UI Name*, *UI Type*, *UI Value* and *UI Boundary* of the focused UI component, and the *UI Name* and *UI Type* of the root parent UI component.

Due to privacy reason, the keyboard monitoring is disabled by default. If keyboard monitoring is enabled, *ActivityTracker* uses a keyboard hook to monitor keyboard actions in a same way as monitoring mouse click actions.

Fig. 2 shows an example of two action records. The first action occurs in Firefox. The developer searches *java calendar* on Google. The second action occurs in Eclipse. The developer works on *TimelineExample.java* source file in code editor.

2) *Screen Capture*: Accessibility API provides a generic way to track the developer's actions in different software applications. However, it requires application developers to

invest additional engineering effort to properly expose the internal data of the application when developing the software. As a result, not all applications expose their internal data to accessibility API, or not all the information is exposed¹.

To deal with the variety of accessibility support and to capture the application context of the developer's actions, *ActivityTracker* uses OS window APIs to record a screenshot of the application when a mouse click or keyboard action occurs in the application. This screenshot provides a supplementary information for accessibility information. The screenshots will be stored together with the corresponding action records in the action records database. The screenshots will be used in two ways. First, it can be aggregated with the mouse position information to infer application-specific actions that cannot be tracked using accessibility API, for example, setting breakpoint in Eclipse code editor. Second, replaying a time-series of screenshots can help the developer recall when he did what by invoking episodic memory [7].



B. Activity Analysis

As a new action record arrives on the server, the *ActionInference* component first infers an *action transcript* (application being used, developer action, document involved and document content) from the action record. Action transcript is stored in the action transcripts database. Then, the *DigestCollector* component updates application usage and document usage statistics for the most recent action transcripts in a fixed-length time window. These usage statistics are referred as *recent activities digest* stored in recent activities digest database. The *ActivityAggregator* component runs at a given time interval to periodically aggregates action transcripts into *history activities* stored in the history activities database. An *activity* represents all the actions performed on a document within an application during a period of time. Further information such as correlated activities, topics in documents used, edit history of documents can be extracted from activities. Note that these activity analyses are time consuming, and thus cannot be performed by the *ActivityTracker*. Otherwise, the *ActivityTracker* cannot satisfy the *unobtrusiveness* and *efficiency* requirements.


1) *Inferring Developer Actions*: The *ActionInference* component first tries to infer action transcript from the window and focused UI information of the action record. Application being used can be determined from the *Process Name* attribute of the window information. Document involved in the action can be determined from the *WinTitle* or *ParentWinTitle* attribute of the window information. For example, in Fig. 2, at time t_1 the application being used is *firefox.exe* and the document involved is *java calendar - Google Search*. At time t_2 the application being used is *eclipse.exe* and the document involved is *.../TimelineExample.java*.

If the focused UI component supports accessibility API, *ActionInference* can use the *UI Name* and/or *UI Type* attributes of the focused UI information to infer the developer action over the UI component. Furthermore, the detailed document content can usually be extracted from the *UI Value* attribute.

For example, at time t_1 the developer action is *Search (UI Name)* and the document content is *java calendar (UI Value)*. At time t_2 the developer is in an edit component (*UI Type*), the source file is *TimelineExample.java* (*Parent UI Name*), and the file content can be extracted from the *UI Value* attribute.

However, not all UI components expose accessibility information to accessibility API. This is especially the case for fine-grained UI components, such as buttons or UI decorators. For example, the breakpoint  on the editor ruler in Eclipse edit component cannot be accessed by accessibility API. As another example, the tab close button  in Firefox tabs cannot be accessed by accessibility API. In such cases, *ActionInference* will use image template matching techniques to determine the UI component that the developer interacts with and to infer the developer's action.

ActionInference uses two kinds of image template matching techniques for different situations, as implemented in our video scraping tool *scvRipper* [8]. The first technique is key point based template matching [9], [10]. The second technique is template matching with alpha mask [11], because some small icons may not have enough key points. To detect the small icons in the screenshot, the *ActivitySpace* user needs to provide some template images of the icons that he wants the system to recognize and associate the template images with some actions.

Once the icon is detected in the screenshot, *ActionInference* will examine the mouse position against the icon position in the screenshot to determine whether the mouse action is over the icon. Then, based on the action definition associated with the icon, *ActionInference* can determine what action the developer performs in the action record. For example, *ActionInference* can infer that the developer action is *set breakpoint in code editor* by recognizing the breakpoint icon  and linking it to mouse action. This computer-vision based action recognition complements the accessibility information.

2) *Generating Recent Activity Digest*: The *DigestCollector* component maintains a recent activity digest for the most recent actions in a fixed-length time window. Once a new action transcript is added in the action transcripts database, *DigestCollector* finds the archived action transcripts within X minutes ($X = 30$ by default) from the latest action transcript. It then collects the application usage and document usage statistics in these action transcripts, including applications being used and the time spent on each application, and document being used and the time spent on each document. These application usage and document usage statistics constitute recent activity digest in the latest activity digest time window, which will be updated in *Recent Activity Digest Viewer*.

3) *Aggregating History Activities*: At a given time interval Y minutes ($Y = 60$ by default), the *ActivityAggregator* component will be launched to aggregate the action transcripts archived in a period of Y minutes into history activities. The new action transcripts can keep arriving after *ActivityAggregator* has been launched. These new action transcripts will be processed in the next execution of *ActivityAggregator*.

ActivityAggregator assumes that the information used by the developer across separate applications exhibit *temporal locality*, i.e., actions which occur at closer points in time are more likely to be conceptually related. Temporal locality has

¹ Visit <http://baolingfeng.weebly.com/accessibility-survey.html> for the results of our investigation of accessibility support in commonly used software applications on Windows, Mac, and Linux operating systems.

been applied with success in numerous settings involving user activity [12]–[14]. Temporal locality allows *ActivityAggregator* to infer a likely relevance among different documents without knowledge of their contents or the purpose of the actions.

Let $\langle a_1, a_2, \dots, a_n \rangle$ be a sequence of action transcripts in the latest Y minutes since the last execution of the *ActivityAggregator*. *ActivityAggregator* regards as an activity all the actions on a particular document within an application window that occur within the time interval threshold T_{close} (30 minutes by default). Given the two actions a_i and a_j ($1 \leq i < j \leq n$), a_i and a_j are regarded as part of an activity if the time span between the two actions $|a_i.time - a_j.time| < T_{close}$ and the two actions have the same process name and the same window title. If an application window is a sub-window of the main application window, e.g. open a dialog window in an application window, *ActivityAggregator* uses the title of the main application window for comparison. If the time span of the two activities overlap, *ActivityAggregator* considers the two activities as correlated activities, and the documents involved in the two activities as correlated documents.

C. Activity Interface

ActivitySpace currently supports three semantic UIs (*Recent Activity Digest Viewer*, *Correlated Activity Snappit Viewer*, and *Topic Summary*), and three episodic UIs (*Activity History Explorer*, *Timeline Inspector*, and *Edit History Inspector*). Semantic UIs enable the developer to find information based on the documents (i.e., *what*) he has used, while episodic UIs enable the developer to find information based on the time (i.e., *when*) and place (i.e., *where*) of his actions. Readers are referred to our tool website² for detailed description of these semantic and episodic UIs.

III. EVALUATION

We have implemented a *ActivitySpace* prototype² on Microsoft Windows. We invited 8 volunteer graduate students in a user study of the *ActivitySpace* prototype. The participants installed the *ActivitySpace* prototype on their working computer. The system was configured to track user activity in web browsers (*Firefox*, *Chrome*, *Internet Explorer*), office software (*Word*, *Excel*, *PowerPoint*), PDF reader (*Adobe Reader*, *Foxit Reader*), text editors (*Notepad*, *Notepad++*), latex editor (*WinEdt*), and IDEs (*Eclipse*, *Visual Studio*). Activity in all other applications are categorized as others. In this section, we analyze the collected activity data and report the post-study survey results.

A. Activity Statistics

The 8 participants used the *ActivitySpace* system for 4 to 22 days. We collected in total 417 hours activity data. The total time of each participant is computed as the sum of the interval of adjacent action records, but we discard long idle time between the two action records whose interval is longer than 30 minutes. Table I summarizes the collected activity data in these 417 hours. The participants S1-S5 used the *ActivitySpace* system for more than 10 days and produced sufficient activity data for analysis. They used the tracked

applications on average more than 4 hours per day and used large amounts of distinct documents.

Table II shows the document usage and application usage statistics by the participants S1-S5. We can see that S1-S5 used many software applications in their work. S1-S5 visited large amounts of distinct web pages. Distinct web pages visited by S1-S5 account for more than 85% of all the documents used by S1-S5, while S1-S5 used a small number of distinct documents in other categories of applications. However, looking at application usage statistics, S1-S5 spent only about 50% of their working time in web browser. This means that these participants usually spent very short time on a web page visited. For example, they often opened some web pages after search, and then closed the page after a quick look. In contrast, the participants usually spent much longer time on a document in other categories of applications.

TABLE I: The Statistics of Activity Data

	#Day	Duration [hour]	#ActionRecord (avg per hour)	#Activity (avg per hour)	#Distinct Document
S1	22	89.25	45667 (324.39)	3981 (28.28)	2401
S2	21	87.09	35010 (381.96)	4471 (48.78)	2898
S3	21	98.15	20202 (185.77)	3465 (31.86)	1843
S4	19	68.45	34576 (439.23)	3433 (43.61)	1644
S5	11	52.63	12993 (245.75)	2161 (40.87)	1078
S6	5	3.97	1961 (496.46)	130 (32.91)	99
S7	5	5.71	635 (115.04)	121 (21.92)	41
S8	4	11.96	4019 (327.55)	635 (51.75)	358
Total	108	417.2	155063	18397	10362

TABLE II: The Document and Application Usage

(a) The Usage of Distinct Documents (number)

	S1	S2	S3	S4	S5
All Applications	2401	2893	1843	1644	1078
Browser	2096 (87.30%)	2641 (91.29%)	1710 (92.78%)	1446 (87.96%)	915 (84.88%)
Office Software	68 (2.83%)	48 (1.66%)	22 (1.19%)	69 (4.20%)	12 (1.11%)
PDF Reader	20 (0.83%)	39 (1.35%)	31 (1.68%)	38 (2.31%)	NA
Text Editor	65 (2.71%)	29 (1.00%)	8 (0.43%)	37 (2.25%)	121 (11.22%)
Latex Editor	22 (0.92%)	89 (3.08%)	59 (3.20%)	34 (2.07%)	30 (2.78%)
Eclipse	110 (4.58%)	24 (0.83%)	13 (0.71%)	20 (1.22%)	NA
Visual Studio	13 (0.54%)	NA	NA	NA	NA
Others	7 (0.29%)	23 (0.80%)	NA	NA	NA

(b) The Usage of Applications (hour)

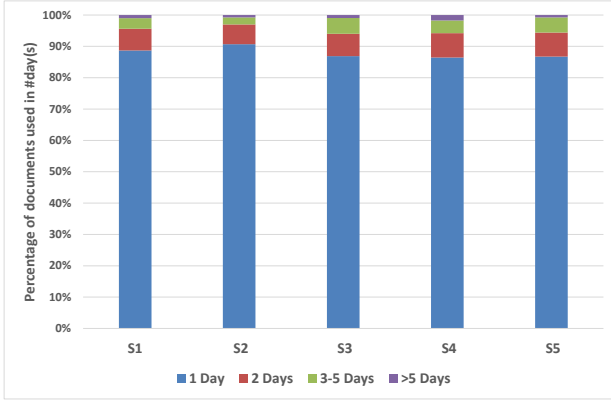
	S1	S2	S3	S4	S5
All Applications	89.25	87.09	98.15	68.45	52.63
Browser	41.26 (46.23%)	50.5 (57.99%)	57.41 (58.49%)	38.42 (56.13%)	26.88 (51.07%)
Office Software	2.58 (2.89%)	3.65 (4.19%)	1.05 (1.07%)	10.63 (15.53%)	0.45 (0.86%)
PDF Reader	0.09 (0.10%)	0.74 (0.85%)	1.46 (1.49%)	1.03 (1.50%)	NA
Text Editor	0.71 (0.80%)	0.26 (0.30%)	0.19 (0.19%)	0.89 (1.30%)	11.67 (22.17%)
Latex Editor	15.08 (16.90%)	28.84 (33.12%)	31.81 (32.41%)	13.33 (19.47%)	13.63 (25.90%)
Eclipse	28.3 (31.71%)	2.82 (3.24%)	6.23 (6.35%)	4.15 (6.06%)	NA
Visual Studio	1.12 (1.25%)	NA	NA	NA	NA
Others	0.11 (0.12%)	0.28 (0.32%)	NA	NA	NA

B. Document Revisits

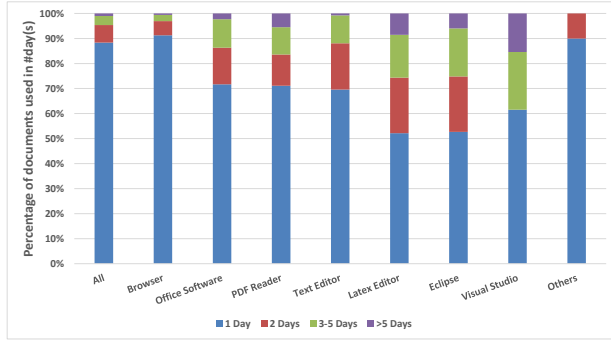
As shown in Fig. 3a, about 10% of the documents that each participant used were used in 2 or more days. Fig. 4 shows the statistics of time intervals (days) of document revisits. We can see that many documents revisits occur after a long time interval. For example, S2 revisited the web page “User GManNickG - Stack Overflow” 14 days after the first visit. Fig. 3b shows the percentage of revisited documents of the participants S1-S5 by different applications. We can see that although the five participants visited thousands of distinct web pages, there were only about 10% of web pages that were visited in 2 or more days. In contrast, in other categories of applications (e.g., Latex Editor and IDEs) higher percentage of documents were revisited in 2 or more days.

As shown in Table III, the participants searched Google with hundreds of distinct queries during the study. A small percentage of queries were reused in 2 or more days. We find that the participants sometimes use Google as a “bookmark” to revisit the relevant web information they visited before.

²Visit <http://baolingfeng.weebly.com/ase2015-demonstration.html> for tool demonstration and installation information.



(a) By Different Participants



(b) By Different Software Applications

Fig. 3: The Statistics of Document Revisits

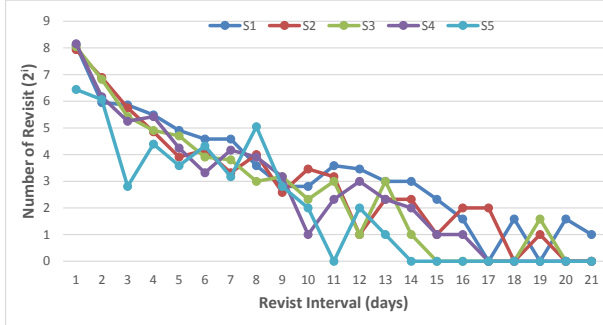


Fig. 4: The Statistics of Document Revisit Intervals

C. Correlated Activities and Documents

If consecutive action transcripts involved different documents within an application, we regard it as a within-application transition. If consecutive action transcripts involve different categories of applications, we regard it as an across-application transition. Fig 5 shows a discrete-time Markov chain for the within- and across-application transitions in the work of the participant S3. Node size is proportional to the percentage of distinct documents used in a particular category of applications (see Table IIb). We can see that within-applications account for large percentage of transitions (blue font). Meanwhile, S3 also frequently switches from one application to another, especially from other applications to browser (red font). This shows that S3 interleaves coding, web search and learning in his work.

TABLE III: The Statistics of Search Queries on Google

	#QueryInGoogle	#Re-UsedQuery	Percentage (%)
S1	421	17	4.038%
S2	227	10	4.405%
S3	448	22	4.911%
S4	342	17	4.971%
S5	234	12	5.128%

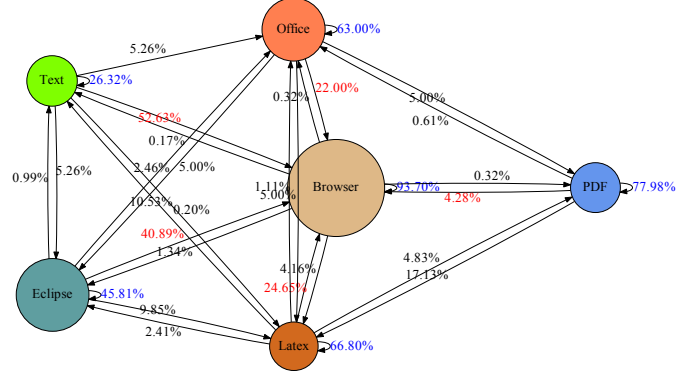


Fig. 5: The Application Transitions in the work of S3

Table IV shows the documents with the most correlated documents for each category of software application in the work of S3. We can see that these documents correlate with many other documents within- and across-applications. For non-browser applications, most of the correlated documents are in different applications.

TABLE IV: The Document with the Most Correlated Documents in the work of S3

	Document With The Most Correlated Documents	#CorrelatedDocument	#CorrelatedDocument InDifferentApplication
Browser	#S3's name# - Outlook	193	37 (19.17%)
Office Software	...full conference name.xlsx	12	11 (91.67%)
PDF Reader	draft.pdf - Adobe Acrobat Reader DC	53	34 (64.15%)
Text Editor	log.log - Notepad	17	16 (94.12%)
Latex Editor	WinEdt 9.0 ... /draft.tex	76	62 (81.58%)
Eclipse	UrlProcess.java (URLMatch/src)	49	43 (87.76%)

D. Post-Study Survey

The 8 participants did not report difficulty in installing the *ActivitySpace* prototype. The *ActivitySpace* prototype can successfully track activity data in the diverse computer settings of the 8 participants. The participants did not feel the overhead of activity tracking. They did not report that the *ActivitySpace* prototype disturbs their work.

TABLE V: The Usefulness of *ActivityInterface* UIs

Activity Interface	Usefulness (mean \pm std)
Recent Activity Digest Viewer	4.0 \pm 0.63
Correlated Activity Snippet Viewer	4.2 \pm 0.75
Activity History Explorer	4.2 \pm 0.40
Topic Summary	3.4 \pm 1.36
Timeline Inspector	3.4 \pm 0.80
Edit History Inspector	3.2 \pm 1.17

We conducted a post-study using a questionnaire in which we asked the participants rate the usefulness of the semantic and episodic UIs in 5-points likert scales (1 being least useful, 5 being most useful). Table V summarizes the participants' feedbacks. The participants found *Correlated Activity Snippet Viewer* useful, because it supports context-sensitive recall of documents previously used together with the current document

(S3). They also found *Activity History Explorer* useful, because it not only provides a comprehensive overview of history activities but also supports effective ways to filter or search relevant activities/documents (S1). Some participants found *Recent Activity Digest Viewer* useful, because it allows them to quickly find recently used documents. Most participants did not find *Topic Summary*, *Timeline Inspector*, and *Edit History Inspector* very useful, except S1 and S4. S4 used *Topic Summary* to find the web pages he visited. S1 made many code changes in Eclipse and found *Edit History Inspector* useful to recall the intermediate code changes that were not committed in version control system.

IV. RELATED WORK

Tracking User Activity. Software applications can be instrumented to log the user's activity within an application, for example TaskTracker [15], Mylyn [4], and Reverb [16]. Software instrumentation usually requires sophisticated reflection APIs provided by applications and GUI toolkits, which are not always available [17]. Furthermore, instrumenting many of today's software systems is considerably complex. Accessibility APIs are standard interfaces built in modern operating systems. They provide a generic way to access application information, but what information an application exposes to the accessibility APIs is determined by the application developers which vary greatly across applications and operating systems¹. Furthermore, it is usually problematic to derive user interest and activity from highly fine-grained accessibility data without the information about application context. To overcome this limitation, our *ActivityTracker* complements accessibility information with screenshots of applications.

Activity-Centric Computing. Our *ActivitySpace* system is inspired by the activity theory [18] and the recent development of activity-centric computing tools [19]–[22]. Activity-centric computing has been proposed as a computing paradigm that supports the users' activities rather than the resources and tools used to perform such activities. Activity-centric computing goes beyond tool integration and task-centric management in separate applications. However, existing activity-centric systems mainly focus on activity construction and activity resumption. In contrast, our *ActivitySpace* system unobtrusively track developer activity in different applications and automatically infer high-level activities and their correlations.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented our *ActivitySpace* system that supports interapplication activity tracking, aggregation, search and exploration. *ActivitySpace* creates interapplication information scents to ease the process of correlating and re-finding cross-cutting activities and documents across applications in developers' daily work. A pilot study of 8 participants demonstrated the usefulness of the *ActivitySpace* prototype for supporting interapplication information needs in software development. In the future, we will investigate activity-centric virtual workspace and community-of-practice for better supporting integrated knowledge work in today's software development practices, which requires higher-level information integration and management than current focus on tool integration and within-application task management.

ACKNOWLEDGMENT

This work was partially supported by the Major State Basic Research Development Program of China (973 ProgramNo.2015CB352201) and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2013BAH01B01). This work is supported by NTU SUG M4081029.020 and MOE AcRF Tier1 M4011165.020.

REFERENCES

- [1] T. H. Davenport, *Thinking for a living: how to get better performances and results from knowledge workers*. Harvard Business Press, 2013.
- [2] R. Frost, "Jazz and the eclipse way of collaboration," *Software, IEEE*, vol. 24, no. 6, pp. 114–117, 2007.
- [3] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack overflow in the IDE," in *Proc. ICSE*, pp. 1295–1298, 2013.
- [4] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proc. AOSD*, pp. 159–168, 2005.
- [5] S. S. Won, J. Jin, and J. I. Hong, "Contextual web history: Using visual and contextual cues to improve web browser history," in *Proc. CHI*, pp. 1457–1466, 2009.
- [6] M. Snir and J. Yu, "On the theory of spatial and temporal locality," 2005.
- [7] M. Lamming and M. Flynn, "Forget-me-not: intimate computing in support of human memory," in *Proceedings of the '94 Symposium on Next Generation Human Interface (FRIEND21)*, pp. 2–4, 1994.
- [8] L. Bao, J. Li, Z. Xing, X. Wang, and B. Zhou, "Reverse engineering time-series interaction data from screen-captured videos," in *Proc. 22nd IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 399–408, 2015.
- [9] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. ICCV*, vol. 2, pp. 1150–1157, 1999.
- [10] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Computer vision and image understanding*, vol. 110, no. 3, pp. 346–359, 2008.
- [11] D. A. Forsyth and J. Ponce, *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, 2002.
- [12] N. Oliver, G. Smith, C. Thakkar, and A. C. Surendran, "Swish: Semantic analysis of window titles and switching history," in *Proc. IUI*, pp. 194–201, 2006.
- [13] T. Rattenbury and J. Canny, "Caad: An automatic task support system," in *Proc. CHI*, pp. 687–696, 2007.
- [14] C. A. N. Soules and G. R. Ganger, "Connections: Using context to enhance file search," in *Proc. SOSP*, pp. 119–132, 2005.
- [15] A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker, "TaskTracer: a desktop environment to support multi-tasking knowledge workers," in *Proc. IUI*, p. 75, 2005.
- [16] N. Sawadsky, G. C. Murphy, and R. Jiresal, "Reverb: Recommending code-related web pages," in *Proc. ICSE*, pp. 812–821, IEEE, 2013.
- [17] A. Hurst, S. E. Hudson, and J. Mankoff, "Automatically identifying targets users interact with during real world tasks," in *Proc. IUI*, pp. 11–20, 2010.
- [18] Y. Engeström, R. Miettinen, and R.-L. Punamäki, *Perspectives on activity theory*. Cambridge University Press, 1999.
- [19] S. Jeuris, S. Houben, and J. Bardram, "Laevo: A Temporal Desktop Interface for Integrated KnowledgeWork," in *Proc. UIST*, pp. 679–688, 2014.
- [20] S. Houben, P. Tell, and J. E. Bardram, "ActivitySpace: Managing Device Ecologies in an Activity-Centric Configuration Space," in *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces - ITS '14*, pp. 119–128, 2014.
- [21] S. Houben, J. E. Bardram, J. Vermeulen, K. Luyten, and K. Coninx, "Activity-centric support for ad hoc knowledge work: A case study of co-activity manager," in *Proc. CHI*, p. 2263, 2013.
- [22] S. Houben, S. r. Nielsen, M. Esbensen, and J. E. Bardram, "NooSphere: An Activity-Centric Infrastructure for Distributed Interaction," in *Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia - MUM '13*, pp. 1–10, 2013.