Enriching In-IDE Process Information with Fine-Grained Source Code History

Sebastian Proksch†, Sarah Nadi*, Sven Amann†, Mira Mezini†
Technische Universität Darmstadt, Germany†, University of Alberta, Canada*
{proksch,amann,mezini}@cs.tu-darmstadt.de, nadi@ualberta.ca

***Abstract*—Current studies on software development either focus on the change history of source code from version-control systems or on an analysis of simplistic in-IDE events without context information. Each of these approaches contains valuable information that is unavailable in the other case. Our work proposes *enriched event streams*, a solution that combines the best of both worlds and provides a holistic view on the software development process. Enriched event streams not only capture developer activities in the IDE, but also specialized context information, such as source-code snapshots for change events. To enable the storage of such code snapshots in an analyzable format, we introduce a new intermediate representation called *Simplified Syntax Trees* (SSTs) and build CAIRET, a platform that offers reusable components to conveniently work with enriched event streams. We implement FEEDBAG++, an instrumentation for Visual Studio that collects enriched event streams with code snapshots in the form of SSTs. We share a dataset of enriched event streams captured from 58 users and representing 915 days of work. Additionally, to demonstrate usefulness, we present three research applications that have already made use of CAIRET and FEEDBAG++.**

I. Introduction

To assist developers in their everyday work, an understanding of developers' activities is necessary, especially how they develop source code. Ideally, this can be done by observing developers in their actual work environment. Several researchers have conducted such observation studies to, for example, assess developer productivity [35], evaluate the user interface of particular tools [43], or study software evolution [11]. Unfortunately, conducting such experiments is very expensive and researchers typically resort to other alternatives.

Two popular alternatives are studying historic development information *after the fact* through the artifacts that are created during the development process (e.g., source code files) and analyzing developer interaction data that was automatically captured while using an Integrated Development Environment (IDE). Examples of the former include using historic data to help developers with change summarization [44], bug localization [10], deriving related files based on their co-change probability [67], and predicting change locations [71]. Examples of the latter include analyzing developer interaction data to recommend artifacts related to the developer's current activities [26], improving developers' productivity [27], and understanding how developers spend their time [39]. We argue that considering historic data and in-IDE interaction data separately misses out on the bigger picture that is necessary to gain a complete understanding of software development.

Analyzing changes committed to public version-control systems (VCS) is the most common way of analyzing software artifacts after the fact (i.e., historic information) and is an important topic in the area of mining software repositories [20]. Some of the approaches in this area work on the level of changed lines [54], others are based on syntactic information [10], and some need fully-resolved types in the sources [71], meaning that the code has to be compilable first. With the rising number of open-source projects hosted on platforms such as GitHub or SourceForge, such data has become easily accessible. However, only looking at the artifacts developers create has the inherent problem that any conclusions drawn are affected by the granularity of the collected artifacts. Many intermediate modifications that are subsequently reverted or again modified will not be observed in such data. A previous study by Negara et al. [46] shows that the history from VCS is not representative of the actual code evolution.

To overcome some of the disadvantages of using VCS data, some researchers refine the granularity of the snapshots by auto-committing intermediate code versions from the IDE to a VCS whenever the developer makes a change in the editor [46], [59] or saves a file [62]. This approach creates a very fine-grained history of commits, which allows a closer inspection of the steps taken by the developer. Finer-grained information provides more insights about how developers write code, enabling us to produce better tools to support them. However, source code snapshots leave behind other relevant information that is available in the developer's IDE. One example is resolved type information, which is important for any static analysis on the snapshots. While it is usually available in the IDE, it can only be reconstructed from a captured code snapshot by compiling the snapshot. This is typically a hard task, as dependencies might be unavailable or compilation might rely on a project-specific environment. Another example is the edit location of the developer, which is known in the IDE, but can only be approximated from a historic change set. The more files involved and the more coarse-grained the history is, the fuzzier this approximation becomes.

Capturing only source code snapshots also leaves behind any information about the development process. The snapshots themselves do not explain how the current piece of code was developed, which might include usages of refactoring tools or code-completion engines. Previous work recovers this information using heuristics [16], [21], but it cannot generally be reconstructed precisely.

Lack of context may lead to missing or incomplete conclusions when studying developers and the code they produce. We argue that *capturing richer and more detailed context information is needed to facilitate the reasoning about captured code snapshots and to get more insights into the development process*. Previous studies have tracked developers' activities within an IDE to capture information about the development process, usually in the form of an event stream (e.g., [3], [26], [39], [60]). However, most of these studies only record command invocations without specific details about them (e.g., they capture that a specific refactoring was invoked, but not to which part of the code it was applied). This makes it hard to interpret the event stream and to align it with source code changes. To the best of our knowledge, only few studies combine the analysis of fine-grained source code evolution with in-IDE process and tool information (e.g., [6], [11], [62]).

In this paper, we propose an approach to create development artifacts that combine process information with source changes. We capture an *enriched event stream* of development activities in the IDE that stores not only simple information about executed *commands*, but also *context* information to enrich the usefulness of these events. This provides a holistic picture of the developer's work and makes it possible, even after the fact, to answer questions that touch both source-code changes and information about the development process, e.g., "*Which files or method implementations did the developer look at while working with a specific API*?", "*What is the effect of using various history granularity levels when analyzing developer behavior, e.g., time-based, commit-based, or activity-based intervals?*", or "*Which parts of the code get changed after a failing test?*"

To realize our proposed approach, this paper makes the following main contributions:

- Enriched event streams, a *representation of the in-IDE development process* that is enriched with specialized context information.
- Simplified Syntax Trees (SSTs), an *intermediate representation for source code* catered to capture in-IDE code snapshots.
- FeedBaG++, a general and extensible *interaction tracker* that captures enriched event streams in Visual Studio.
- C[A]RET, a *platform* that provides tooling around enriched event streams and SSTs.
- A *dataset* of enriched event streams collected from 58 developers.

All tools, documentation, examples, and the dataset can be found on our artifact page [4].

II. Overview

In this paper, we introduce C[A]RET, a platform that enables researchers to work with a fine-grained change history and connect it to the development process. Figure 1 shows an overview of our infrastructure, including the tools C[A]RET and FeedBaG++ and the data structures we built to enable this connection. We now briefly explain the figure and use it as an outline to explain the structure of the paper.

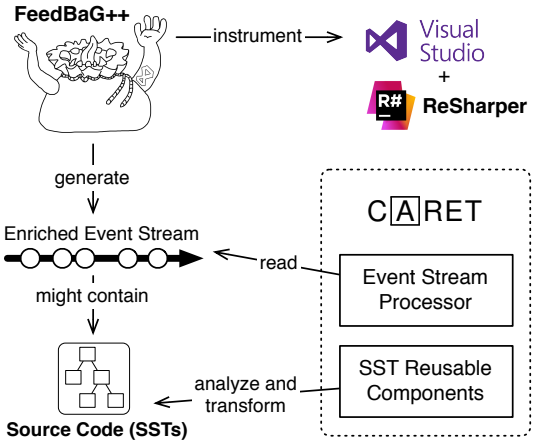

Figure 1: Overview of our Infrastructure

The top-left corner of Figure 1 shows FeedBaG++, our interaction tracker for Visual Studio. FeedBaG++ captures all the actions developers perform in the IDE. It then generates an *enriched event stream* that not only captures which events take place in the IDE, but also stores relevant context information about them. Once an enriched event stream is captured, it can be processed using C[A]RET's *Event Stream Processor*. C[A]RET is the tooling we built to work with our new representation of enriched event streams. The abbreviation stands for *Composable Analyses and REusable Transformations*. The event stream processor allows reading the detailed information of each event. Section III introduces enriched event streams, the individual event types we capture, and our tool, FeedBaG++, which we use to capture them from within Visual Studio.

One particular event we capture is a change event, which we enrich with a snapshot of the source code under edit. We store these snapshots in the form of *Simplified Syntax Trees*, an intermediate representation (IR) we developed for this purpose. Thus, as shown in the bottom left of Figure 1, an event in the enriched event stream may contain an SST. To analyze the code represented by this SST, C[A]RET provides *SST Reusable Components*, such as visitors for traversing the tree and a points-to analysis for reasoning about the code. Section IV describes SSTs, how we use them to capture fine-grained source code changes, and the tooling we built to analyze them.

Overall, C[A]RET provides a stable basis for research on and with a fine-grained in-IDE development history. It can be used to analyze new data collected by installing our in-IDE interaction tracker, FeedBaG++, in the researcher's chosen setting or by using our pre-collected dataset of in-IDE development events. We discuss this dataset in Section V as part of the applications we already used our infrastructure for.

III. Capturing Enriched Event Streams

Our goal is to capture developers' interactions with the IDE and to combine this data with richer and more detailed context information. This allows getting more insights into the development process and facilitating the reasoning about captured code snapshots. In this section, we discuss how we capture enriched event streams from within the IDE. We also

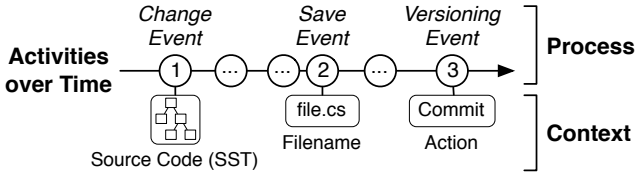

Figure 2: Enriched Event Stream

describe the events we currently capture and how additional events and context information can be added in the future.

A. Representation and In-IDE Tracker

Figure 2 shows how we represent enriched event streams, which consists of two levels: *process* and *context*. On the process level, we capture a stream of individual events with basic information, such as the event's *type* (e.g., code change or save), its *invocation time*, and its *duration*. On the context level, we store specific data depending on the event type. For example, *save events* store the name of the file that was saved and the way in which the save action was invoked (e.g., by short cut or by menu selection).

To capture enriched event streams, we built FeedBaG++ for Visual Studio on top of our previously published tool, FeedBaG [2], [3], which only captured command invocations. FeedBaG++ is a plugin for ReSharper, a very common extension to Visual Studio that adds many convenient development tools and analyses. For simplicity, when we talk about Visual Studio in the rest of the paper, we always mean the combination of Visual Studio and ReSharper, because FeedBaG++ needs ReSharper to work. FeedBaG++ is available for installation from the ReSharper extension gallery, such that anyone can use it.

Once installed, FeedBaG++ runs transparently in the background and requires no interaction. It monitors developers' actions and captures the following types of enriched events: command invocations, mouse movements and clicks, ctrl-click navigations, usages of the search tool, changes (e.g., edit and code completion), document actions (e.g., create, open, and save), project builds, test runs, debugger usage, version-control usage, IDE-state changes (e.g., startup and quit), window events (e.g., move), and system events (e.g., screen lock and suspension).

For most events, the context captures the target on which the event was invoked, such as the name of the file that was saved, information about projects that were built, or the Id of the command that was invoked. For change events, the context also captures a code snapshot of the file-under-edit. These code snapshots are not simply stored as plain text, as this would make them very hard to analyze outside of the IDE environment. Instead, we use a specialized intermediate representation called *Simplified Syntax Trees* (SSTs) that preserves typing information and stores additional information, such as the edit location and details about code-completion interaction. We provide detailed information about SSTs in Section IV.

The event data is collected locally and developers are asked to regularly upload the collected data to a server, whose URL they can configure themselves. After the upload, our tooling consolidates uploads from the same developer and cleans up noisy data, e.g., by removing duplicated uploads.

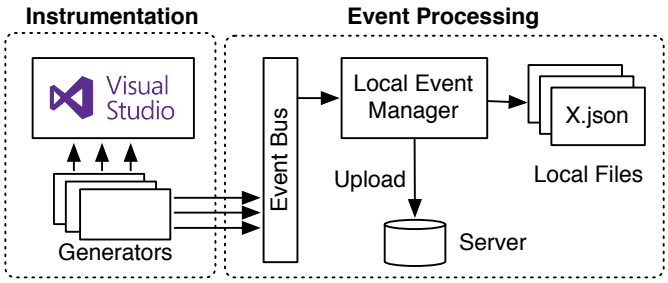

Figure 3: Workflow of Event Generation in FeedBaG++

FeedBaG++ and all post-processing tools are open source and can be used to collect data in controlled experiments, field studies, or data collections for specific target populations.

B. Extending the Enriched Event Stream

While we already capture a wide range of events from the IDE, researchers may be interested in additional events or additional context data. To explain how this can be done, we illustrate the workflow of creating and managing the event stream in FeedBaG++ in Figure 3. This workflow consists of two parts: the instrumentation and the event processing.

The *instrumentation* of the IDE represents the part in which the actual events are captured. Visual Studio and ReSharper both offer APIs that can be used to access information about the current state of the IDE. The researcher performing the instrumentation needs to identify the information she is looking for and, accordingly, use the relevant parts of the APIs.

To allow FeedBaG++ to receive and process any collected data from the instrumentation, it is necessary to implement a new *generator* that is automatically instantiated every time Visual Studio is launched. The first step is to register the generator as a listener for relevant activities (e.g., menu clicks) and react accordingly when they occur. As a registered listener, the generator can use the APIs of Visual Studio, ReSharper, or any other API available in Visual Studio's runtime that offers relevant information. In a final step, an event has to be initialized that can store the captured information. The event is then published on the *Event Bus* for further processing.

The *event processing* part of FeedBaG++ is responsible for the machine-local management of the enriched event stream. More specifically, it is responsible for the local storage and the upload of the collected event stream. The interface from the instrumentation to the event processing is the event bus. The *local event manager* consumes all events that are published on this bus. Each incoming event is serialized to JSON and stored in files on the local drive of the developer's machine. The local event manager also provides the interface for FeedBaG++'s end-user, which allows users to review the collected data and regularly reminds them to upload it.

While collecting new data requires updating the instrumentation, there is no need to update the event processing part to handle newly created event types in the enriched event stream, as long as they extend the event base class.

Given the above workflow and design, the event stream can be extended in two ways: First, it is possible to add new event types that might define their own specific context. Second, the context of existing event types can be extended with additional specific information. To illustrate both ways, we discuss how we added an extension to FeedBaG++ that captures navigation information. More specifically, how we extended FeedBaG++ to capture data about when developers use *control-click* to navigate the code base.

The first step was to create a new event type and adding a generator that intercepts any control-clicks in the editor window. Each time such a navigation takes place, the generator instantiates an event, fills in the basic information (e.g., the time), and publishes the event. Capturing this event type allowed us to analyze how often this kind of navigation is used in practice. We soon realized that, to make sense of the data, we needed more context information about the navigation. Since we already had the event type in place, we addressed this by extending the existing generator to capture the additional data, namely the *fully-qualified name* of both the current location and the target location, and by adding this specific context information to the event type.

IV. Capturing Fine-Grained Source Code Changes

As part of our enriched event stream, we want to capture fine-grained code evolution to enable the answering of many related research questions. A simple textual snapshot of a source code file poses many analysis challenges. It is necessary to reconstruct the project environment to make it compile and, for example, dependencies or project specific configuration files might be missing. Therefore, we designed a new intermediate representation (IR) for source code called *Simplified Syntax Trees* (SSTs) to facilitate static analyses and the creation of source-based tools. We first describe five design considerations for such a representation (and the tooling around it) and then present the implementation. The design considerations are motivated by needs of existing source-based research techniques and by potential uses we foresee. The last part discusses our implemented transformation for C# code.

A. Design Considerations

(DC1) Resolve typing information: Similar to analyzing source code from commit history, current techniques for creating in-IDE source code snapshots have the problem of not capturing typing information. This is because only text-based snapshots of the code are captured. The typing information can be reconstructed if the sources can be compiled. However, compiling a project is a hard task as projects include dependencies that might be unavailable or because they rely on a project-specific build environment. We, therefore, advocate that captured source-code snapshots should contain resolved typing information to facilitate analyses done on top of them and to save their users the effort of resolving dependencies.

(DC2) Capture the names and versions of dependencies: It is important to consider the used libraries and frameworks when studying how developers write code. Such libraries often get upgraded to a newer version or exchanged with an alternative better suited for the task. To detect this kind of change, it is crucial to capture the list of included dependencies as well as changes to how they are used in the code.

Modern provisioning frameworks such as NuGet, Maven, or P2 make it easy to identify the required dependencies and to extract the respective name and version. However, simply storing the dependencies of a project (e.g., in a VCS) does not help in understanding the impact of different versions for recommending the right code (changes) to the developer. While the change of project dependencies suggests a library migration, it does not indicate anything about the required code changes.

We argue that the library references should be stored together with the typing information in a code snapshot to make the relationship between types and libraries explicit. In this way, it is possible to identify code elements that point to the old framework and, in a later snapshot, code elements that point to the new framework.

(DC3) Contain edit location: To get a holistic view on source-code changes, they have to be aligned with tool interactions in the IDE. For example, consider the case in which subsequent snapshots indicate a variable renaming. To study the evolution, it is interesting to distinguish cases in which the developer used a refactoring tool from cases in which the renaming was done manually. Another example is understanding how developers use a code-recommender system that proposes the next method to invoke, which was one of our early motivations for this work. Enriched event streams contain this information, but in both cases, it would be very useful to know the location in the source code in which the tool was invoked, a piece of information that is readily available in the IDE. We argue that a code representation should go beyond the syntactical content of a source file and also include information that makes it easy to align source code and the process.

(DC4) Handle incomplete code: As opposed to code in VCSs, in-IDE code snapshots suffer from the problem that source code under edit is often incomplete or contains invalid parts. Thus, when creating the representation of the code snapshot, these parts must be handled. We argue that this burden should not be left to the researchers who only work with the snapshot, because this issue is easier to solve in-IDE, where all surrounding information is still available.

(DC5) Capture stability indicators for changes: One potential problem with capturing code snapshots from the IDE is that the granularity might be too fine to find meaningful patterns, because the changes are no longer grouped. For example, assume the developer continues changing the code for one minute until it gets to a stable state and then she invokes the save command. The snapshots taken during this minute might have a different flavor than the one taken at the save trigger, since the latter might indicate slightly more stable code. We argue that having some basic versioning indicators such as *on change*, *on save*, or *on commit* correlated with the code snapshot can allow different groupings of snapshots and the analysis of the data at different levels of granularity, depending on the intention and heuristics used.

B. Simplified Syntax Trees (SSTs)

It is not possible to simply store all available information about the development context in each code snapshot, because this would include all sources, dependencies, and configuration files of the project. In addition, directly analyzing source code has certain challenges, such as implicit information in the *Abstract Syntax Tree* (AST), e.g., implicit `this` references, or arbitrary complexity of the syntax tree (e.g., nesting or chaining of expressions). To make a data collection of such fine-grained changes feasible and to ease future analyses built on top of it, we design a new intermediate representation for the snapshots.

There are two main styles of representing source code for later static analyses that we considered: ASTs and 3-address representations. In the former, the representation is close to source code and best reflects the view of the developer. In the latter, it is easier to write static analyses, because the complexity of the language is reduced to simple register operations and jumps with labels. We decided to combine the advantages of both styles with the following goals in mind:

- Improve the analyzability, when compared to source code.
- Preserve control structures (stay close to source code).
- Avoid implicit information (make everything explicit).

As a result, we defined SSTs, a new tree-based intermediate representation for source code. An SST is a tree-based structure that is very close to the AST of the original source code. It represents the current type under edit and still contains most syntactic elements found in the source code, such as declarations, invocations, or control structures. To reduce the size of the representation, there was a tradeoff between capturing everything and having only the commonly used information. We decided to leave out some information, such as white space and comments, that is typically not used when reasoning about the source code.

In addition to the syntactical information, we also store information from the type system: references and hierarchy.

References. Each reference to the language model (e.g., a return type of a method) is stored in a *fully-qualified* form that, in addition to the namespace-qualified type, also contains the originating framework and its version (the *assembly* in C# terminology). An example of such an assembly-qualified name is "`a.b.C, foo.dll, 1.0`", where `a.b.C` is the type's fully-qualified name, `foo.dll` is the assembly it belongs to (e.g., a library or framework), and `1.0` is the version of the assembly. Project-local types do not have a version, so we simply omit it, as in "`x.y.Z, MyProject`". The versioning information may be used to study the evolving use of certain libraries and addresses design considerations DC1 and DC2.

The naming scheme also captures *generics*. For example, "`A`1[[T→B, b.dll,1.0]], a.dll,2.0`" points to a generic type `A` (from "`a.dll,2.0`") with one (`` `1 ``) generic parameter `T` that is bound to `B` (from "`b.dll,1.0`").

When transforming source code to SSTs, we replace every explicit or implicit type reference by the corresponding fully-qualified name. This means that we also store the respective fully-qualified names for the declaring type, any parameter types, and the value or return type of every reference expression.

Hierarchy. We store information from the type system about the edited type, i.e., the type hierarchy of the type under edit. We also store information about all its method declarations, i.e., which method declarations are entry points, which ones override an existing declaration, or which implement an interface. Capturing this information is motivated by the design consideration DC1.

One of the things we were interested in early on is understanding how developers use code completion tools. Therefore, we added the new expression type in addition to the regular code elements defined in the C# language. We use this expression type as a marker of the edit location and to store information about code completion events in this expression, which addresses design consideration DC3. For example, we store the object reference on which the code completion was triggered. When an SST is created without having the developer edit the file (e.g., by transforming a piece of code outside of the IDE), this additional element is never included, because no interaction takes place.

Including this information as a language element ensures consistent source code transformations both inside and outside of the IDE and saves the effort of maintaining two variants. For example, this makes it easy to integrate any recommender system that relies on source code into the IDE, because it can count on receiving the same input whether it is working on in-IDE source code or source code from repositories.

The remaining design considerations are not addressed by the representation, but by the tools we have built around it. Design consideration DC4 ("incomplete code") is addressed by building a resilient transformation. Design consideration DC5 ("versioning indicators") is addressed by capturing process information about saves and version control in addition to the change events.

We note that we have previously published a data paper that made use of SSTs for gathering a dataset of C# code [51]. However, the data paper focused on the creation of the dataset, its representation, and potential use cases and did not focus on the in-IDE snapshots or process information. We did not elaborate the requirements for the representation before; the data paper only discusses the SST grammar.

C. C# Transformation

We implemented a transformation for C# that transforms source code to SSTs. It is written in C# and is based on the ReSharper framework [56]. The transformation can generate SSTs from the source code of Visual Studio solutions in a batch job. The same transformation is available as a service for ReSharper plugins that run in Visual Studio, which, for example, allows FeedBaG++ [13] to capture the source code under edit from Visual Studio's editor. Since source code in an editor may be non-compilable, the transformation is resilient against missing statements or incorrect expressions. It processes all information that ReSharper's parser is able to retrieve from the code (see DC4).

```
1 o.A();
2
3 if (o.IsX()) {
4
5
6     o.B(C()).D();
7     var e = E();
8     e.t□
9 }
```

(a) Original C# Source Code

```
1 o.A();
2 bool $0 = o.IsX();
3 if ($0) {
4     int $2 = this.C();
5     string $1 = o.B($2);
6     $1.D();
7     int e = this.E();
8     completion(e, t)
9 }
```

(b) CARET's IR (SST)

Figure 4: Transformation of a Source Code Example

The transformation is influenced by two design decisions:

- Simplify the representation.
- Unify over alternative styles of writing.

We implemented several transformation steps to follow these principles. The transformation simplifies nested expressions that are more complex than literals or reference expressions by creating artificial intermediate variables to which the formerly nested expressions are assigned. The transformation removes syntactic sugar from the source code and replaces it with the desugared variant. For example, C# allows to initialize properties directly as part of a constructor call. Our transformation reduces this initialization to a constructor call and assignments.

An example of our transformation is provided in Figure 4. Figure 4a shows a piece of C# code and Figure 4b shows the corresponding SST (when rendered to a source-code like representation that omits details like the fully-qualified type references). The latter makes implicit information visible. For example, the `var` keyword on Line 7 of the original code is replaced by a type reference in the SST and all `this` references are explicitly named. Furthermore, the SST unifies over alternative styles of writing. For example, the nested and chained calls in Line 6 of the original source code are assigned to intermediate variables in Lines 4 to 6 in the SST.

D. CARET's SST Tools

As part of CARET, we offer various tools that enable other researchers to work with SSTs. Figure 5 shows these tools. SSTs are created by the C# transformation component that was presented in the previous section. The component can be used to capture SSTs from files under edit in Visual Studio or to transform a given C# solution to SSTs. It is implemented in C# and is part of FeedBaG++. However, since many research tools are written in Java, we provide our reusable tooling for the analysis and manipulation of SSTs in Java. In the following, we will describe these components.

Visitor: SSTs implement the visitor pattern [14]. The goal of the pattern is to separate an algorithm from the data structure on which it operates. This means that new static analyses can be implemented on top of SSTs without altering the data structure. The visitor pattern also makes traversing the SST tree structure very convenient.

Inlining: It is a common guideline and a good coding practice to keep method bodies short and to identify coherent building blocks that can be outsourced to a helper function [32].

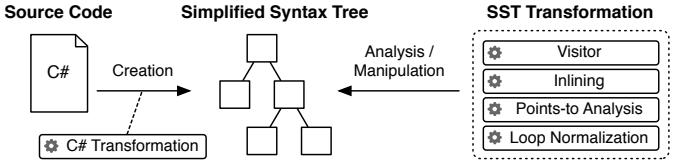

Figure 5: Reusable Tools for Simplified Syntax Trees

Unfortunately, from the point of view of a researcher that tries to identify related method calls, this makes it harder to write a static analysis. We provide an inlining component that inlines calls to private helper methods (non-entrypoints) into the calling method; only the entry points of an SST remain in the end. This allows restricting a static analysis to an intra-procedural scope, but still get some benefits of inter-procedural analyses.

Points-to Analysis: When studying source code, it is often the case that a researcher is interested in the objects that are referenced or addressed in a specific statement. We provide points-to analyses for SSTs that can be used to identify the abstract location to which variable references point to during the execution of a program. We have four implementations: two simple implementations that are based on types or reference names, a Steensgard-style unification analysis [63], and an implementation that is based on constraint inclusion [58].

Loop Normalization: Many code elements can be used to express a loop in a program, e.g., `while`, `for`, `foreach` etc. Depending on their preference or knowledge of the programming language, different developers might select different loop constructs when implementing the same piece of code. This makes it harder, for example, to find patterns in the resulting code, because the code structure is less similar. We provide a reusable component that normalizes all loop constructs into a `while` loop, with the goal of unifying source code written by different developers.

V. Real Applications of FeedBaG++ and CARET

To demonstrate the usefulness of the tools and data structures we built, we describe three research directions that we already used them in. We also indicate potential future applications.

A. Collecting In-IDE Development Events

To make use of CARET, it is necessary to capture data that can be analyzed. We published FeedBaG++ in the official plugin repository of ReSharper [13]. We advertised the project in our social media channels and at various conferences. Our invitation to use CARET was open to any interested developer, which means we did not target a specific population of users. However, we added a voluntary questionnaire to learn about participants' backgrounds. Our assumption is that a high number of random participants and long observation times provide a rich data set with a variety of projects and participant backgrounds.

So far, we have received 5.6 millions events that have been uploaded by 58 developers. Out of these developers, 30 come from industry, *two* are researchers, *four* are students, and *six* are hobby programmers. *Sixteen* participants did not provide the (optional) profile information. The submissions cover 915 days and span *six* months, but not all developers participated

the whole time. On average, each developer provided ~ 97,000 events (median ~ 42,000) that have been collected over 15.8 days (median 9.0) The dataset and more detailed statistics are available on our artifact page [4] and can be used by others.

Note that we previously described a different interaction dataset [3] that we collected using FeedBaG [2], the predecessor of FeedBaG++. This previous dataset is different from the one we describe here in the following ways: First, it was created with FeedBaG, which did not yet capture enriched event streams, the main contribution of this paper. Second, it was based on a deployment with an industry partner, which prevented us from publishing the dataset.

Other researchers can easily reuse FeedBaG++ and CARET. The server to which the data is uploaded can be configured in the tool. As FeedBaG++ and the optional server that aggregates the uploads are open-source, researchers can extend FeedBaG++, if necessary, and collect their own dataset.

It can be used in different settings. For example, we decided to collect the reference dataset in a field-study-like setup. In a more controlled setup, FeedBaG++ could also be used to collect data from the subjects and then be complemented with qualitative data collected during or after the experiment.

B. Learning How Developers Test

Previous work by Beller et al. [6] analyzed how Java developers test. They applied WatchDog in the Java IDEs Eclipse and IntelliJ. The enriched event streams captured by FeedBaG++ provide an opportunity to extend their work to Visual Studio.

The experiments in their paper were based on intervals of several activities (i.e., IDE open, active periods of the developer, reading and typing in a file, test execution). By default, FeedBaG++ captures any commands initiated by the developer so all of these activities were already included in our event stream. However, for test executions, we only captured that an execution was initiated, but no further details about the individual tests. We added an additional generator to our platform that instruments the test runner of ReSharper to fill this gap. It captures the names of each executed test, as well as the duration and the result of the run. We then designed a *test event* data structure to store the relevant information.

A technical difference between both tools is that we capture (and upload) a fine-grained event stream, whereas WatchDog lifts this stream to intervals on the client side and only uploads the resulting intervals. An interval captures when and for how long an activity took place. We implemented an offline conversion in CARET from enriched event streams to the intervals described in their paper to make our enriched event stream compatible with WatchDog. The original authors confirmed that the created intervals are sufficient to run the experiments in their pipeline.

While this project is still an on-going collaboration with the WatchDog team, it provides an indication of the research possibilities that enriched event streams open up. It shows that having FeedBaG++ made it easy to extend WatchDog to a new IDE. It also shows that enriched event streams already contain a wide range of context information and that new generators can be added to capture more.

C. Creating Static Analyses

One of the core principles we followed when designing SSTs was to allow the ability of building extensive static analyses on top of them and to support some of these re-occurring analysis tasks with reusable components. As an early use case, we re-implemented the static analysis of a method call recommendation system we had previously built [53], which was built for Java Bytecode using Wala [66]. We replicated the recommender for C# with the goal of checking if SSTs contain all required information for this specific static analysis.

The recommender approach relies on *object usages* as input, a data structure that stores all the context information relating to a single *object* in the current enclosing method. For example, it stores the surrounding method definition as the *method context*, how the object was initialized (e.g., passed as a parameter or by a constructor call), which methods were invoked on the object, and the method invocations to which the object was passed to as a parameter. Objects are also tracked into private helper methods to make the method context more meaningful.

The revised static analysis was implemented as a visitor on SSTs. We made use of the points-to analysis provided by CARET to identify potential object instances and collected the required information while traversing the syntax tree.

During the design of SSTs, we considered other source-based recommender systems such as other method call recommenders (i.e., [1], [7], [22], [34], [55], [68]), snippet recommender (i.e., [23], [31], [47], [48], [64]), and tools for code search (i.e., [24], [25], [41], [69]), documentation (i.e., [33], [36], [37], [70]), and anomaly detection (i.e., [30], [40], [50]). We have documented these requirements in an online appendix that is available on the artifact page [4] such that other researchers can check the data captured in SSTs and decide if it is applicable to their research questions.

D. Designing More-Realistic Evaluations

Since a realistic ground truth for evaluating a code-recommendation system is usually not readily available, evaluations of such systems typically rely on creating artificial queries from stable code found in VCSs (e.g., by removing one or more method calls). In the dataset we created, CARET provided us with a fine-grained change history of source code from several developers. This allowed us to understand how the source code evolved over time and which method completions helped the developer to reach the final state. Having this kind of information in an easily processable way allowed us to use it as the ground truth to evaluate a method-call recommender we built before (see Section V-C) and understand how artificial evaluations compare with more realistic settings [52]. Both the final code state as well as the intermediate states to create queries from where taken from the enriched event stream.

We compared an artificial evaluation that would be conducted if only the final state would be available with a realistic evaluation that is based on the fine-grained source code evolution. To train the recommender, we needed to collect large amounts of existing API usages. Therefore, we extracted a dataset of SSTs from 360 GitHub repositories containing

C# source code [51]. The intention was to complement the interaction data and to select projects that contain examples of the same APIs that appear in the collected interaction data. The transformation from C# code to SSTs was done with CARET's C# transformation introduced in Section IV-C.

Our experiments showed that evolving code contains information that is changed or removed before the source code is committed. If this evolving context is included in a realistic evaluation, it has a big impact on the recommendation system. We concluded that an evaluation that ignores the fine-grained history would report a much higher quality for the recommender than what would be achieved in practice [52].

E. Discussion

The above applications demonstrate the practical use of the enriched event stream and SSTs, its intermediate representation for source code snapshots. We have shown the potential and the applicability of the reusable components, as well as the usefulness of the fine-grained change history, including the additional information stored in the SST, i.e., the edit location and the typing information. We also demonstrated the extensibility of the generator framework for the event stream.

These directions only represent our experience and the applications we have used CARET for so far. However, the availability of the novel event stream of fine-grained developer actions opens up new research directions. We foresee many further uses of the current event stream, such as learning more about debugging activities of developers, analyzing navigation behavior, or understanding more specific details of code evolution. Even though we only replicated the static analysis of a single recommendation engine, we also foresee that SSTs are applicable for various recommendation systems in software engineering, such as snippet recommenders or code search. The fine-grained evolution of the source code might also be a valuable source for the evaluation of such systems.

We encourage other researchers to contribute generators to FeedBaG++ to further enrich the event stream with more specialized event types. Our continuous advertising leads to an increasing user base and a growing dataset that enables investigations of further research questions at a larger scale.

VI. Limitations of CARET

Empirical research on developers is a complex topic. FeedBaG++ focuses on capturing data from the developer while working in-IDE. Depending on the research question, a more holistic picture of the development process is required. For example, studies in psychology or applied social studies might need information about the sentiment of the developer or any utterances they express in the experiment. Another example are studies that involve out-of-IDE tools (e.g., the command line or websites) or data (e.g., biometric information). We do not currently capture such data, but we designed FeedBaG++ to be extensible, such that any event source could potentially be combined with the in-IDE event stream

FeedBaG++ and the SST transformation are currently only available for Visual Studio and the C# programming language. However, the concepts presented in this paper are generic enough to be applied to other programming languages and respective IDEs. The tooling provided in CARET is language and IDE independent, offering reuse opportunities for studies across language and IDE boundaries.

Regarding the in-IDE collection of developer activities, we group the current limitations of our infrastructure into three categories: limitations of the process representation, the SST representation, and the SST transformation.

Process: We introduced two abstraction levels that describe the change process in an IDE: an event stream that describes the development process and an intermediate representation for code snapshots. While both abstraction levels are general and extensible for future requirements, it might be that a conceptually different style of representing the historic information is required for a specific approach, or that we abstract certain needed information. For example, we capture the events *on change*, *on save*, and *on commit* in our enriched event stream to indicate different versioning information. However, some approaches also use the actual commit message to derive further information, for example to classify the change type [54]. Since commit messages were not central to our research, we do not currently support them. However, they can be added in the future by extending the corresponding VCS generator.

Simplified Syntax Trees: SSTs represent source code in a normalized form by avoiding the nesting or chaining of expressions. This design decision was driven by the desire to simplify analysis tasks. At the same time, this design makes it harder for approaches that work with source code as plain text. We mark the artificial intermediate variables, so inverting the normalization is possible. In future work, we will consider providing the normalization as a reusable component to better separate the snapshot creation from the SST transformation.

SSTs store typing information, but it is impossible to store the entire language model in every snapshot. Therefore, we restrict the captured information to the immediately relevant parts, even though we might miss information that is required by a specific approach. For example, a snapshot does not include pointers to all methods that are overridden by a method declaration, but only to the super method and to the first declaration that introduced the specific signature. We argue that the effect of this tradeoff is rather small. The most valuable information from the type system are references to reusable APIs, whereas project-specific references do not carry any reusable information. These references to public APIs can still be resolved after the fact. For example, by building an index of typing information extracted from published libraries and frameworks. Since SSTs contain fully-qualified type information, it is possible to query this index for missing information. This still means that project-specific information is lost, if the project is not published as open-source. However, since researchers are usually interested in information regarding reusable APIs, this solution presents a reasonable tradeoff.

Even though our abstraction leaves out information from the source code, e.g., comments or attributes, this does not

limit the generality of the approach. Adding support for such information is a matter of extending the SST grammar and spending some engineering effort to extend the transformation.

Transformation: In its current form, the transformation of C# source code changes the semantics of short-circuit evaluation. This is because all sub expressions in a boolean expression are assigned to intermediate variables and, therefore, always evaluated. We acknowledge that this transformation is unsound, but it only affects specific analyses that rely on this information such as data-flow analyses that detect missing null checks, for example. In addition, this limitation can be easily addressed by spending additional engineering effort and does not represent a limitation of concept or design.

VII. Related Work

In this section, we discuss four areas that are closely related to the concepts presented in this paper: (A) intermediate representations of programs, (B) platforms that support reusable analyses of source code, (C) tools that collect snapshots of code, and (D) in-IDE interaction trackers.

We would also like to note that there is a large research direction that integrates information from additional data sources, such as mailing lists and issue trackers (e.g., [9], [15], [19]), to analyze software evolution or software metrics. While the focus of CAReT is on source code and process information, such line of work is orthogonal to ours. Another interesting line of research captures biometric information from the developer, e.g., which code fragments they looked at [57]. Such information could be used to further enrich our stream of events and SSTs. Opportunities and challenges of integrating both directions into CAReT should be investigated in the future.

A. Intermediate Representations

Various IRs of source code have been proposed to facilitate different kinds of analyses. We subsequently discuss these IRs and compare them to SSTs.

Gomez et al. [17] introduced Ring, a unified meta model for Smalltalk that provides extensions to model changes and history. Their model resembles a simple AST-like representation that does not resolve typing information, while our SST representation captures fully-qualified type identifiers.

Necula et al. [45] introduced the C Intermediate Language (CIL) as an IR for C code. Similar to ours, their transformation from C to CIL resolves ambiguities in the source language and ensures unique type names by moving all type declarations to the top level and disambiguating their names. Contrary to their approach, our transformation also deals with types declared external to the program being transformed.

Proteus [65] converts source code into an IR called Literal-Layout AST (LL-AST). The approach focuses on the problem of preserving document layout, i.e., comments and formatting, in automated code transformations. It deals with many C/C++ specific problems, such as preprocessors and macros. In contrast to LL-ASTs, SSTs contain resolved type information, but exclude comments and formatting. We argue that, while both need to be preserved when modifying source code, they are mostly irrelevant to investigations on how developers work.

JavaML [5] is an XML-based IR for source code. The transformation to JavaML inserts tags around source elements to identify the element's nature, e.g., whether it is a keyword, a type name, or an identifier. This allows analysis and transformation of source code using standard XML tools, such as XQuery or LINQ, instead of custom programmatic processing of ASTs. srcML [8] is an infrastructure for the exploration, analysis, and manipulation of source code. It generalizes the idea of JavaML by abstracting over programming languages. The format and transformation handles C, Cpp, C++, C#, and Java. Similar to JavaML, the transformation annotates source elements by XML tags, but leaves the source code otherwise unchanged. A big difference between our IR and both JavaML and srcML is that we provide resolved type information to facilitate analysis tasks.

Jimple [29] is a typed three-address representation that was developed as part of the Soot framework to simplify control- and data-flow analyses of JVM Bytecode. Generating a Jimple representation for a Java class requires its .class file, i.e., it must be compilable. In contrast, our transformation deals with partially non-compilable source code, as often encountered when taking snapshots of the IDE editor. SSTs are also closer to source code, which allows analyses to present results in a human-readable form, and makes it independent of the JVM.

Alitheia Core [18], [19] provides a language-agnostic AST-like representation of the source code, which is neither normalized nor contains resolved type information. The SOFAS platform [15] provides a FAMIX meta model of the source code. The model does not provide resolved type information.

In addition to the above points, SSTs encode further information that none of the IRs above capture: First, they encode trigger points of code completion and edit locations. This information is useful for understanding how developers use code recommender systems, how code evolves, and for realistic evaluations of source-based tools. Second, our type information contains API versions, which is useful both from the perspective of studying API use as well as code evolution.

B. Reusable Program Analysis Platforms

RASCAL [28] is a DSL to write analyses and transformations of source code. The core approach is language agnostic, with language-specific constructs being provided as reusable libraries. The language environment provides resolved type information when working directly on the abstract syntax tree. Therefore, it needs compilable sources.

The BOA project [12] provides a domain-specific language and mining infrastructure for analyzing large-scale repositories, such as all Java projects from SourceForge and GitHub. BOA users write their own analyses to query pre-collected datasets. The environment and language are, however, unsuitable for more advanced static source-code analyses, such as pointer analysis. Additionally, type resolution is not supported.

Static analysis frameworks such as Wala [66], Soot [61], and Opal [49] provide reusable modules for common static analysis

tasks. They work on low-level IRs, such as Java Bytecode or Jimple. Since we cannot use these formats as our IR (see Section VII-A), we cannot directly use any of these frameworks. One could, however, convert SSTs to these formats and make use of the rich set of static analyses provided by the frameworks on top of CARET.

C. Snapshots Collectors

Several tools exist that collect fine-grained source code snapshots from the IDE. The closest work to ours is the work by Dias et al. [11]. They introduce EPICEA, which captures code changes of the developers on a structural level together with process information that describes refactorings, test runs, and version control. This makes it quite similar to FEEDBAG++, but EPICEA only captures actions related to source code whereas we capture general interactions. Another difference is that we capture saves, because we consider them as additional versioning indicators. EPICEA stores source code changes incrementally in a log, using the RING format. This is different to *SSTs* in two regards. First, method bodies are only stored as text. They can be parsed to an AST later, but do not contain resolved type information. Second, information is captured incrementally, e.g., in case of a name refactoring, they only store the specific information (i.e., the old and the new name), whereas we would store two complete SST snapshots. While the incremental approach is more convenient for analyses of code evolution, it complicates static analyses that typically require complete pieces of code. In addition, their model makes it necessary to instrument every individual refactoring command to capture the specific information; in our case, we always capture complete snapshots and all invoked commands, so it is possible to extract specific information after the fact.

Schneider et al. [59] introduce an Eclipse plugin that maintains a shadow repository, i.e., a CVS repository to which the user's in-IDE edits are automatically committed. This evolution history was used, together with information from other sources, to investigate team collaboration.

MARMOSET [62] uses an Eclipse plugin to automatically commit a snapshot of the current work to a VCS on each save. It was used to analyze the evolution of student projects in a software engineering course. Marmoset's snapshots are more course-grained than ours, being captured only at the level of saves.

Negara et al. [46] use the Eclipse plugin CODINGTRACKER to track AST operations such as add, delete, and update. From this data, they recover the fine-grained evolution of source code under edit, minus the formatting. They use this information to compute how much of the change history of a project is not captured by regular VCS usage.

All these approaches have in common that they store plain source code. In order to retrieve full typing information from this data, we would need to retrospectively resolve the dependencies of the captured code, which cannot generally be automated. Capturing SSTs directly within the IDE solves this problem, because the developer already sets up the necessary environment for the respective project. Furthermore, none of the approaches captures development process events.

D. Development Event Trackers

Several interaction trackers exist for different IDEs. To the best of our knowledge, they all capture only a subset of the development events we consider in this work and do not capture code snapshots. MYLYN [42] tracks development events in ECLIPSE to identify the project and IDE resources relevant to the current task. DFLOW [38] captures development events in PHARO, an IDE for Smalltalk, to identify high-level activities developers spend their time on. BLAZE [60] tracks which commands a developer invokes in VISUAL STUDIO. Unfortunately, neither the tool nor the dataset is publicly available. WATCHDOG [6] captures testing-related development events and which files developers read or edit in ECLIPSE, but aggregates the events on the client already.

VIII. SUMMARY

Previous work on developer behavior either studies the artifacts that are created during the development process after the fact or observes developers while using an IDE. In this paper, we combine the advantages of both and propose *enriched event streams*, a new data representation that can store both process information and fine-grained context information, including source code snapshots. We propose a new intermediate representation for such snapshots, called Simplified Syntax Trees (SSTs), that solves several challenges that occur when analyzing plain source code. For example, it is unnecessary to compile SSTs, because they already contain type information.

We created FEEDBAG++, an interaction tracker for Visual Studio that captures an enriched event stream and built CARET, a platform that provides reusable components to support researchers in working with the enriched events and SSTs. We deployed the tracker with 58 developers to record their actions in the IDE and share the collected dataset. We described several research questions and directions that we were able to explore using CARET and the rich historic information captured in our dataset. Having enriched event streams paves the road for novel studies on developer behavior. We encourage researchers to use and extend our tools and datasets.

IX. ACKNOWLEDGMENTS

We would like to thank our students Jonas Schlitzer (inlining) and Simon Reuß (points-to) for their invaluable work on the reusable modules and Andreas Bauer for helping with the transformation. The work presented in this paper was partially funded by the German Federal Ministry of Education and Research (BMBF) with grant no. 01IS12054. The authors assume responsibility for the content.

REFERENCES

[1] S. Amann, S. Proksch, and M. Mezini. Method-call Recommendations from Implicit Developer Feedback. In *International Workshop on CrowdSourcing in Software Engineering*. ACM, 2014.

[2] S. Amann, S. Proksch, and S. Nadi. FeedBaG: An Interaction Tracker for Visual Studio. In *International Conference on Program Comprehension*, ICPC'16, 2016.

[3] S. Amann, S. Proksch, S. Nadi, and M. Mezini. A Study of Visual Studio Usage in Practice. In *International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[4] Artifact Page. http://www.st.informatik.tu-darmstadt.de/artifacts/caret/.

[5] G. J. Badros. JavaML: A Markup Language for Java Source Code. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking*, 2000.

[6] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Joint Meeting on Foundations of Software Engineering*, 2015.

[7] M. Bruch, M. Monperrus, and M. Mezini. Learning from Examples to Improve Code Completion Systems. In *Proc. of ESEC/FSE*. ACM, 2009.

[8] M. L. Collard, M. J. Decker, and J. I. Maletic. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *International Conference on Software Maintenance*. IEEE, 2013.

[9] D. Cubranic and G. C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *International Conference on Software Engineering*. IEEE, 2003.

[10] V. Dallmeier and T. Zimmermann. Extraction of Bug Localization Benchmarks from History. In *International Conference on Automated Software Engineering*, ASE'07, 2007.

[11] M. Dias. *Supporting Software Integration Activities with First-Class Code Changes*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, 2015.

[12] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. BOA: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *International Conference on Software Engineering*. IEEE, 2013.

[13] FeedBag++. https://resharper-plugins.jetbrains.com/packages/KaVE.Project/, October 19, 2016.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[15] G. Ghezzi and H. C. Gall. *Distributed and Collaborative Software Analysis*, pages 241–263. Springer, Heidelberg, Germany, 2010.

[16] M. W. Godfrey and L. Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *Transactions on Software Engineering*, IEEE, 2005.

[17] V. U. Gómez, S. Ducasse, and T. D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Computer Languages, Systems & Structures*, 38(1):44–60, 2012.

[18] G. Gousios and D. Spinellis. A Platform for Software Engineering Research. In *International Working Conference on Mining Software Repositories*, 2009.

[19] G. Gousios and D. Spinellis. Alitheia Core: An Extensible Software Quality Monitoring Platform. In *International Conference on Software Engineering*, 2009.

[20] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance (FoSM '08)*, pages 48–57. IEEE, 2008.

[21] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained Version Control System for Java. In *International Workshop on Principles of Software Evolution and Annual Workshop on Software Evolution*. ACM, 2011.

[22] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel. Identifier-based context-dependent API Method Recommendation. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012.

[23] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *International Conference on Software Engineering*, 2012.

[24] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *Transactions on Software Engineering*, 32(12):952–970, 2006.

[25] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[26] M. Kersten and G. C. Murphy. Mylar: A Degree-of-interest Model for IDEs. In *International Conference on Aspect-oriented Software Development*, 2005.

[27] M. Kersten and G. C. Murphy. Using Task Context to Improve Programmer Productivity. In *International Symposium on Foundations of Software Engineering*. ACM, 2006.

[28] P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation*, SCAM'09, 2009.

[29] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The Soot Framework for Java Program Analysis: A Retrospective. In *Cetus Users and Compiler Infastructure Workshop*, CETUS'11, 2011.

[30] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Joint meeting of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering*, 2005.

[31] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Conference on Programming Language Design and Implementation*. ACM, 2005.

[32] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[33] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *International Conference on Program Comprehension*. ACM, 2014.

[34] F. Mccarey, M. Ó. Cinnéide, and N. Kushmerick. Rascal: A Recommender Agent for Agile Reuse. *Artificial Intelligence Review*, 24(3-4):253–276, 2005.

[35] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. Software Developers' Perceptions of Productivity. In *International Symposium on Foundations of Software Engineering*, pages 19–29. ACM, 2014.

[36] A. Michail. Data Mining Library Reuse Patterns in User-selected Applications. In *International Conference on Automated Software Engineering*. IEEE, 1999.

[37] A. Michail. Data Mining Library Reuse Patterns Using Generalized Association Rules. In *International Conference on Software Engineering*. ACM, 2000.

[38] R. Minelli and M. Lanza. Visualizing the Workflow of Developers. In *Working Conference on Software Visualization*, 2013.

[39] R. Minelli, A. Mocci, and M. Lanza. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *International Conference on Program Comprehension*, 2015.

[40] M. Monperrus, M. Bruch, and M. Mezini. Detecting missing method calls in object-oriented software. In *ECOOP 2010–Object-Oriented Programming*. Springer, 2010.

[41] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How Can I Use This Method? In *International Conference on Software Engineering*, 2015.

[42] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[43] E. Murphy-Hill and G. C. Murphy. Recommendation Delivery. In *Recommendation Systems in Software Engineering*. Springer, 2014.

[44] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *International Workshop on Mining Software Repositories*, 2005.

[45] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.

[46] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In *European Conference on Object-Oriented Programming*, 2012.

[47] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *International Conference on Software Engineering*, 2012.

[48] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *Joint meeting of the European Software Engineering Conference and the Symposium on Foundations of Software Engineering*. ACM, 2009.

[49] Opal. http://www.opal-project.de/, October 19, 2016.

[50] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically Checking API Protocol Conformance with Mined Multi-object Specifications. In *International Conference on Software Engineering*. IEEE, 2012.

[51] S. Proksch, S. Amann, S. Nadi, and M. Mezini. A Dataset of Simplified Syntax Trees for C#. In *International Conference on Mining Software Repositories*, 2016.

[52] S. Proksch, S. Amann, S. Nadi, and M. Mezini. Evaluating the Evaluations of Code Recommender Systems: A Reality Check. In *International Conference on Automated Software Engineering*. ACM, 2016.

[53] S. Proksch, J. Lerch, and M. Mezini. Intelligent Code Completion with Bayesian Networks. *Transactions on Software Engineering and Methodology*, 2015.

[54] R. Purushothaman and D. E. Perry. Toward Understanding the Rhetoric of Small Source Code Changes. *Transactions on Software Engineering*, IEEE, 2005.

[55] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Conference on Programming Language Design and Implementation*, 2014.

[56] ReSharper SDK. https://www.jetbrains.com/help/resharper/sdk/, October 19, 2016.

[57] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers. In *International Conference on Software Engineering*, 2014.

[58] A. Rountev, A. Milanova, and B. G. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2001.

[59] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a Software Developer's Local Interaction History. In *In Proceedings of the International Workshop on Mining Software Repositories*, MSR'04, 2004.

[60] W. Snipes, A. R. Nair, and E. Murphy-Hill. Experiences Gamifying Developer Adoption of Practices and Tools. In *Companion of the International Conference on Software Engineering*, ICSE'14, 2014.

[61] Soot. https://sable.github.io/soot/, October 19, 2016.

[62] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. In *International Workshop on Mining Software Repositories*, 2005.

[63] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Symposium on Principles of Programming Languages*, 1996.

[64] S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *International Conference on Automated Software Engineering*. ACM, 2007.

[65] D. Waddington and B. Yao. High-fidelity C/C++ Code Transformation. *Science of Computer Programming*, 68(2):64–78, 2007.

[66] Wala. http://wala.sourceforge.net, October 19, 2016.

[67] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *Transactions on Software Engineering*, IEEE, 2004.

[68] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic Parameter Recommendation for Practical API Usage. In *International Conference on Software Engineering*. IEEE, 2012.

[69] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In *European Conference on Object-Oriented Programming*. Springer, 2009.

[70] H. Zhong, L. Zhang, and H. Mei. Inferring Specifications of Object-oriented APIs from API Source Code. In *Asia-Pacific Software Engineering Conference*, 2008.

[71] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining Version Histories to Guide Software Changes. *Transactions on Software Engineering*, IEEE, 2005.