The IDE as a Scriptable Information System

Dimitar Asenov
Dept. of Computer Science
ETH Zurich, Switzerland
dimitar.asenov@inf.ethz.ch

Peter Müller
Dept. of Computer Science
ETH Zurich, Switzerland
peter.mueller@inf.ethz.ch

Lukas Vogel
Ergon Informatik AG
Zurich, Switzerland
lukas.vogel@ergon.ch

ABSTRACT

Software engineering is extremely information-intensive. Every day developers work with source code, version repositories, issue trackers, documentation, web-based and other information resources. However, three key aspects of information work lack good support: (i) combining information from different sources; (ii) flexibly presenting collected information to enable easier comprehension; and (iii) automatically acting on collected information, for example to perform a refactoring. Poor support for these activities makes many common development tasks time-consuming and error-prone. We propose an approach that directly addresses these three issues by integrating a flexible query mechanism into the development environment. Our approach enables diverse ways to process and visualize information and can be extended via scripts. We demonstrate how an implementation of the approach can be used to rapidly write queries that meet a wide range of information needs.

CCS Concepts

•**Software and its engineering** → **Integrated and visual development environments;** *Software maintenance tools;* •**Information systems** → *Information integration;* •**Human-centered computing** → *Visualization;*

Keywords

code queries; software visualization; refactoring

1. INTRODUCTION

Software development is an information-intense activity. While programming and designing software, developers ask a wide variety of questions [5, 11, 14, 17] and seek information from numerous sources, such as the source code itself, analysis tools, version control information, issue trackers, documentation, project wiki pages, and community resources. In trying to meet their information needs, developers are faced with three issues.

ASE '16 Singapore

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3845-5/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2970276.2970329

First, developers often need to combine information from more than one source, but tool support for piecing information together is lacking [17]. For example, in order to understand a performance regression, it is useful to combine information from the source code (code structure and control flow), the version control system (recent commits and changes to affected code), performance analysis tools (runtime measurements), and an issue tracker (bugs associated with relevant commits). In situations that require diverse information, developers are forced to manually connect the different pieces of information, which is an error-prone and time-consuming process. Such an information search is also tedious to refine as this usually requires the developer to manually repeat a part of the process.

Second, tools most often present information in a fixed form. For example, searching results in a list of matches; querying a program with Ferret [3] results in a hierarchical tree-view. Such one-size-fits-all presentations are not always a good match for a developer's specific information need, but there is very little or no flexibility for customizing the presentation in existing tools. This could hinder the comprehension of the results and makes domain- and project-specific visualizations impossible.

Third, even after a developer finds the information they need, they often have to take action manually. For example, to understand how a set of methods are called, one has to manually set breakpoints or insert print statements. Repeatedly performing an action manually is time-consuming, error-prone, and frustrating. While some tools support automation (e.g., JunGL [18] and Rascal [6] for refactoring), they cannot integrate arbitrary information sources and are usually limited to certain modifications of source code.

To address these three issues, we designed a query system that integrates directly with the integrated development environment (IDE). We make the following contributions:

(i) An approach for querying information within an IDE, which enables the integration of diverse information resources, flexible result presentations, extensibility via scripts, and automated execution of actions.

(ii) An implementation of the approach in the open-source Envision IDE [1].

(iii) An evaluation of the applicability of the system in a number of use cases with diverse information needs.

A video demonstrating our system can be seen at youtu.be/kYaRKuUy9rA. An extended version of this paper [2] contains additional examples, python scripts, and details of our implementation.

2. MOTIVATING EXAMPLES

We will introduce our approach on two practical examples.

2.1 Investigating a Regression

Suppose that a developer is investigating a recently reported regression, where the incorrect behavior occurs after a specific button is pressed. The developer needs to know what code is executed in the button handler and what recent changes might affect this code.

With current tools, the developer will likely first explore what code is being called from the button handler and manually correlate that to recent changes. In particularly hard cases, it might pay off to design a specific test case for this regression and run a binary search on the version repository in order to find the offending commit (e.g., using `git bisect`). Both of these approaches are time-consuming due to ineffective ways of combining source code information (the call graph) with version information (what changed recently).

Our approach offers an alternative solution. The developer can select the handler of the button in the source code, bring up a query prompt and type:

```
callgraph -nodes | changes -c 5 -nodes
```

The `callgraph` query returns the nodes (methods) in the callee graph of the currently selected method. The bug is likely among these methods, but there may be many of them. To narrow down the search, the methods are piped into the `changes` query, which returns only those methods from its input that have changed in the last five commits. After the query is executed, the remaining methods will be highlighted, helping the developer to more quickly find the issue.

Enabling this workflow are three key components of our approach: **(i)** a **context-sensitive query prompt** that enables developers to type and combine queries; **(ii)** diverse **queries** that can access arbitrary resources such as the source code or version repository; and **(iii)** a **unified data format** that enables queries to be combined in order to refine searches.

2.2 Heatmap of Code Execution

Imagine that a developer wants to get a visual overview of the often-executed parts of the code to gain a general understanding of which classes are relevant for performance. Thus, it is preferable to see frequently executed methods in a broader context.

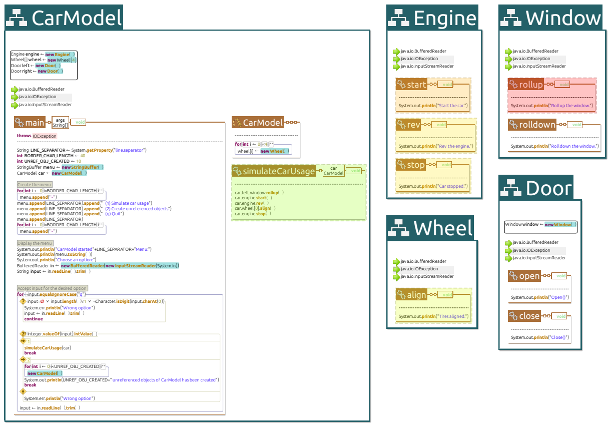
Profiling tools typically provide timing information in the form of a chart, graph, or a list. However none of these presentations fits the developer's need in this example, as the code around performance critical methods is also important.

This is one example where the presentation of information is critical for understanding and where our approach's support for flexible visualizations can help. The developer could, for example, export the timings to a CSV file and use a query to import it into the IDE and visualize the results:

```
importProfileCSV profile.csv | heatmap
```

The first time they do this, they will have to write the `importProfileCSV` Python script (about 15-20 lines) that reads the CSV file into the data format understood by our system. The data is piped into the `heatmap` query, which highlights different parts of the code with a color in the red-green spectrum based on the value of a number (Fig. 1).

This example illustrates two more essential components of our approach: **(iv)** integration with a **scripting language**,



```
Engine engine ← new Engine( )
Wheel[] wheel ← new Wheel[4]
Door left ← new Door( )
Door right ← new Door( )

java.io.BufferedReader
java.io.IOException
java.io.InputStreamReader

main
throws IOException
String LINE_SEPARATOR ← System.getProperty("line.separator")
int BORDER_CHAR_LENGTH ← 40
int UNREF_OBJ_CREATED ← 10
StringBuffer menu ← new StringBuffer( )
CarModel car ← new CarModel( )

Create the menu
for int i ← 0 i<BORDER_CHAR_LENGTH i++
    menu.append("-")
menu.append(LINE_SEPARATOR).append("  (1) Simulate car usage")
menu.append(LINE_SEPARATOR).append("  (2) Create unreferenced objects")
menu.append(LINE_SEPARATOR).append("  (q) Quit")
menu.append(LINE_SEPARATOR)
for int i ← 0 i<BORDER_CHAR_LENGTH i++
    menu.append("-")

Display the menu
System.out.println("CarModel started"+LINE_SEPARATOR+"Menu:")
System.out.println(menu.toString( ))
System.out.println("Choose an option:")
BufferedReader in ← new BufferedReader(new InputStreamReader(System.in))
String input ← in.readLine( ).trim( )

Accept input for the desired option
for ¬input.equalsIgnoreCase("q")
    ? input=∅ ∨ input.length( )≠1 ∨ ¬Character.isDigit(input.charAt(0))
        System.err.println("Wrong option")
        input ← in.readLine( ).trim( )
        continue
    ?= Integer.valueOf(input).intValue( )
    1
        simulateCarUsage(car)
        break
    2
        for int i ← 0 i<UNREF_OBJ_CREATED i++
            new CarModel( )
        System.out.println(UNREF_OBJ_CREATED+" unreferenced objects of CarModel has been created")
        break
        System.err.println("Wrong option")
    input ← in.readLine( ).trim( )

CarModel
for int i ← 0 i<4 i++
    wheel[i] ← new Wheel( )

simulateCarUsage
car.left.window.rollup( )
car.engine.start( )
car.engine.rev( )
car.wheel[0].align( )
car.engine.stop( )
```

```
Engine
java.io.BufferedReader
java.io.IOException
java.io.InputStreamReader
start
System.out.println("Start the car.")
rev
System.out.println("Rev the engine.")
stop
System.out.println("Car stopped.")

Wheel
java.io.BufferedReader
java.io.IOException
java.io.InputStreamReader
align
System.out.println("Tires aligned.")

Window
java.io.BufferedReader
java.io.IOException
java.io.InputStreamReader
rollup
System.out.println("Rollup the window.")
rolldown
System.out.println("Rolldown the window.")

Door
Window window ← new Window( )
java.io.BufferedReader
java.io.IOException
java.io.InputStreamReader
open
System.out.println("Open()")
close
System.out.println("Close()")
```

Figure 1: A heatmap overlayed on top of a visual presentation of code showing a few classes (▣) and methods (⚙). The heatmap is visualized as a set of translucent overlays on top of methods; each overlay has a color in the spectrum between red and green, indicating how often a method was executed. Similar visualizations are also possible in traditional text-based IDEs, e.g., using line or file highlights.

allowing easy extension to new information resources and customized queries; and **(v) flexible visualizations**, which enable task-specific display of information, helping comprehension.

3. APPROACH

Our goal was to design a system that is highly expressive and extensible by the user in order to satisfy a wide range of information needs. The architecture of our system is shown in Fig. 2. Below, we discuss each component in detail.

3.1 Query Execution Model

The core of our approach is the ability to **compose** and **execute queries**. This functionality is provided by the execution engine, which enables queries to be connected in a network (directed acyclic graph), where the edges represent data flow. Such a network of queries is illustrated in Fig. 3.

Each query has a **context**, which is the AST node (e.g., a method) on which the query was invoked. A query has two additional ways to access information.

First, a query may be connected to the output of other queries via any number of required or optional inputs, which comprise the **inter-query data exchange**. For example, a query might receive on its input a set of method AST nodes, and output their names as a set of strings. The data flowing between queries uses a unified format (see Sec. 3.3).

Second, a query may access (that is, read and modify) **external resources** (see Sec. 3.2.1), such as the program's source code (e.g., to perform a refactoring), call a method of the IDE (e.g., to access the AST or show a message), or use a REST service (e.g., to create a bug report).

A query is executed only after all of its inputs are read. When a query is run, it can perform arbitrary computation, which typically includes accessing external resources and computing outputs. Once a query has finished executing, its outputs are forwarded to any downstream queries.

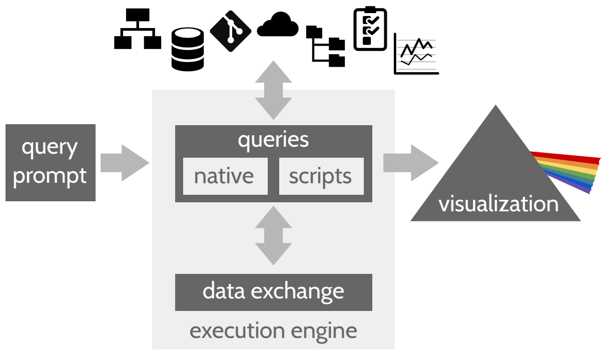


Figure 2: The architecture of our system. Using a query prompt, a developer can invoke and combine queries. Queries can access diverse information resources, make computations, and produce visualizations, and can be either native or implemented via scripts. A unified data exchange format facilitates the cooperation between multiple queries. An execution engine orchestrates query execution and the information flow between queries.[1]

3.2 Query Types

In principle a query can perform arbitrary computation. However, to facilitate composition, we divide queries into three types: **resource-access**, **visualization**, and **operator**. Below, we define each type and explain how we designed corresponding queries in order to improve usability.

3.2.1 Resource Access

Resource-access queries are used to read and modify resources external to the execution engine. They connect a network of queries to the AST and external tools and data. A resource-access query can be a **source of data** for other queries. For example, a query may read the contents of a file and provide it as inputs to other queries for processing. A resource-access query may also **modify external data**. For example, it could create a new record in a database, or modify the program. There is no limit on what resources a query may access; common ones are the program AST, the version repository, the issue tracker, files, and on-line services.

To facilitate composability, we designed resource-access queries according to the following guidelines. First, a resource-access query provides access to only one resource. This restriction allows accessing one resource without imposing requirements on another. Second, complex resources are accessed by multiple queries, which enables each query to focus on a particular aspect of the resource. For example, when reading the program's AST, one query is used to select nodes while another provides control-flow information. Third, when integrating tools that have a command line interface, we created queries with a similar interface. This enables developers to transfer some of their existing knowledge from the terminal command to the query. For example, in a query that accesses a Git repository, commits can be specified by commit id, branch name, or reference, like the `git` command allows.

A noteworthy resource available through resource-access queries is the IDE itself. A query may call available IDE APIs

[1]The git logo by Jason Long and icons made by Freepik from www.flaticon.com are licensed by CC BY 3.0.

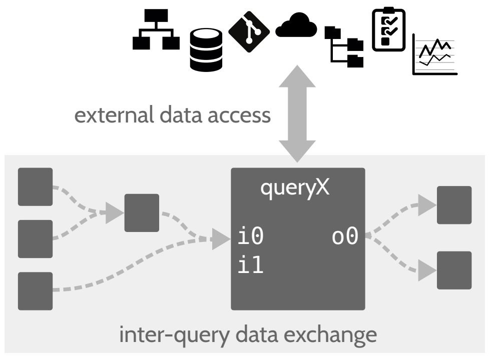


Figure 3: A query network with seven interconnected queries. *queryX* is shown in detail. It has two inputs (`i0`, which is the union of two outputs from other queries and `i1`, which is unused here), and one output (`o0`, which is duplicated).

to get information or to perform IDE functions. For example, most IDEs maintain a code model that provides easy access to the program's AST. Such queries are not limited to extracting information. Depending on available IDE APIs, queries might be used for navigation between code fragments, setting breakpoints, running tests, displaying warnings or errors, and refactoring code.

3.2.2 Visualization

Visualization queries are used to **render information on the screen**. Different visualization queries can be used to render the same piece of information in different ways in order to better match the specific information needs of the developer. For example, we support three ways to visualize relations between code elements: (i) show the relations using a textual notation – useful for a dense summary; (ii) highlight on screen all code elements that appear in the relations – useful when searching for particular patterns; (iii) show the relations using arrows between code elements – useful when exploring call graphs or data flow. Some visualizations are provided by the underlying IDE, while others could be done via external tools. Common visualizations in IDEs are highlighting a program fragment, showing a list or a tree of result entries, and displaying error messages and warnings. More visual IDEs could even provide a map of the code that enables intuitive arrow overlays to explore connections between elements or even a heatmap visualization similar to Fig. 1.

Our approach imposes no limit on how information can be visualized. If an IDE exposes general drawing routines, a query could use those to implement an entirely custom visualization. Some IDEs provide a built-in HTML rendering engine, which could be used to easily and quickly implement new ways for visualizing information. Using a combination of HTML5 and Javascript, it is even possible to create interactive visualizations as we demonstrate in Sec. 4.1. This high degree of flexibility is indispensable for domain- and project-specific visualizations.

We designed visualization queries according to the following guidelines in order to make them easier to use. First, each available visualization mechanism has its own query, which makes it clear what will appear on the screen when it is invoked. Second, visualizations impose as few requirements on the input data as possible, so that one visualization can be easily swapped for another. Third, if a visualization query is not explicitly provided, but there is unconsumed data at the end of a query-network execution, a visualization is automatically chosen based on the structure of the result. This frees developers from having to always explicitly specify a visualization that could be automatically inferred.

3.2.3 Operator

Operator queries (operators) are used to **perform internal computation**, for example, to refine results. Operators do not access any external resources but rather help **filter and combine data** in complex query networks. They work solely with the unified data format, which we discuss in Sec. 3.3. As operators work with sets and relations, they naturally map to operations from set and relational algebra such as union, intersect, select, and join. Building on these primitives, we have also pre-defined more elaborate operators in order to simplify common cases. For example, in Sec. 4.3, we demonstrate the `reachable` operator, which filters out elements unreachable from a starting point via a relation – in essence a combination of transitive closure and selection. Such a convenience operator is very useful in answering reachability questions, which are very common [14].

3.3 Inter-query Data Exchange

To enable query composition, all queries communicate using a simple unified structure for exchanging data. This unified exchange structure is a **set of tuples**, because it is sufficiently **expressive** and provides a **simple mental model** for developers to work with.

Each input and output of a query is a set of tuples, and each tuple consists of an arbitrary number of named elements. The names of elements within a single tuple have to be unique. The elements of a tuple may be strings, integers, and references to AST nodes. Each tuple has a tag, which is an identifier that is either explicitly provided or is identical to the name of the tuple's first element. This minimal structure allows us to conveniently encode and access typical structures such as sets, lists, and graphs.

Filtering and combining can be easily done based on the name or value of elements, or the tags of tuples. Result visualizations can be automatically selected based on the tags and tuples present in the final output. For example, a tuple with the tag `message` could be shown as a message associated with a code location.

3.4 Extensibility: Scripts and Native Queries

Using our system, the simplest way to access and manipulate information is to compose existing queries directly on the query prompt and get results immediately. Query composition covers a broad range of common needs with minimal effort, but inevitably, developers will need specialized behavior where composition will not be enough. To support custom information processing we provide two extensibility mechanisms.

The first one is to implement new queries as light-weight scripts, in our case using Python. Scripts allow orchestrating complex query flows and provide access to specialized resources, which is made easier by existing libraries for the scripting language. Scripts have access to a limited API: the context, inputs, and outputs of the query they represent, and the program's AST. This API is enough to make

```python
for node in Query.input.tuples('ast'):
  if isinstance(node.ast, IfStatement):
    if node.ast.elseBranch.size() > 0:
      Query.result.add( node )
```

Figure 4: A Python script that filters input nodes.

scripts versatile while keeping them extremely simple. For example, Fig. 4 shows the complete script for selecting only if-statements that have else-branches from the input. Implementing a query via a script is as easy as writing the script file. Any available scripts are directly invokable on the query prompt and scripts are seamlessly composable with other queries. The execution engine translates the inter-query data format to and from the native environment and the scripting language's virtual environment.

The second extension mechanism is to create new native queries within the host IDE of our system. Native queries give the developer unlimited power to perform specialized computation and allow deep IDE integration — unlike scripts, native queries have access to all IDE APIs. The drawback of this approach is that it is more demanding and time-consuming than writing a script, since developers will have to, essentially, extend the host IDE (e.g., write an Eclipse plug-in).

In practice, native queries, which provide deep IDE integration, and scripts, which access a custom resource, complement each other well, as we show in Sec. 4.

4. CASE STUDIES

We will demonstrate the applicability of our approach in three practical programming scenarios. All examples have been tested using our implementation inside the Envision IDE [1], which offers a more visually-rich code presentation compared to traditional IDEs and allows us to more easily implement diverse visualization queries.

4.1 Flexible Visualizations

Programmers frequently need to understand control flow and, in particular, follow paths through several method calls [14]. As the call graph is a commonly needed piece of information, we provide a native query:

```
callgraph
```

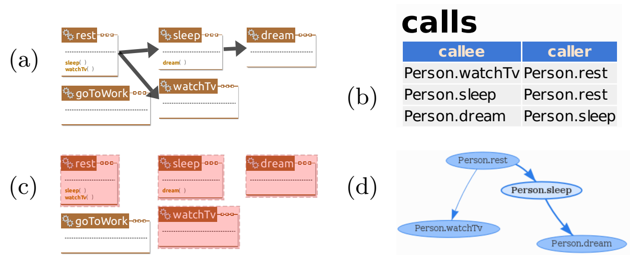
Because it is **context-sensitive**, this query will automatically return the callee graph of the method that contains the cursor. The query itself just returns a set of tuples and does not produce any visualization. As no visualization is explicitly provided, the execution engine automatically chooses one based on the structure of the result. By default, results that represent relations between AST nodes are visualized as arrows connecting the nodes' representations on screen as shown on Fig. 5(a). Alternatively, the developer could show all result tuples in a table (Fig. 5(b)):

```
callgraph | table
```

or highlight visually the methods that are part of the call graph without showing arrows (Fig. 5(c)):

```
callgraph -nodes
```

Here, the execution engine will detect that the output is a set of AST nodes and will highlight them on the screen. The visualizations from Figs. 5(a), 5(b), and 5(c) use Envi-



calls

callee	caller
Person.watchTv	Person.rest
Person.sleep	Person.rest
Person.dream	Person.sleep

Figure 5: Different ways of visualizing results that describe a relation: (a) arrows between related elements; (b) a table; (c) highlighting relation elements; (d) a custom HTML visualization: in this case, an interactive graph rendered using `vis.js`.

sion's visual code presentation, but analogous visualizations also exist in conventional IDEs: line highlights, and arrows between lines. These three visualization options illustrate an important aspect of our approach: the results of queries are decoupled from their visualizations. This decoupling enables **flexible visualizations** that allow the presentation of the results to more accurately match the information need of the developer. Common presentations such as highlights and arrows can be readily provided by the IDE. As we show next, developers can also add their own visualizations.

In some projects or domains, specialized visualizations enable better information comprehension. Our approach enables the creation of custom visualizations using web technologies. For example, we used the open-source *vis.js* library and a custom Python script to implement the visualization from Fig. 5(d). With a total of 80 lines of Javascript and Python code we can run the query:

```
callgraph | toHtmlGraph
```

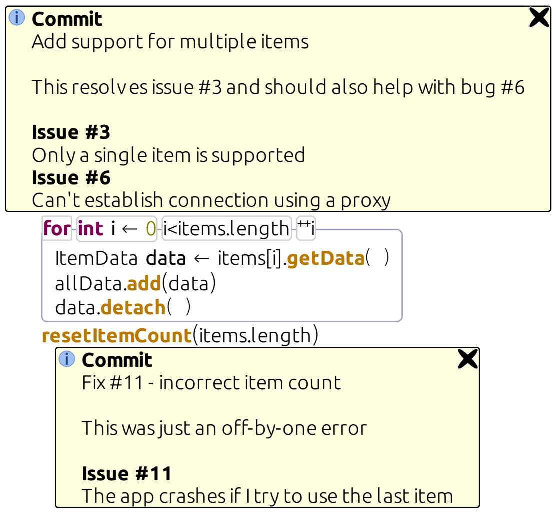
and get an interactive view of a graph. The `toHtmlGraph` script converts the tuple set from our system into an HTML page that uses *vis.js*, which renders the interactive graph.

4.2 Combining and Transforming Information

Programmers often need to know the reason some code exists or looks the way it does (e.g., questions 8-10 from Fritz and Murphy [5]). One approach to gain this knowledge is to connect a piece of code with version control and bug database information. Here is how such a query could look:

```
ast -type Statement -topLevel
| changes -intermediate
| join change.id,commit.message,ast
  -as data
| associatedBugs
```

This is a more complex query, but building it piece by piece is rather straight-forward. We want information about each statement in a method, so we first get all top-level (non-nested) statements using `ast -type Statement -topLevel`. The result is piped into the `changes` query, which yields change information about all commits that modify any of the input statements. The result consists of two different kinds of tuples: the first one associating each node with the id of the commit where it changed, the second associating a commit id with the commit's meta data (e.g., the commit message). Using the `join` query, we merge those two different kinds of tuples into a single kind that relates nodes and



Commit
Add support for multiple items

This resolves issue #3 and should also help with bug #6

Issue #3
Only a single item is supported
Issue #6
Can't establish connection using a proxy

```
for int i ← 0 i<items.length ++i
    ItemData data ← items[i].getData( )
    allData.add(data)
    data.detach( )
resetItemCount(items.length)
```

Commit
Fix #11 - incorrect item count

This was just an off-by-one error

Issue #11
The app crashes if I try to use the last item

Figure 6: A for-loop and a method call with corresponding explanations for why they were last changed. The information bubbles are standard visualizations in the Envision IDE.

commit messages and that we call `data`. We pipe `data` into the `associatedBugs` script, which can be seen in [2]. The script uses a REST request to associate commits with issues on GitHub. The output of the script is a set of tuples representing `info` messages, which are automatically shown using standard Envision information bubbles next to the corresponding code (Fig. 6).

Functionality for explaining the reason for a piece of code is also available in other tools like the `blame` command for version control systems, or the Eclipse annotate feature. However, our approach allows building this information from **elementary blocks** and precise controlling of what is included. We could easily **combine different information sources** and **transform the information** to fit our needs.

4.3 Automating Actions

Programmers often have to change code at scale. Development tools typically provide only a limited set of refactorings. The fallback solution is most often textual search and replace. However, many situations are not expressible by regular expressions. For example, we might want to modify the program so that each recently changed recursive method prints the values of its arguments when called. This could be useful in debugging a termination issue. We could add the necessary print statements using the query:

```
callgraph -global | reachable -self
| changes -c 5 -nodes | insertArgPrinting
```

First, we refine the query from Sec. 2.1, by using the `reachable` query, which filters out AST nodes that cannot reach themselves following the relations from the input. Thus, we get all recursive methods that have changed recently. Then we pass these methods to the `insertArgPrinting` script, which we show in [2]. For each provided method the script inserts code that prints its name and lists all arguments.

Several things are worth noting here. First, it was possible to **easily refine** a previous query, which Sillito et al. [17] identify as a gap in existing tools. Second, we executed a query which **modified a resource**, in this case the source code. Third, this modification uses both non-trivial program properties (recursion) and information other than the source code itself (version information). Thus, it is outside the reach of regular expressions and most refactoring tools, which can only use information from the source code. Fourth, the support for scripts makes it easy to perform program edits within the overall query mechanism. Just like any other query, edits can also depend on input from other queries and external data resources, which enables **data-driven changes to the code**.

5. RELATED WORK

A number of recent studies [5, 11, 14, 17] investigate what questions developers ask and how they seek answers to these questions. Our approach supports answering many of these questions and addresses some of the major gaps identified by earlier work, e.g., the need to share and transform information [11] and the lack of support for combining information, refining questions, and using context [5, 17].

Researchers have developed a variety of languages and techniques for querying source code [4, 7, 10, 15, 16, 19]. Such query tools offer powerful and efficient ways to query program properties, but are typically restricted to the program source and do not integrate other information resources. A number of tools [3, 5, 9] do allow the combination of different data sources. Our approach is different from these approaches in several ways: (i) the link between different sources does not need to be predefined, the developer has full control over what information is linked; (ii) the simple tuple set interface of queries allows for the easy addition of a wide range of information resources; (iii) result visualizations are flexible to better match specific information needs.

There is a wealth of tools for visualizing information related to software. However these are typically coupled to a specific kind of inquiry. Most tools that provide general query capabilities provide only a single way to view results or only basic visualization flexibility. Common result presentations are list or tree views [3, 4, 5, 7, 10, 19] and graphs [13, 16]. Some tools allow configurable views [5, 15] or advanced interfaces [8, 13, 12]. Unlike these other tools, our approach is more general and offers flexible and easily extensible visualizations, which can also be interactive.

Some program querying tools can also make modifications to code. Two scripting languages for refactoring are JunGL [18] or Rascal [6], which offer powerful capabilities to analyze and transform the source code. Our system also enables complex code changes via Python scripts and it offers two major advantages: (i) scripts can use external resources in addition to the source code, enabling data-driven code modification; and (ii) modifying code is just a special-case for our system's general support for automating arbitrary actions.

6. CONCLUSION

We showed an approach to turn an IDE into a powerful and customizable information system. A **unified data exchange format** enables **query composition** and allows developers to combine **diverse information resources**. The results of such queries can be presented using a **wide range of visualizations**, enabling a better fit for a developer's particular information needs – a key in improving comprehension. Our approach can also be used for **automating developer actions** and **data-driven modification** of code. We showed that the integration of the Python **scripting language** unlocks great extensibility options, providing a platform for rapidly integrating new information resources or performing complex data manipulation. A user study is future research, which would enable us to further asses the usability of our approach.

7. REFERENCES

[1] D. Asenov and P. Müller. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *VL/HCC '14*.

[2] D. Asenov, P. Müller, and L. Vogel. The IDE as a Scriptable Information System (extended version). In *arXiv:1607.04452*.

[3] B. de Alwis and G. Murphy. Answering conceptual queries with Ferret. In *ICSE '08*.

[4] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ '11*.

[5] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *ICSE '10*.

[6] M. Hills, P. Klint, and J. J. Vinju. Scripting a refactoring with Rascal and Eclipse. In *WRT '12*.

[7] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD '03*.

[8] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *UIST '11*.

[9] A. Kellens, C. De Roover, C. Noguera, R. Stevens, and V. Jonckers. Reasoning over the evolution of source code using quantified regular path expressions. In *WCRE '11*.

[10] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *ASE '11*.

[11] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE '07*.

[12] J.-P. Krämer, J. Kurz, T. Karrer, and J. Borchers. Blaze: Supporting two-phased call graph navigation in source code. In *CHI '12*.

[13] T. LaToza and B. Myers. Visualizing call graphs. In *VL/HCC '11*.

[14] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *ICSE '10*.

[15] O. Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .ql: Object-oriented queries made easy. In *GTTSE '07*.

[16] T. Schafer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE TSE*, 32(9).

[17] J. Sillito, G. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE TSE*, 34(4).

[18] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A scripting language for refactoring. In *ICSE '06*.

[19] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall. Supporting developers with natural language queries. In *ICSE '10*.