Integrating Runtime Values with Source Code to Facilitate Program Comprehension

Matúš Sulír
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Košice, Slovakia
matus.sulir@tuke.sk

***Abstract*—An inherently abstract nature of source code makes programs difficult to understand. In our research, we designed three techniques utilizing concrete values of variables and other expressions during program execution. RuntimeSearch is a debugger extension searching for a given string in all expressions at runtime. DynamiDoc generates documentation sentences containing examples of arguments, return values and state changes. RuntimeSamp augments source code lines in the IDE (integrated development environment) with sample variable values. In this post-doctoral article, we briefly describe these three approaches and related motivational studies, surveys and evaluations. We also reflect on the PhD study, providing advice for current students. Finally, short-term and long-term future work is described.**

***Index Terms*—integrated development environment, documentation, debugging, dynamic analysis, variables**

I. Introduction

In this article, we would like to summarize some of the main results of the thesis [1]. We also describe the lessons learned and directions for future research.

A. Background

Maintenance of existing software systems requires the developers to understand the programs of interest. This is accomplished by gradually building a mental model of selected parts of the program [2]. One way to build such a mental model is to read the source code lines in the editor. However, the source code provides only a static and abstract view of the program, separated from its runtime properties. To connect these two separate worlds, there exists a large variety of methods, approaches and tools.

In our research, we are particularly interested in three types of activities related to program comprehension.

First, there is a need to find the relevant pieces of code. This process is known as concept location (or feature location [3]). Second, to gain an overview of the behavior of the individual methods in the code, developers often read API (Application Programming Interface) documentation [4]. Third, to understand the details of a particular method, the developers can read the source code of the method definition. To alleviate this, many tools try to visually augment the source code directly in the editor to provide additional information in-place [5].

This work was supported by project KEGA 047TUKE-4/2016 Integrating software processes into the teaching of programming.

Dynamic analysis, i.e., the analysis of a running program, is a well-known approach to facilitate software comprehension and maintenance (see, e.g., [6], [7], [8]). However, the program execution is usually captured at a high level. The execution is often perceived only as a sequence of method calls, object creations or line executions. For example, none of the feature location approaches described in the articles surveyed by Dit et al. [3] analyzed concrete values of local or member variables during executions.

B. Synopsis

The main goal of our research is to ease program understanding by integrating runtime information with the source code. Particularly, we are focused on concrete values of individual variables and expressions (such as local variables, arguments, return values or member variables). We designed three techniques aiming to help the developers to perform the three aforementioned activities – searching, documentation reading, and source code reading:

- RuntimeSearch, a debugger extension which allows for searching a given text in all string expressions in a running program [9],
- DynamiDoc, an automated documentation generator producing sentences with examples of arguments, return values and object state changes collected during executions [10],
- RuntimeSamp – an IDE (integrated development environment) plugin showing a sample value for each variable at the end of each line in the source code editor [11].

Along with the design of these three tools, we performed supporting empirical studies and conducted related surveys. In the following chapters, we will briefly describe each of the approaches and related findings.

In Fig. 1, there is an example of how the three designed techniques might be used together. However, note that each tool is also useful on its own.

II. Searching

An important task during software maintenance is to find where the given functionality is implemented. Especially if the software is large, finding an initial investigation point in the codebase is difficult. Although there exists a large number of

 This is the accepted version of: M. Sulír. Integrating Runtime Values with Source Code to Facilitate Program Comprehension. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 743–748. http://doi.org/10.1109/ICSME.2018.00093

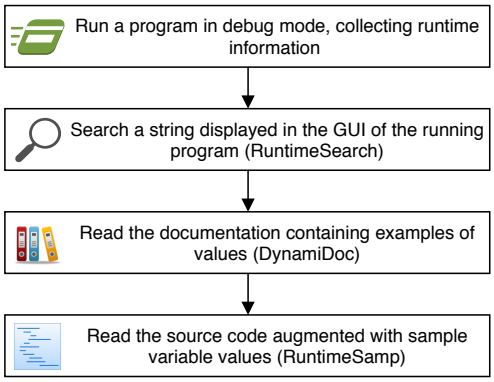

Fig. 1. An example of a combination of the designed techniques

feature location methods, they are rarely used in practice – industrial developers prefer traditional approaches such as a textual search in the source code [12], [13].

A. Empirical Study

The search queries of developers often contain terms obtained by an observation of a running program. For instance, a developer can try to search for a label displayed in the graphical user interface (GUI) of a running application [14]. A programmer also tends to ask what part of the code generated the displayed error message [15].

A naive strategy is to statically search the displayed string in the code as-is. In our small-scale study, we aimed to find to what extent this strategy is sufficient [16]. Four desktop Java applications were scraped to produce a list of strings and words displayed in their GUIs, such as menu items or button labels. We found that about 11% of strings displayed in the GUIs of running programs were not found in the source code at all, making this strategy ineffective in these cases. More than 24% of them had more than 100 occurrences, which can be considered too much to be practical for the inspection of all results.

B. RuntimeSearch

Given this motivation, we designed RuntimeSearch – a variation of a traditional text search, but for a running program instead of the static source code [9]. The target application is executed in the debug mode. At any time, the programmer can enter a string into a text field provided by RuntimeSearch. It is subsequently searched in all string-typed expressions being evaluated, such as all string variables and method return values. When a match is found, the program is paused and the traditional IDE debugger is open, offering all standard debugging possibilities, including the inspection of current variable values, stepping and resuming the program. If the current location is irrelevant, we can continue by finding next occurrences.

In contrast to conditional breakpoints, RuntimeSearch searches in all expressions in the program (or the selected packages/classes), not only the selected lines. On the other hand, its capabilities are currently limited. Particularly, it supports only simple string matching. More search options, such as regular expressions, are planned for the future.

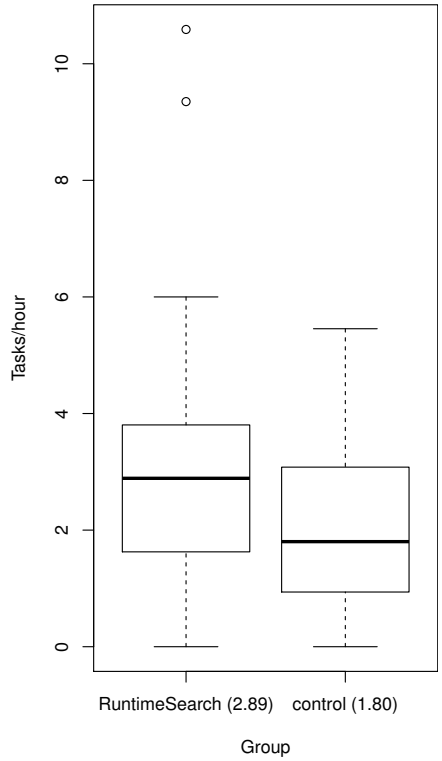

Fig. 2. Results of the RuntimeSearch controlled experiment

C. Evaluation

First, in a case study on a 350 kLOC (thousands of lines of code) program, we found RuntimeSearch can be useful [9]:

- to find an initial point of investigation (e.g. search for a text displayed in the GUI),
- to search for multiple occurrences of the same string across multiple layers, such as from the GUI through helper methods to file-related routines,
- to search for non-GUI strings, e.g., texts located in files,
- to confirm programmer's hypotheses (for instance, trying to find the string "https://" if the HTTPS connection is used).

The second mentioned point can also be achieved by using a technique we called the "fabricated text technique": to enter a dummy text into a part of program accepting textual input (e.g., a text field) and observe the data flow through multiple layers by finding its occurrences using RuntimeSearch.

Next, to validate our approach, we performed a (not yet published) controlled experiment with 40 human participants [1]. One group used RuntimeSearch to perform simple search-focused program maintenance tasks, while the other group could use only standard IDE features. The results of the experiment are in Fig. 2. The treatment group achieved 60% higher median efficiency in terms of tasks per hour. The difference was statistically significant.

The participants of the experiment were masters students. We received positive feedback from them, multiple students asked whether this tool is publicly available[1]. We consider RuntimeSearch a tool which can be soon ready for an industrial transfer – we plan to publish it on the JetBrains Plugin Repository[2]. However, first we should make the plugin more production-ready: clean the code, reduce manual steps required to setup the plugin, write the documentation, etc.

III. Documentation

While API documentation is a useful resource for programmers, writing it and keeping it consistent with the source code requires a huge effort. Therefore, many automated approaches to documentation generation were devised. However, they traditionally process only the static source code or artifacts like mailing lists [17]. Although there exist documentation generators utilizing runtime information, they are specialized – e.g., FailureDoc [18] for failing unit tests or SpyREST [19] for RESTful (Representational State Transfer) APIs.

A. DynamiDoc

We designed DynamiDoc, an example-based documentation generator utilizing runtime information collected during unit test executions or debugging [10]. For each method (function), it collects:

- string representations of arguments and return values,
- the string representations of the target object (`this`) before and after calling the given method,
- and thrown exception types.

The representations of objects are obtained using the standard `toString()` method in Java, which has an alternative in almost all languages.

Then, using a decision table with sentence templates, DynamiDoc generates documentation sentences containing examples of these values. For instance, an excerpt from the documentation of the method Range.lowerBoundType() from Google Guava[3] may look like this:

```
When called on (5..8), the method
 returned OPEN.
When called on [5..8), the method returned
 CLOSED.
```

B. Evaluation

Using a qualitative evaluation [10], we found out DynamiDoc is particularly useful for the documentation of utility methods and data structures. On the other hand, methods which manipulate classes not having the `toString()` method meaningfully overridden and methods interacting with the external world are not the best candidates for DynamiDoc documentation.

We also performed a preliminary quantitative evaluation [20]. We found that on average, one documentation sentence has 10% of the length of the method it describes, so it is sufficiently succinct. By manually inspecting a sample of documentation sentences, we found 88% of the described objects have the `toString()` method overridden. Therefore, we fulfilled basic prerequisites for the usefulness of this approach.

IV. Augmentation

Since an understanding of a program only by reading its source code is difficult, many tools augment it with various metadata – from manually written notes through performance data to information about related emails.

A. Surveys

In our article [21], we described a taxonomy of source code labeling. The taxonomy consists of four dimensions: source (where the metadata come from, such as static or dynamic analysis), target (granularity – whole method, line, etc.), presentation (in the editor or a separate tool) and persistence.

Then we performed a systematic mapping study [5], summarizing existing tools which visually augment the textual source code editor with various icons, graphics and textual labels. We found more than 20 tools augmenting the code with runtime information, but very few of them aim to display examples of concrete variable values. IDE sparklines [22] are limited to numeric variables, Debugger Canvas [23] requires the developer to manually select individual states during debugging and the prototype by Krämer et al. [24] suffers from scalability issues. Tralfamadore [25], [26] displays only arguments and return values.

B. RuntimeSamp

Our IDE extension RuntimeSamp [11] collects a few sample values of each variable during normal executions of a program by a developer, such as testing or debugging. Then, at the end of each line, one sample value is shown for each variable read or written on the given line. A demonstration, showing an excerpt from the Apache Commons Lang[4] library can be seen in Fig. 3. The idea behind the tool is that concrete variables should help the developers to get the "feeling" of runtime and concreteness in the inherently abstract and static source code.

Compared to DynamiDoc, RuntimeSamp provides more fine-grained data – it displays information for individual lines and variables instead of whole methods. Furthermore, it is an interactive IDE extension, while DynamiDoc generates static textual documentation.

In our article [11], we asked 7 questions which should be answered for RuntimeSamp to be useful in practice:

- How to represent complicated objects succinctly?
- When should we capture the variable values (e.g., is one value per line sufficient)?
- If one line is executed more than once, how to decide which iteration to display?
- How to detect and present such iterations?

[1] Similar to other tools mentioned in this paper, RuntimeSearch is available online: http://sulir.github.io/runtimesearch

[2] http://plugins.jetbrains.com

[3] https://github.com/google/guava

[4] https://commons.apache.org/lang/

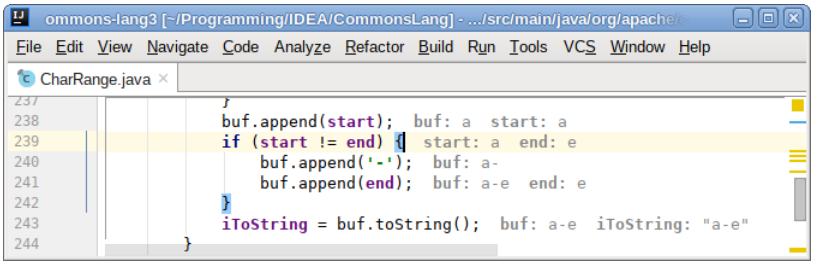

Fig. 3. RuntimeSamp showing source code augmented with sample variable values (gray)

- How to keep the time overhead reasonable during the data collection?
- Is it necessary to filter the displayed variables?
- When to invalidate the data?

For now, we answered these questions mainly in naive ways. To display the values of objects, we use their standard string representations (`toString`). We capture the values at the end of each line. Since we consider the caret (text cursor) as an implicit pointer to the programmer's focus point, the first iteration which covers the line at the cursor is always displayed. An iteration is defined as a forward execution (without backward jumps) within one method. When collecting one sample value for each variable, the time overhead is about 78–213%, which is not prohibitive, but certainly requires an improvement. The measurement was performed using the DaCapo benchmark [27]. We filter the displayed data using a simple rule to prevent redundancy and invalidate all data on any edit (which is only a preliminary solution).

V. Lessons Learned

In this section, we would like to describe reflections on the PhD study and advice for other students.

A. Seek Collaboration

Some of the most valuable publications (e.g., [28]) during the PhD study were written in collaboration with other members of our research group. More people can afford to complete more time-consuming tasks – this is particularly true if they can be easily divided to sub-tasks, such as certain kinds of controlled experiments or systematic reviews.

Since international collaboration is not an integral part of the research process at our institution, and we did not actively seek such a collaboration, none of the papers included in the dissertation was co-authored by people outside our research group. Therefore, cooperation with other institutions is planned in the near future. A good piece of advice for students is to actively search for opportunities to collaborate with people with similar research ideas during their studies, e.g., at conferences.

B. Focus on Your Topic

Although collaboration is useful, it can be also considered a double-edged sword. Since the persons you collaborate with may have slightly different research interests than you, the cooperation with them can act as a distraction from the main goals of your thesis. This may make the process of your dissertation completion challenging: You will be left with an option to either make your dissertation topic too broad or exclude a large number of valuable papers from the thesis.

Of course, collaboration is not the sole cause of distraction from the thesis topic. During the initial periods of the PhD study, we had multiple potential ideas for the dissertation topic and we even tried to pursue some of them although they had little in common. While this resulted in some interesting research results (e.g., about build system failures [29]), it also delayed the progress on the main topic.

VI. Future Work

Finally, we will present our short-term future research tasks and long-term visions.

A. Short-Term Goals

Currently, we are working on the first question mentioned in section IV-B: How to represent an object, consisting of many properties, on a limited space?

Before considering graphical representations, let us focus on the textual ones. The solution used in RuntimeSamp (and also in DynamiDoc) is to convert it to a string using a standard "toString"-like method, available in many languages, including Java. However, this representation must be written manually by the programmers, which is a reason it is sometimes left with its default (useless) implementation. Using machine learning, we try to automatically generate the string representation of objects, listing only subsets of their member variables which are considered important by programmers.

Another short-term goal is to evaluate DynamiDoc and RuntimeSamp using experiments with human participants.

B. Long-Term Goals

The first long-term goal is to extend the object representation question to graphical representations. We can recognize two extremes: On one end, there are generic tree-based and graph-based (such as DDD [30]) visualizations displaying all properties of the objects, suitable for any kind of data, but revealing little domain-specific information. On the other hand, approaches such as Moldable Inspector allow the developers to craft graphical representations perfectly suitable for a particular domain, but they require manual coding effort [31]. Finding a right compromise between these two extremes is the challenge we would like to address next. This can be even

more complicated if we consider not only one state, but also a difference of two or multiple states.

Our main long-term goal is to blend the activities of source code reading/editing and an observation of the runtime properties of the application, so that the line between them will be almost indistinguishable.

One of the research areas aiming to clear this boundary is the area of live programming systems. A large amount of work was done in this field – from the design of live programming languages [32], [33] and their visual augmentation [34] to experiments [24] and integration with unit testing [35], just to name a few advances.

Although live-coding ideas are innovative and exciting, a majority of the approaches are looking at live programming from the "clean slate" perspective: They do not try to integrate live features into existing mainstream programming languages and IDEs. Even when they do, the ideas are often presented on "toy examples", with their application on large industrial systems being disputable.

Note that in reality, it is impractical to throw away existing systems, libraries, the knowledge of programmers and begin from scratch. Therefore, our vision is to gradually improve the experience of developers regarding the connection of source code and runtime in the existing languages and IDEs, without disrupting their current workflow.

We consider RuntimeSamp to be the first step toward our ambitious goal. After improving the object representation, we would like to focus on the data invalidation problem. Instead of deleting all data dependent on the changed parts, we would like to recompute them whenever possible. To prevent cognitive overload, showing only task-relevant runtime information will be necessary. Finally, sufficient performance improvements could make the approach suitable for industrial use.

References

[1] M. Sulír, "Integrating runtime metadata with source code to facilitate program comprehension," Ph.D. dissertation, Technical University of Košice, 2018. [Online]. Available: http://sulir.github.io/other/Thesis.pdf

[2] A. von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, Aug. 1995.

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[4] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 266–276.

[5] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, "Visual augmentation of source code editors: A systematic review," *Computer Languages, Systems & Structures*, 2018, submitted (arXiv:1804.02074).

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, Sep. 2009.

[7] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz, "Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks," *Software Engineering, IEEE Transactions on*, vol. 38, no. 3, pp. 579–591, May 2012.

[8] F. Beck, O. Moseler, S. Diehl, and G. Rey, "In situ understanding of performance bottlenecks through visually augmented code," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 63–72.

[9] M. Sulír and J. Porubän, "RuntimeSearch: Ctrl+F for a running program," in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 388–393.

[10] M. Sulír and J. Porubän, "Generating method documentation using concrete values from executions," in *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 3:1–3:13.

[11] M. Sulír and J. Porubän, "Augmenting source code lines with sample variable values," in *Proceedings of the 2018 26th IEEE/ACM International Conference on Program Comprehension (ICPC)*, May 2018.

[12] K. Damevski, D. Shepherd, and L. Pollock, "A field study of how developers locate features in source code," *Empirical Software Engineering*, vol. 21, no. 2, pp. 724–747, 2016.

[13] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 213–222.

[14] T. Roehm, "Two user perspectives in program comprehension: End users and developer users," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 129–139.

[15] J. Sillito, G. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 434–451, Jul. 2008.

[16] M. Sulír and J. Porubän, "Locating user interface concepts in source code," in *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 51. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 6:1–6:9.

[17] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016.

[18] S. Zhang, C. Zhang, and M. D. Ernst, "Automated documentation inference to explain failed tests," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 63–72.

[19] S. M. Sohan, C. Anslow, and F. Maurer, "SpyREST: Automated RESTful API documentation using an HTTP proxy server," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 271–276.

[20] M. Sulír and J. Porubän, "Source code documentation generation using program execution," *Information*, vol. 8, no. 4, p. 148, 2017.

[21] M. Sulír and J. Porubän, "Labeling source code with metadata: A survey and taxonomy," in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2017, pp. 721–729.

[22] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf, "Visual monitoring of numeric variables embedded in source code," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sep. 2013, pp. 1–4.

[23] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger Canvas: Industrial experience with the Code Bubbles paradigm," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1064–1073.

[24] J.-P. Krämer, J. Kurz, T. Karrer, and J. Borchers, "How live coding affects developers' coding behavior," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Jul. 2014, pp. 5–8.

[25] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Tralfamadore: Unifying source code and execution experience," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 199–204.

[26] A. Bradley, "IDE integration for execution mining data," University of British Columbia, Tech. Rep. CPSC 538W Final Project Report, Apr. 2010. [Online]. Available: http://www.cs.ubc.ca/~awjb/pubs/2010-cs538w-final-report.pdf

[27] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Lan-*

guages, and Applications. New York, NY, USA: ACM, 2006, pp. 169–190.

[28] M. Sulír, M. Nosáľ, and J. Porubän, "Recording concerns in source code using annotations," *Computer Languages, Systems & Structures*, vol. 46, pp. 44–65, Nov. 2016.

[29] M. Sulír and J. Porubän, "A quantitative study of Java software buildability," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2016. New York, NY, USA: ACM, 2016, pp. 17–25.

[30] A. Zeller and D. Lütkehaus, "DDD—a free graphical front-end for UNIX debuggers," *ACM SIGPLAN Notices*, vol. 31, no. 1, pp. 22–27, Jan. 1996.

[31] A. Chiş, O. Nierstrasz, A. Syrel, and T. Gîrba, "The moldable inspector," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 44–60.

[32] S. McDirmid, "Living it up with a live programming language," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 623–638.

[33] A. Sorensen and H. Gardner, "Programming with time: Cyber-physical programming with Impromptu," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 822–834.

[34] B. Swift, A. Sorensen, H. Gardner, and J. Hosking, "Visual code annotations for cyberphysical programming," in *Live Programming (LIVE), 2013 1st International Workshop on*, May 2013, pp. 27–30.

[35] T. Imai, H. Masuhara, and T. Aotani, "Making live programming practical by bridging the gap between trial-and-error development and unit testing," in *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, ser. SPLASH Companion 2015. New York, NY, USA: ACM, 2015, pp. 11–12.