# Towards Efficient Software Comprehension with Code Analysis

Robert Husák[1][a]
[1]*Faculty of Mathematics and Physics, Charles University, Ke Karlovu 3, Prague, Czech Republic*
*husak@ksi.mff.cuni.cz*

## 1 RESEARCH PROBLEM

As modern software systems are becoming more and more complex, the difficulty of their development increases as well. In order to maintain the development complexity, it is crucial to help software developers perform their tasks efficiently and correctly (Thomson, 2021; Nadeem, 2021). While developers take part in multiple different tasks, studies have shown that the most challenging problem is the comprehension of existing code (LaToza et al., 2006; Meyer et al., 2017). Xia et al. have discovered that developers spend as much as 58% of their time on program comprehension activities. The study of LaToza et al. shows that the knowledge of different parts of the code is distributed between multiple teams and developers. Therefore, when a developer works on a task spanning multiple code parts, other colleagues are often asked to share their related knowledge, causing interruptions and fragmentation of their work.

Development teams usually tackle these problems by adjusting the development process. For example, if developers ask too often about the same part of the system, it may be efficient to explicitly capture its most essential features into written documentation. To reduce the influence of work interruptions, developers can plan explicit meetings, e.g., morning standup meetings. When applied successfully, these activities can indeed make software development more efficient. However, they still bring non-negligible overhead, such as the maintenance of the documentation or extra time regularly spent on scheduled meetings.

A possibly more efficient way to help developers with program comprehension might be to equip them with better tools which empower them to distribute their knowledge more easily. For example, it is possible to create novel integrated development environments (IDE) which utilise the recent trends from the field of human-computer interaction (HCI) (Chiş et al., 2015; Adeli et al., 2020). However, these tools usually originate from academia and rarely gain any significant adoption in practice. Also, most of them do not use automated code analysis techniques, leaving the actual code exploration and reasoning to developers themselves.

On the other hand, there is a plethora of automated code analysis techniques, for example, data-flow analysis, abstract interpretation, constraint-based analysis or symbolic execution. The academic community is very active in developing these techniques and creating new ones. Regrettably, the research is usually focused mainly on their technical qualities and not on the improvement of their practical usability by a wide audience (Nadeem, 2021). As a result, their adoption in practice is limited to specific domains where the increase in code quality justifies the time investment, e.g. in embedded software development.

My research aims to discover whether it is possible to combine HCI and code analysis knowledge to create practically usable program comprehension tools.

## 2 OUTLINE OF OBJECTIVES

In order to consider the research successful, it must fulfil two objectives:

First, it is crucial to identify the most promising code analysis techniques usable for code comprehension. In general, precise techniques are usually not very scalable (e.g. symbolic execution) and vice versa (e.g. data-flow analysis). Both kinds of techniques might be helpful due to the way how developers usually approach code comprehension (Sillito et al., 2008). Initially, a developer usually needs to find *focus points*, i.e. the places in the code most relevant to the given development task. This phase demands using a technique that can efficiently analyse the whole potentially large system to display dependencies between its modules and help the developers find all the relevant source files. Afterwards, the developer inspects *subgraphs* formed by the focus points, examining their detailed semantics and behaviour. A more complex code analysis technique can be used for this task, taking advantage of the limited scope and providing more detailed information about the system.

[a] https://orcid.org/0000-0001-7128-6163

The second objective is to combine the most promising code analysis techniques with a suitable user interface (UI) for a developer-facing tool. This task is challenging due to two conflicting factors. On the one hand, the UI must be simple to use so that it does not drain too much mental energy from developers, which would likely hinder their effort in program comprehension itself. On the other hand, the nature of program comprehension tasks varies a lot, requiring the UI to be highly customisable. To find the reasonable balance between the complexity and capability of the UI, it might be helpful to follow best practices in user experience (UX) and HCI.

There are two levels of the second objective: conceptual and technical. The conceptual level focuses on identifying the users' needs and discovering the essential program comprehension scenarios. The technical level concerns the selection techniques, their combination, and the tool's implementation to cover the scenarios.

## 3 STATE OF THE ART

Multiple studies thoroughly analysed the difficulty of program comprehension tasks. Some of the studies focus on the overall analysis of developers' work (LaToza et al., 2006; Meyer et al., 2017). In contrast, others target directly the problem of program comprehension itself (Maalej et al., 2014; Xia et al., 2018) or the questions programmers often ask during the process (Sillito et al., 2008; LaToza and Myers, 2010).

Another essential motivation behind my research is the importance of using code analysis tools in practice, and voices from practice confirm this view. The articles from two GitHub senior software engineers state the significance of static analysis tools for the development of complex software (Thomson, 2021) and emphasise the need to develop these tools with HCI in mind (Nadeem, 2021).

In the field of HCI, several interesting tools have been created to help developers with code comprehension, for example: Reacher (LaToza and Myers, 2011) provides a simple way to ask interprocedural reachability questions within a codebase. Python Tuton (Guo, 2021) visualises the execution of short snippets of code, helping people to learn to program. Synectic (Adeli et al., 2020) is an integrated development environment (IDE) with a spatially-oriented interface which enables to easily organise source files, notes and documentation when working on a specific task. Moldable Development (Chiş et al., 2015) is an alternative way of programming which places importance on creating custom developer tools throughout the whole software development process.

As we can see, while the mentioned tools use modern trends in HCI to achieve good usability, they rarely employ advanced code analysis techniques. There is a plethora of them available, and each one provides a different level of scalability and precision. The most precise techniques are usually not very scalable and vice versa. For example, data-flow analysis (Kildall, 1973; Kam and Ullman, 1977) can be used on large programs during compilation, but its result is over-approximated and not very detailed. On the other hand, symbolic execution (King, 1976) and model checking (Clarke et al., 1986) can reason very precisely about program semantics, but they are practically usable only for embedded systems and simple programs.

Program comprehension is also tackled in requirements engineering and conceptual modelling (Insfrán et al., 2002). Instead of reasoning about the actual software system, it is possible to create a model capturing the system's relevant characteristics and inspect the model. Recent studies show that modelling can simplify many software development and maintenance tasks. RoboSMi (Wood et al., 2020) combines static analysis and model-driven engineering (MDE) to simplify the migration of robotic software between hardware platforms. Other articles show, e.g., how to automate the detection of security flaws (Tuma et al., 2020), ensure transparency in machine learning systems (Nolte and Kaczmarek-Heß, 2022) or inspect systems during runtime (Sakizloglou et al., 2020). Although many software engineers use unified modelling language (UML) diagrams to describe high-level characteristics of software systems, full-fledged adoption of MDE among software development companies is rare (Whittle et al., 2017). Therefore, models developed alongside software systems usually serve only for communication and documentation purposes instead of automated reasoning.

## 4 METHODOLOGY

Extensive research of existing code analysis techniques is needed to fulfil the first objective, followed by serious engagement with the selected ones. The engagement should take the form of creating or improving tools which use these techniques. To verify the contribution of my work on each tool, automated tests of performance and precision can be performed. This way, I can both gain insight into the techniques and create added value for the scientific community, possibly enhancing the techniques to better fit in solving the overall research problem.

The second objective cannot be efficiently performed without involving target users. Therefore, a pilot tool for program comprehension will be developed in collaboration with at least one software company. Based on the needs of day-to-day developers, it will be possible to design the tool's features to cover the crucial program comprehension scenarios. The feedback from developers will enable me to incrementally improve the tool and make sure it solves their problems.

A quantitative study with developers will be performed to objectively measure whether the discovered approach indeed helps developers with program comprehension. For the study, a set of code comprehension tasks will be created. The participants will be randomly separated into two groups where the first one will get acquainted with the pilot tool, and the members of the second one will rely only on their experience. Apart from the objectively measured speed and correctness, each participant will also fill out a questionnaire with subjective questions. This way, it will be possible to assess how the tool affects developer productivity and whether it is convenient to use. Regarding statistical methods and other details, the study will be inspired by existing studies performed by authors of different tools (LaToza et al., 2006; Xia et al., 2018).

Eventually, a long-term study with a company might be even more valuable, as it could show the effect of the tool in more realistic settings. However, it would be probably more demanding for the quality of the pilot tool, requiring it to be close to production grade. Therefore, this ambition will likely not fit within the scope of the Ph.D. research.

## 5 EXPECTED OUTCOME

After the research objectives are fulfilled, we will know how to improve the software comprehension process by employing automated code analysis techniques and what UI they should use. This step will pave the way for the future development of related methods and tools in academia and industrial practice.

These projects might foster tighter collaboration between research teams and companies from a long-term perspective. An ideal state would be to gather a community of both industrial developers and computer scientists. In this community, people from both of these parts may cooperate, utilising various techniques to improve program comprehension as well as other parts of the software development process. Creating the community brings several complex challenges on its own and is out of the Ph.D. research scope.

## 6 STAGE OF THE RESEARCH

The first research objective has been successfully fulfilled, and the work on the second one is currently in progress.

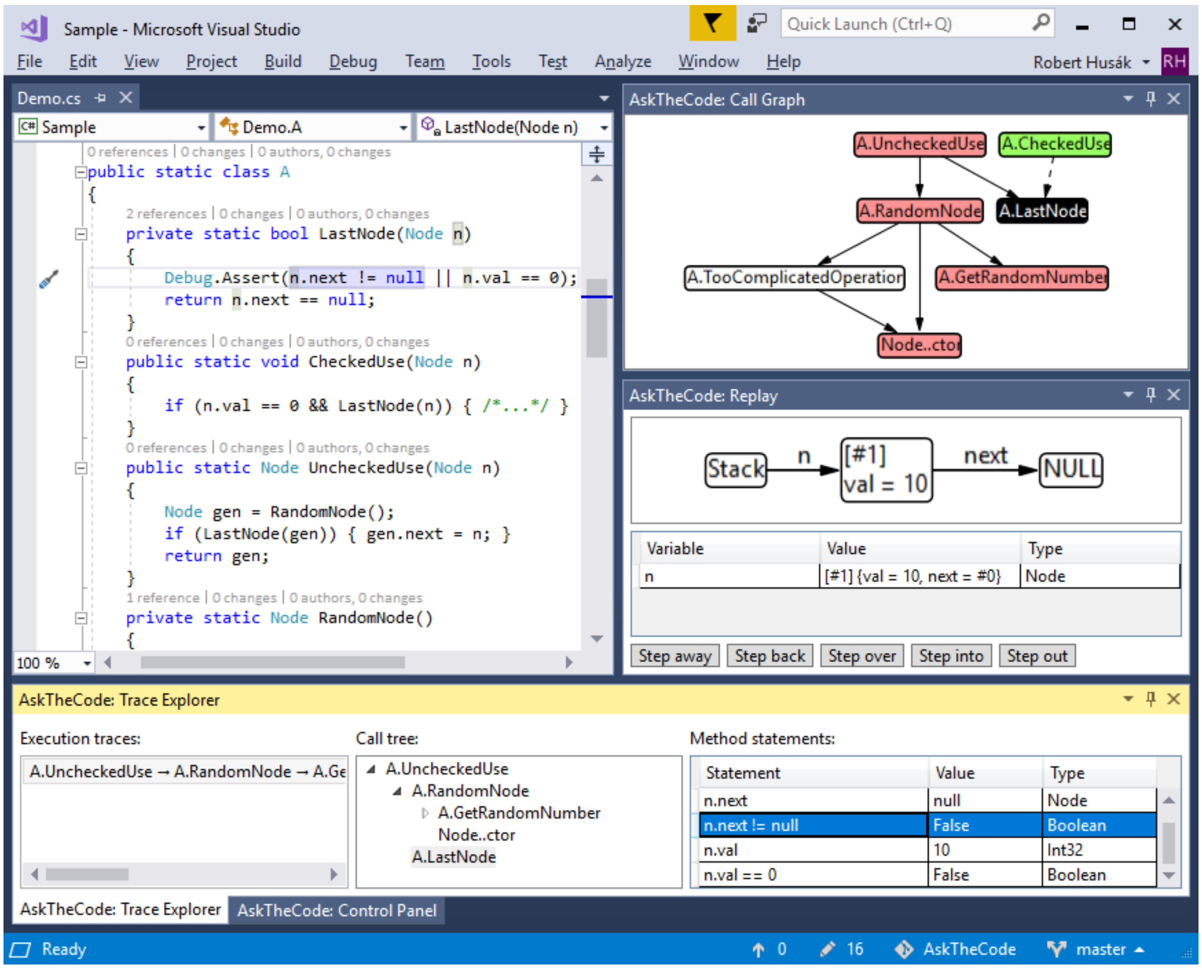
As the first technique to be used in a program comprehension tool, I selected symbolic execution (King, 1976). Its main benefit is the ability to reason about code semantics very precisely. Using symbolic execution, many complicated questions about program behaviour can be formulated as reachability problems. For example, to check whether a variable `x` can contain the value `42`, the developer might write the following code:

```
if (x == 42) {
  place:
}
```

Afterwards, the reachability of the label `place` can be assessed via backward symbolic execution. Even simpler way might be to write `assert(x != 42)` in the code and let a tool automatically unwind this code to the previously shown `if` statement.

To validate whether symbolic execution can be used for program comprehension, AskTheCode[1] was created (Husák and Zavoral, 2019; Husák et al., 2019; Husák et al., 2020). It is a Microsoft Visual Studio extension which can analyse whether a selected place in the C# code is reachable from public methods. Its user interface is shown in Figure 1, notice that in this case it uses reachability to verify the validity of a `Debug.Assert` statement in the method `A.LastNode`. It is called by two public methods, `A.CheckedUse` and `A.UncheckedUse`. While the call from the former cannot break the assertion, the call from the latter can. AskTheCode discovered an example of a particular `A.UncheckedUse` input leading to this situation. Those values, as well as the whole execution trace, can be inspected by the user in the panels *Trace Explorer* and *Replay*. The panel *Call Graph* provides a high-level overview of inspected methods and their relations. Different colours of method node backgrounds signalise the gathered information. Calls from green methods have been successfully verified not to break the assertion. Red methods take place in execution traces leading to assertion break. White methods were not appropriately analysed because they were too complicated.

[1]http://www.askthecode.net



```
0 references | 0 changes | 0 authors, 0 changes
public static class A
{
    2 references | 0 changes | 0 authors, 0 changes
    private static bool LastNode(Node n)
    {
        Debug.Assert(n.next != null || n.val == 0);
        return n.next == null;
    }
    0 references | 0 changes | 0 authors, 0 changes
    public static void CheckedUse(Node n)
    {
        if (n.val == 0 && LastNode(n)) { /*...*/ }
    }
    0 references | 0 changes | 0 authors, 0 changes
    public static Node UncheckedUse(Node n)
    {
        Node gen = RandomNode();
        if (LastNode(gen)) { gen.next = n; }
        return gen;
    }
    1 reference | 0 changes | 0 authors, 0 changes
    private static Node RandomNode()
    {
```

| Variable | Value | Type |
|---|---|---|
| n | [#1] {val = 10, next = #0} | Node |

Execution traces:

A.UncheckedUse → A.RandomNode → A.Ge

Call tree:

- A.UncheckedUse
  - A.RandomNode
    - A.GetRandomNumber
    - Node..ctor
  - A.LastNode

Method statements:

| Statement | Value | Type |
|---|---|---|
| n.next | null | Node |
| n.next != null | False | Boolean |
| n.val | 10 | Int32 |
| n.val == 0 | False | Boolean |

Figure 1: AskTheCode in Microsoft Visual Studio 2017.

AskTheCode uses Microsoft .NET Compiler Platform ("Roslyn")[2] to obtain semantic information from the source code, Z3 theorem prover (de Moura and Bjørner, 2008) to check the satisfiability of SMT formulas used within symbolic execution and Microsoft Automatic Graph Layout (MSAGL)[3] to visualise call graph and heap snapshot in the UI. These technologies are freely available and have proven to be very powerful. Without their foundation, creating any sophisticated tool for C# analysis would be considerably more challenging.

While AskTheCode can provide the user with detailed information about a small piece of code, its practical usage is blocked by poor scalability of symbolic execution. Therefore, it was necessary to find a complementary technique capable of reasoning about the whole program efficiently. With such a technique, it would be possible to efficiently demarcate the part of the code to run symbolic execution on.

For this purpose, data-flow analysis based on monotone frameworks and fixed point computation (Kildall, 1973; Kam and Ullman, 1977) was selected. Even its interprocedural variant is present in production-grade compilers, proving its ability to be used on large projects.

In order to deepen my knowledge in data-flow analysis, I joined the research and development of PeachPie[4], a compiler which produces Common Language Infrastructure (CLI) assemblies from PHP projects, effectively connecting the ecosystems of PHP and .NET (Misek and Zavoral, 2019). PeachPie uses interprocedural data-flow analysis to estimate types of variables and function return values. I focused on ways how to improve the type analysis by performing code transformation (Husák et al., 2020) and specialization (Husák et al., 2022).

This approach is highly relevant to program com-

[2]https://github.com/dotnet/roslyn

[3]https://github.com/microsoft/automatic-graph-layout

[4]https://www.peachpie.io

prehension because it enables simplifying pieces of code under a specific context. Consider the following PHP code:

```
function caller(array $a) {
  $no = callee($a);
  // ...
}

function callee($x) {
  if (is_array($x)) {
    return count($x);
  } else if ($x instanceof MyClass) {
    return $x->valueCount()
  } else {
    return null;
  }
}
```

The function `callee` behaves differently according to different types of its parameter `$x`. However, if the user is interested only in the semantics of `caller`, `$x` is guaranteed to be of the type `array`. Therefore, the user might be interested only in the following specialised version of `callee`:

```
function callee_array(array $x) {
  return count($x);
}
```

The ability to specialise and simplify code this way might be crucial for both the user and symbolic execution to reason about the program more efficiently.

Having identified symbolic execution and data-flow analysis as the techniques of choice, the focus of the research has moved towards their combination into a pilot tool. The tool is called Slicito[5], as it aims to enable developers to "slice" interesting code parts from a large software system. To ensure that Slicito eventually becomes useful for real-life scenarios, the plan is to develop it incrementally and cooperate with possible future users.

The first goal of the tool is to help developers with two specific scenarios inspired by a real-life cloud-based application:

- Find all places where a particular identifier is received as an argument of the application API and ensure its format is appropriately validated.
- Enrich several database entities with the information about which users own them. Then, ensure that when selecting or modifying an entity from this set, the currently logged-in user is always considered.

While Slicito should not be bound to the needs of a particular company or a product, having the support for these scenarios as the first goal helps to shape the set of supported features. Making the tool as general as possible while supporting these specific tasks is essential.

At the time of the writing, Slicito contains only a simple set of functions, enabling the user to analyse a project with Microsoft Roslyn and render interesting parts of the code to a graph using MSAGL. It is recommended to use it from a .NET Interactive[6] notebook. This way allows the user to query the same C# project for different kinds of information and immediately see the result in the form of a graph.

The graph is currently a static `.svg` image, but there are plans to introduce some degree of interactivity in the near future. For example, if the user clicks on a graph node representing a method, the corresponding file will be opened in Microsoft Visual Studio on the line where the method is defined. Gradually, even more interactivity will be added for scenarios where it proves to be useful.

The most challenging step will be the integration of code analysis techniques. At first, data-flow analysis will be added, providing the user with high-level information about the relations between different parts of the code. Secondly, symbolic execution will be implemented into Slicito. Since AskTheCode already contains most of the needed functionality, I expect to either reuse a lot of its code or keep it as a standalone tool and integrate it with Slicito via an API. Cooperation with colleagues both from academia and industry will be instrumental throughout the whole development process, including the planned quantitative study.

## ACKNOWLEDGEMENTS

I thank my supervisor Filip Zavoral, Ph.D. and my colleague doc. Jan Kofroň, Ph.D. for their help and support during my Ph.D. research. This work has been supported by the Czech Science Foundation project no. 20-07487S, Charles University Grant Agency (GA UK) project no. 896120, the project PROGRESS Q48 and the grant SVV-260588.

## REFERENCES

Adeli, M., Nelson, N., Chattopadhyay, S., Coffey, H., Henley, A., and Sarma, A. (2020). Supporting code comprehension via annotations: Right information at the right time and place. In *2020 IEEE Symposium*

[5]https://github.com/roberthusak/slicito

[6]https://github.com/dotnet/interactive

*on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–10.

Chiş, A., Nierstrasz, O., and Gîrba, T. (2015). Towards moldable development tools. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, page 25–26, New York, NY, USA. Association for Computing Machinery.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263.

de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

Guo, P. (2021). Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 1235–1251, New York, NY, USA. Association for Computing Machinery.

Husák, R., Kofroň, J., and Zavoral, F. (2019). Askthecode: Interactive call graph exploration for error fixing and prevention. *Electronic Communications of the EASST*, 77.

Husák, R., Kofroň, J., and Zavoral, F. (2020). Handling heap data structures in backward symbolic execution. In *Formal Methods. FM 2019 International Workshops*, pages 537–556, Cham. Springer International Publishing.

Husák, R., Kofroň, J., Míšek, J., and Zavoral, F. (2022). Using procedure cloning for performance optimization of compiled dynamic languages. In *In Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*.

Husák, R. and Zavoral, F. (2019). Source code assertion verification using backward symbolic execution. *AIP Conference Proceedings*, 2116(1):350004.

Husák, R., Zavoral, F., and Kofroň, J. (2020). Optimizing transformations of dynamic languages compiled to intermediate representations. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 145–152.

Insfrán, E., Pastor, O., and Wieringa, R. (2002). Requirements engineering-based conceptual modelling. *Requirements engineering*, 7(2):61–72.

Kam, J. B. and Ullman, J. D. (1977). Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–317.

Kildall, G. A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA. Association for Computing Machinery.

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

LaToza, T. D. and Myers, B. A. (2010). Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, New York, NY, USA. Association for Computing Machinery.

LaToza, T. D. and Myers, B. A. (2011). Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 117–124.

LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 492–501, New York, NY, USA. Association for Computing Machinery.

Maalej, W., Tiarks, R., Roehm, T., and Koschke, R. (2014). On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4).

Meyer, A. N., Barton, L. E., Murphy, G. C., Zimmermann, T., and Fritz, T. (2017). The work life of developers: Activities, switches and perceived productivity. *IEEE Transactions on Software Engineering*, 43(12):1178–1193.

Misek, J. and Zavoral, F. (2019). Semantic analysis of ambiguous types in dynamic languages. *Journal of Ambient Intelligence and Humanized Computing*, 10.

Nadeem, A. (2021). Human-centered approach to static-analysis-driven developer tools: The future depends on good hci. *Queue*, 19(4):68–95.

Nolte, M. and Kaczmarek-Heß, M. (2022). Enterprise modeling in support of transparency in the design and use of software systems. In Augusto, A., Gill, A., Bork, D., Nurcan, S., Reinhartz-Berger, I., and Schmidt, R., editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 157–172, Cham. Springer International Publishing.

Sakizloglou, L., Ghahremani, S., Barkowsky, M., and Giese, H. (2020). A scalable querying scheme for memory-efficient runtime models with history. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 175–186, New York, NY, USA. Association for Computing Machinery.

Sillito, J., Murphy, G. C., and De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451.

Thomson, P. (2021). Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue*, 19(4):29–41.

Tuma, K., Sion, L., Scandariato, R., and Yskout, K. (2020). Automating the early detection of security design flaws. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 332–342, New York, NY, USA. Association for Computing Machinery.

Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., and Heldal, R. (2017). A taxonomy of tool-related issues affecting the adoption of model-driven engineering. *Software & Systems Modeling*, 16(2):313–331.

Wood, S., Matragkas, N., Kolovos, D., Paige, R., and Gerasimou, S. (2020). Supporting robotic software migration using static analysis and model-driven engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 154–164, New York, NY, USA. Association for Computing Machinery.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., and Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976.