The Journal of Systems and Software 211 (2024) 111980

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Property probes: Live exploration of program analysis results[☆]

Anton Risberg Alaküla [a,*], Görel Hedin [a], Niklas Fors [a], Adrian Pop [b]

[a] Lund University, Ole Römers Väg 3, 221 00, Lund, Sweden
[b] Linköping University, Mäster Mattias väg, 581 83, Linköping, Sweden

ARTICLE INFO

Dataset link: https://github.com/lu-cs-sde/codeprober, https://zenodo.org/doi/10.5281/zenodo.10607231

Keywords:
Program analysis
Debugging
Property probes
CodeProber

ABSTRACT

We present *property probes*, a mechanism for helping a developer explore partial program analysis results in terms of the source program interactively while the program is edited. A node locator data structure is introduced that maps between source code spans and program representation nodes, and that helps identify probed nodes in a robust way, after modifications to the source code. We have developed a client–server based tool CodeProber supporting property probes, and argue that it is very helpful in debugging and understanding program analyses. We have evaluated our tool on several languages and analyses, including a full Java compiler and a tool for intraprocedural dataflow analysis. Our performance results show that the probe overhead is negligible even when analyzing large projects.

1. Introduction

Modern software tooling includes many kinds of program analysis. For instance, compilers do type analysis, IDEs support type-based navigation and editing, and bug-finding tools may use analyses based on dataflow and effects. However, developing new analyses can be difficult. There are often many subanalyses, and they might need to handle many corner cases of the analyzed language.

In this paper we propose a new interactive mechanism, *property probes*, to help the analysis developer. The main idea is to allow the developer to inspect and display *properties*, i.e., (partial) analysis results tied to specific parts of an editable source code (as plain text). Examples of properties include name bindings, types, generated code, propagated constant values, control-flow edges, and data flow properties like live variables. The developer can interactively explore analyses by creating probes for different program elements, and the results are updated live as the source code is edited, and even after updates to the analysis tool itself.

It is a challenge to match a probe for a particular property to the corresponding program element after source code changes, and we provide a robust algorithm for this purpose. Our approach also supports the probing of properties of implicit program elements that are not directly visible in the edited source code, e.g., imported libraries or predefined elements built into the programming language, like `class Object` in Java. We see property probes as a complement to traditional development support such as automated tests and traditional breakpoint/step-debuggers or “print debugging”.

We have implemented a property probe tool, CodeProber, specifically targeting analyses built with Reference Attribute Grammars (RAGs) (Hedin, 2000). The attributes of an attribute grammar match the probed properties, and the interactive probing fits the demand evaluation used in RAGs. However, the concept of property probes can in principle be applied to any analysis that uses an abstract syntax tree (AST) as the spanning tree over its program representation, and that associates partial analysis results with nodes of the tree. We therefore expect the ideas to be useful for analyses built with a much wider range of approaches than RAGs.

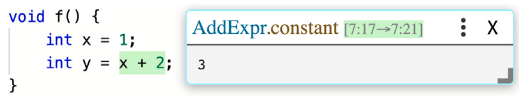
We have applied CodeProber to a number of different languages and analyses implemented using the JastAdd metacompiler (Ekman and Hedin, 2007b) that supports RAGs and demand evaluation. In particular, we have applied it to ExtendJ (Ekman and Hedin, 2007a), a full Java compiler, and to IntraJ (Riouak et al., 2021), an extension of ExtendJ that supports intraprocedural control-flow and dataflow analysis. Furthermore, we have used CodeProber in a course on compiler construction and in a course on program analysis.

Our contributions are as follows:

- We introduce the concept of property probes (Section 2).
- We present the CodeProber tool and the different kinds of property probes it supports (Section 3).
- We present a data structure *node locator*, used for robust identification of the probed AST nodes after changes of the source code (Section 4).

☆ Editor: Aldeida Aleti.
* Corresponding author.
E-mail addresses: anton.risberg_alakula@cs.lth.se (A. Risberg Alaküla), gorel.hedin@cs.lth.se (G. Hedin), niklas.fors@cs.lth.se (N. Fors), adrian.pop@liu.se (A. Pop).

https://doi.org/10.1016/j.jss.2024.111980
Received 10 March 2023; Received in revised form 18 October 2023; Accepted 19 January 2024
Available online 1 February 2024



```
void f() {
    int x = 1;
    int y = x + 2;
}
```

AddExpr.constant [7:17→7:21] ⋮ X

```
3
```

Fig. 1. Probe for the compile-time `constant` property of a selected addition expression.

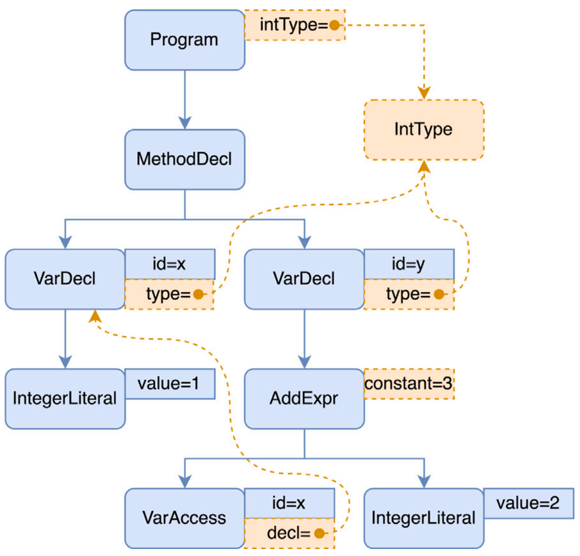


Fig. 2. AST with intrinsic (solid blue) and computed (dashed orange) properties. `IntType` is a synthetic AST node. Straight lines are child edges, and curved lines are references.

- We present the algorithms that are needed for implementing node locators. We also present optimizations for improving performance and user experience when using node locators in larger projects (Section 5).
- We present the architecture of CodeProber, and the requirements an AST must satisfy to be used with CodeProber (Section 6).
- We present experiences from using CodeProber in case studies (Section 7), and performance measurements to explore its limitations with regards to the size of the edited code and number of active probes (Section 8).

Finally, we present related work in Section 9, and then conclude in Section 10.

This paper is an extension of a previous conference paper at SLE 2022 (Risberg Alaküla et al., 2022). Major additions include detailed descriptions of CodeProber (Section 3), algorithms (Section 5), and more extensive case studies (Section 7).

2. Property probes

In this section, we explain what we mean by *properties* and present the concept of a *property probe*. We also briefly discuss different use cases for property probes.

2.1. Properties

We use the term *property* for a named compile-time value associated with an AST node, and computed by some compile-time (static) analysis. For example, an addition expression could have a property `int constant`, holding the compile-time integer value of the expression, as computed using constant propagation. Fig. 1 exemplifies this. Here, an addition expression x+2 has been selected, identifying an AST node of type `AddExpr`. Its property `constant` is shown, having the value 3.

We distinguish between *intrinsic* and *computed* properties. An *intrinsic* property is part of the AST constructed by a parser or an editor, and is directly available without further computation. Examples include token values like variable names and literal values. A *computed* property is computed by an analysis of the AST. Examples include name bindings, types, and the `constant` property mentioned above. A computed property may be *higher-order* in that its value can be a fresh AST subtree whose nodes may have their own properties. We refer to such computed AST nodes as being *synthetic*. An example is an AST node for a primitive type that is not present in the parsed program, but that is useful to reify as an AST node.

Fig. 2 illustrates a (slightly simplified) AST with intrinsic and computed properties for the example in Fig. 1. Here, intrinsic properties include the `id` properties of variable declarations and accesses, and the `value` property of an integer literal. Computed properties include `decl`, `type`, `constant`, and `intType`. Here, the `decl` of a variable access is a reference to an AST node representing the declaration, `type` is a reference to an AST node representing a type, `constant` is the constant value of an expression as discussed above, and `intType` is a reference to a synthetic AST node representing the primitive type `IntType`.

Optionally, a property can take arguments, i.e., serve as a function on a given AST node. An example could be a property `boolean visible(String id)` for a statement node, that returns *true* if there is a declaration named `id` visible at the position of the statement.

Different compiler and static analysis tools use different strategies to compute property values. For tools built using RAGs, the properties are computed on demand: if a client asks for a particular property of a particular node, that property will automatically be evaluated, and any other properties it depends on will be recursively evaluated, memoizing subresults for efficiency. After an edit, the memoized values can be thrown away and new results are recomputed when the values are asked for the next time. This often gives a very short response time after an edit, but the time of course depends on what property is being asked for. More traditional tools are often pass-oriented, traversing the complete program several times, computing all properties of all AST nodes. After an edit, the whole computation needs to be redone, thus giving long response times. We do not prescribe any particular strategy to be used, but assume that there is an automatic way to trigger computation of the properties, so that their up-to-date values can be presented.

2.2. Property probes

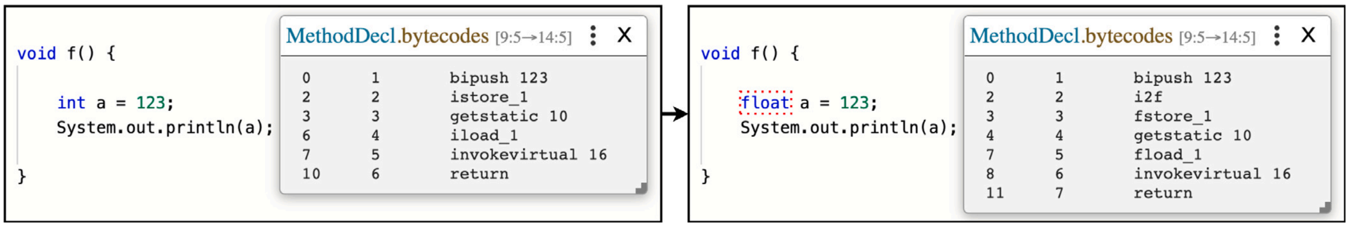
A property probe is an interactive element, presented in the context of a source code text editor, and acting as a live observer of a property of an AST node. In CodeProber, each property probe is displayed as a small window. An example can be seen in Fig. 1.

Internally, a probe is represented by a *node locator* (a way of identifying a particular node in the AST), a *property name*, optionally a number of *arguments* (if the property takes arguments), and a *result value*, i.e., the most recently computed value of the property. The result value is a collection of primitive values, like string or integer, and AST node references (represented as node locators).

A probe has two main responsibilities:

1. It adjusts the node locator after source code edits.
2. It presents up-to-date results to the user, reevaluating the property as needed.

Evaluating a property means, in practical terms, invoking a function in the context of an AST node. Keeping the result up-to-date means re-invoking the same function whenever needed, such as when the source code is modified. Fig. 3 shows an example in CodeProber of how a probe is updated when the user edits the source code. In this case, the user has created a probe for the bytecode of a method. When the user edits the source code, changing the type of the variable from `int` to `float`,



```
void f() {
    int a = 123;
    System.out.println(a);
}
```

MethodDecl.bytecodes [9:5→14:5]

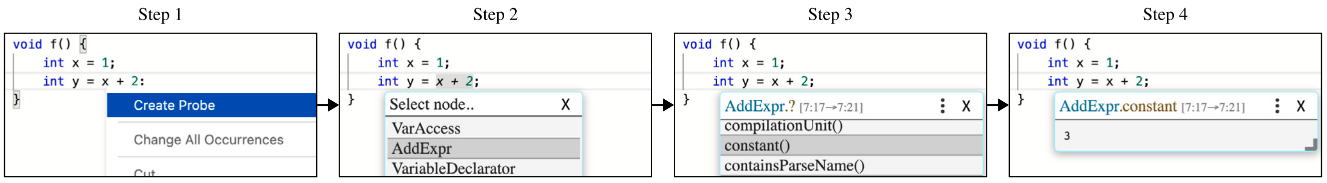
```
0     1     bipush 123
2     2     istore_1
3     3     getstatic 10
6     4     iload_1
7     5     invokevirtual 16
10    6     return
```

```
void f() {
    float a = 123;
    System.out.println(a);
}
```

MethodDecl.bytecodes [9:5→14:5]

```
0     1     bipush 123
2     2     i2f
3     3     fstore_1
4     4     getstatic 10
7     5     fload_1
8     6     invokevirtual 16
11    7     return
```

Fig. 3. Probe for the bytecode of a method. The probe result is automatically updated after editing the type from `int` to `float` (dotted box).



```
void f() {
    int x = 1;
    int y = x + 2;
}
```

(1) The user right-clicks on the addition expression in the source text and selects ”Create Probe”.

(2) The AST nodes that span the clicked position appear in a menu, and the user selects the desired node, `AddExpr` in this case.

(3) The user is presented with a menu of all available properties on the `AddExpr` node, and selects the `constant` property.

(4) Finally, the property probe window appears, showing the result value, in this case the number 3.

Fig. 4. Steps to create a probe for the `constant` property of the addition expression `x + 2`.

the bytecode in the probe result is updated, for example, changing the `istore_1` and `iload_1` instructions to `fstore_1` and `fload_1`.

The user can create a probe starting from a location in the text, selecting the desired AST node in case several nodes match the same location. It is also possible for a user to create a new probe starting from the result of another probe. This allows property exploration not only directly related to the edited text, but also by exploring probe results, which can again be explored further, supporting an interactive way of investigating partial results of an analysis.

Exploration of probe results opens for exploring properties of nodes that have no matching location in the edited source text. One example is nodes corresponding to ASTs of imported libraries. Another example is synthetic nodes created to represent implicit program entities. Examples include built-in types like the `IntType` in Fig. 2 or `class Object` in Java, desugared representations of language constructs, and computed complex properties resulting from a static analysis. Attribute grammars can use higher-order attributes for computing synthetic nodes. A higher-order attribute is an attribute whose value is a new AST node subtree, and where these new nodes may themselves have attributes (Vogt et al., 1989).

2.3. Use cases

Property probes help understanding the inner data structures of compilers and program analysis tools through interactive exploration. We see a variety of situations when property probes can be useful, both in education and for production work. In education, they can help students understand core compiler concepts such as ASTs, name bindings and type checking, as well as program analysis concepts like control flow and dataflow analysis. In extending an existing compiler, for example to extend the supported language, property probes can be useful for understanding the existing functionality and internal APIs. For program analysis tools, there are similar use cases, when constructing a new analysis that build on existing ones. When there are bugs in a tool, property probes can be used for interactively pinpointing what computations are correct and which are faulty. Furthermore, property probes can be useful for prototyping interactive language services, like code completion, semantic navigation, etc., by expressing the service data as computed properties.

Property probes do not replace ordinary control-flow focused debugging that uses step/breakpoints or print statements, since they do not address the order in which properties are computed. However, they provide a useful complement to such debugging, and might reduce the need for it.

We have used property probes in two university courses and in development of compilers and static analyzers, and have had overall positive results. This is discussed further in Section 7.

2.4. Tool architecture

To implement property probes for a specific analysis tool, we propose a client–server architecture, see Fig. 14. The client side uses a customizable code editor such as Monaco[1] or similar. The server side consists of two components; a server and an analysis tool.

It is the responsibility of the analysis tool to parse the edited text into an AST (as well as any other files needed for the analysis), and to populate the AST with the functionality to be explored using probes. The server uses an API to access the analysis tool, e.g., to get the root of the current AST and to query properties on different AST nodes. The server also handles the communication with the client-side code editor.

The probes are stored on the client-side, using node locators as references to node objects. This allows the AST to be reparsed at any time, or even the analysis tool to be restarted from scratch during the editing session. It also allows the server to be completely stateless.

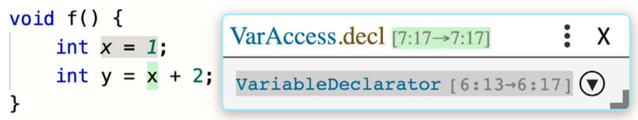
3. CodeProber

CodeProber supports property probes on an underlying compiler or analysis tool. In this section, we present its key features, using the ExtendJ Java compiler as the underlying tool.

3.1. Creating a probe

The user can create a probe interactively via CodeProber’s text editor. Fig. 4 shows an example. The user right clicks in the source code and selects the menu option “Create Probe” (1). A list appears, showing AST nodes that overlap with the clicked location. The user selects one of them (2). A new list appears, showing all properties available on the selected AST node. The user selects a property (3). A

[1] https://microsoft.github.io/monaco-editor/. Accessed February 1, 2024.

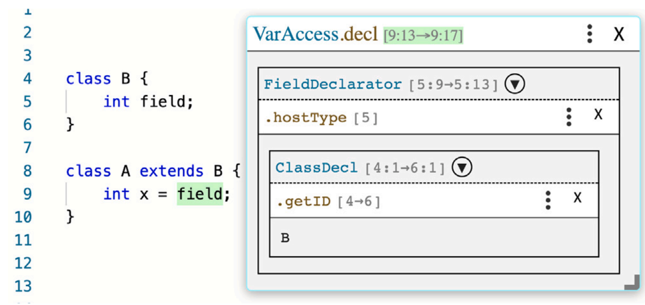


```
void f() {
    int x = 1;
    int y = x + 2;
}
```

VarAccess.decl [7:17→7:17]

VariableDeclarator [6:13→6:17]

Fig. 5. Example of a reference result. The `decl` property of the variable `x` has been probed, resulting in a reference to a `VariableDeclarator` node. The user hovers the result, causing the corresponding span in the source code (`x = 1`) to be highlighted in gray.



```
2
3
4   class B {
5       int field;
6   }
7
8   class A extends B {
9       int x = field;
10  }
11
12
13
```

VarAccess.decl [9:13→9:17]

FieldDeclarator [5:9→5:13]

.hostType [5]

ClassDecl [4:1→6:1]

.getID [4→6]

B

Fig. 6. Nested probes to show what class the `field` variable is declared in. The user has first created a probe for `field`'s property `decl`, resulting in a `FieldDeclarator`. Then a nested probe was created, showing the `FieldDeclarator`'s `hostType` property, resulting in a `ClassDecl`. Finally, the user has probed its name (the `getID` property), which is `B`.

window (property probe) appears which shows the result of evaluating the selected property on the selected node (4). The user can click the position indicator ([`7:17→7:21`]) to get the same green highlighting as displayed in Fig. 1.

3.2. Advanced probes

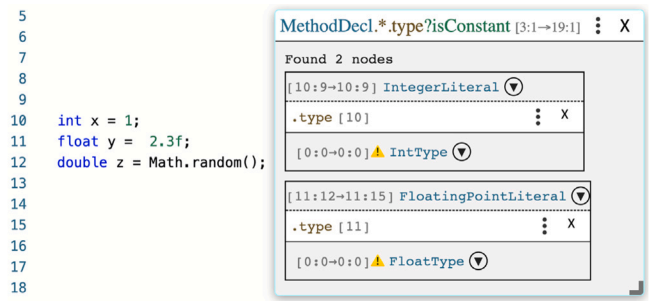
The probe window created in Fig. 4 is simple in that it shows a single property for a single AST node, and the probe result is displayed as plain text (the value 3). Based on our experience from using the tool, we have developed support for several more advanced kinds of probes: *nested probes* (local probes created from probe results), *search probes* (showing a set of properties), *AST probes* (visual presentation of the AST), and *probes contributing diagnostics* (that is shown in the source text). We will now discuss these in turn.

3.2.1. References and nested probes

A probe result can be a reference to an AST node. The user then can hover over the result to see the corresponding source text for that node. Fig. 5 shows an example where the `decl` property of the variable access `x` is a reference to a `VariableDeclarator` node, highlighted in gray.

The user can explore a result reference by clicking on it to create a new probe for the referenced node. This can be done iteratively to follow chains of AST node references. Instead of creating a new window for the new probe, it can be nested inside the original probe. In addition to saving screen space, this retains the link between the probes, so that an inner probe will be re-evaluated on the result of the outer one when the user edits the program.

Fig. 6 shows an example where a chain of nested probes is created to show what class the instance variable `field` is declared in. Starting at the AST node v for the `field` variable, the class name can be accessed via the chain of calls v`.decl().hostType().getID()`. The user has first created a probe for the `decl` property for the `field` variable, then a nested probe for `hostType`, and finally another nested probe for `getID`. If the user were to change `field` on line 9 to another variable, that would cause the nested probes to display potentially different values.



```
int x = 1;
float y =  2.3f;
double z = Math.random();
```

MethodDecl.*.type?isConstant [3:1→19:1]

Found 2 nodes

[10:9→10:9] IntegerLiteral

.type [10]

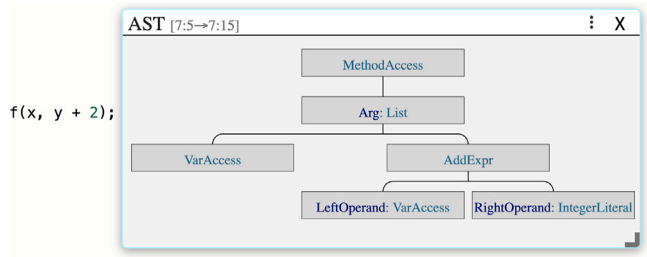
[0:0→0:0]⚠ IntType

[11:12→11:15] FloatingPointLiteral

.type [11]

[0:0→0:0]⚠ FloatType

Fig. 7. Search probe that finds all nodes under a `MethodDecl` where `isConstant` is true, and opens a nested probe for `type` on them.



```
f(x, y + 2);
```

Fig. 8. AST probe showing the AST of a method call. More probes can be created by clicking on individual nodes in the AST.

3.2.2. Search probes

The user may want to see a given property for multiple AST nodes. CodeProber supports this through *search probes*. A search probe lets the user specify a query to find all AST nodes that pass a given filter. Fig. 7 shows an example. Here, the query `MethodDecl.*.type?isConstant` selects all AST nodes in the subtree of a given method declaration for which the property `isConstant` is true, selecting two nodes in this case. Furthermore, the query includes the `type` property, which is shown as nested probes. If desired, the user can then investigate details of the probe results, using more nested probes.

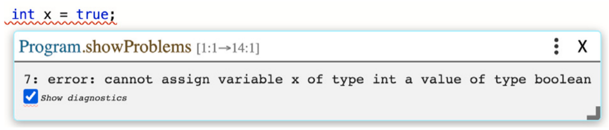
Search queries do not have to specify both a property and a filter. For example, the query `*.type` on the root node would show the `type` property for all nodes in the whole AST. The query `?isConstant` would select all nodes that are constant, but not evaluate any property on them.

3.2.3. AST probes

To select what nodes to create probes for, the user needs an understanding of the AST structure and the different AST node types used in the compiler or program analysis tool. To support this, CodeProber provides *AST probes* where the AST is rendered graphically. Fig. 8 shows an example, showing the AST for the method call `f(x, y + 2)`. The nodes can be hovered to highlight their corresponding span in the source code. They can also be clicked to create new probes. This can be useful when learning about the internal structure of a compiler.

3.2.4. Probes contributing diagnostics

A probe can be used for contributing diagnostics information that is displayed directly in the text editor, for example, squiggly lines, arrows, and hovering behavior. This can be used for prototyping language service support. Fig. 9 shows an example of a probe for `showProblems`, a property holding a set of error messages. The error messages are displayed in the probe. In addition, each error message object contains information about where to place squiggly lines in the text editor. The visibility of the diagnostics can be toggled on and off with a checkbox (*Show diagnostics*).

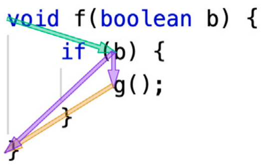


```
int x = true;
```

Program.showProblems [1:1→14:1]

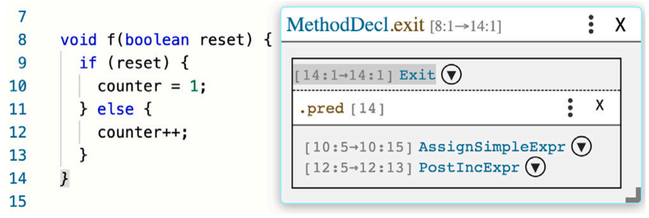
```
7: error: cannot assign variable x of type int a value of type boolean
```

Fig. 9. Squiggly line diagnostics contributed by a probe.



```
void f(boolean b) {
    if (b) {
        g();
    }
}
```

Fig. 10. Arrow diagnostics representing a control-flow graph contributed by a probe. Each arrow is rendered with a pseudorandomly generated color. This is to make it easier to see where one arrow ends and another begins.



```
void f(boolean reset) {
  if (reset) {
    counter = 1;
  } else {
    counter++;
  }
}
```

MethodDecl.exit [8:1→14:1]

```
[14:1→14:1] Exit
.pred [14]
[10:5→10:15] AssignSimpleExpr
[12:5→12:13] PostIncExpr
```

Fig. 11. Probe on a synthetic Exit node of a control-flow graph, listing the predecessors of the node.

Fig. 10 shows another example where the property is a set of call graph edges. Here, each edge object contains information about where to draw an arrow in the text editor.

3.3. Liveness

CodeProber probes are live in that their results are updated after each edit to the source code, as was shown in Fig. 3. Probe results are updated also if the underlying compiler or static analysis tool is rebuilt. These updates happen automatically, as CodeProber listens to changes in the file system, and can detect when the underlying tool has been replaced. This supports a tight development loop, for example, when the developer fixes bugs in the underlying tool. In principle, a modified underlying tool might imply that existing probes can no longer be matched. For example, if the abstract grammar is changed, the new AST structure might differ from the previous one, even if the source code is unchanged. CodeProber makes a best effort to match probes anyway, as will be discussed in Sections 4 and 5.

3.4. Synthetic nodes

As discussed in Section 2, the value of a property might be a reference to a synthetic AST node, i.e., a node that is computed rather than constructed by the parser. Through the use of probes, these nodes can be investigated even if they do not have any source code representation. Fig. 11 shows an example from a control-flow graph analysis. The analysis constructs synthetic `Entry` and `Exit` nodes for each method so that the control-flow graphs have well-defined starting and ending points. In the figure, the user has created a probe for the synthetic `Exit` node, accessed via the `exit` property of the method. The user has also created a nested probe for the `Exit` node's `pred` property, to investigate its predecessors.

If a synthetic node is given artificial line and column information, this can be used for mapping the node to a source code position. In the figure, the analysis has set the line and column information for the `Exit` node so that it appears to be located at closing brace for the method. This allows the user to hover over the reference to the synthetic node, and see which method it belongs to.

4. Node locators

As mentioned earlier, so-called *node locators* are used at the client side to identify the AST nodes referenced by probes. These node locators need to be updated after changes of the source code in the text editor. There are many potential ways to identify where an AST node is located. Some examples in plain English are:

1. "The call expression on line 12, column 9"
2. "The third child of the fifth child of the root AST node"
3. "The class declaration with ID set to 'Foo'"

These ways of identifying nodes might work, but they are fragile to changes: The first example will break if, for example, a statement is added at the beginning of the source code; The second example will break if, for example, the construct of interest is nested inside a new statement; The third example will break if, for example, the class is renamed to 'Bar'.

Furthermore, there can be probes on synthetic nodes that have no textual representation, and that require more sophisticated identification methods.

We have designed the node locators with the goal of making them both resilient to different kinds of changes of the source code, and efficient to apply, i.e., to resolve them to actual object references. Another design goal is that node locators should be as language agnostic as possible, and not make assumptions of how source text is parsed. This prevents potential solutions that rely on inserting tracking markers in the code, for example using annotations or block comments, as annotations and block comments are not supported in all languages. In addition, such tracking markers would not be possible to use with synthetic nodes.

Our current design is the result of an iterative development process where we have tried out probes on many different properties for several different analysis tools and for different languages. In our experience the design works very well in practice, although there will always be corner cases when node locators can fail, for example when the corresponding code is completely removed. This is discussed in more detail in Section 4.5.

4.1. Node locator steps

A node locator is a list of *steps* where each step moves a current position to a new position in the AST. The steps are applied in order, starting at the root of the AST. Any of the steps can fail, in which case the whole application of the node locator fails, and no node could be identified. The following steps are supported:

Child has the form

$\texttt{Child}(i)$

It means go to the i-th child node.

TAL stands for *Type At Location* and has the form

$\texttt{TAL}(t, d, l_s : c_s \rightarrow l_e : c_e)$

Here, t is an AST node type, d is a number of steps down in the AST from the current node, and $l_s : c_s \rightarrow l_e : c_e$ is a line/column start/end span in the text. This step moves the current position to the "best" node of type t in the subtree of the current position and whose text span has at least one character overlap with

$l_s : c_s \to l_e : c_e$.[2] If there are multiple compatible nodes in the AST, then they are sorted by how closely they match d and $l_s : c_s \to l_e : c_e$ (see Section 5.2 for implementation details). If multiple nodes are equally good matches, then the first one in a depth-first traversal is chosen.

FN stands for *Function* and has the form

$$\mathrm{FN}(f, a_1, \dots, a_n)$$

where f is the name of a function on the current node, and $a_1, \dots, a_n$ are arguments to the function ($n \geq 0$). The function is expected to return an AST node reference, and the current position is moved to that node.

The Child steps provide a simple way of locating a node in an AST, but it is not very resilient to changes. Even a small change, like changing something in the beginning of the source code, would result in old node locators failing or resolving to the wrong node in the new AST.

The TAL steps were introduced to provide a resilient solution. They can handle variations in the placement of the nodes due to additions, deletions, and nesting changes. The TAL steps also make use of text spans in the edited source code. For this to work, the editor needs to adapt the TAL text spans in its stored probes as the text is edited. If, for example, the user inserts a newline between lines N and M, then for all TAL steps on line $L \geq M$, the line count is increased by 1. Similar adjustments are made for lines and columns on all insertions and removals.

The FN steps were introduced to handle synthetic nodes, constructed by the analysis tool in a different stage than parsing. In particular, they can be used for higher-order attributes (HOAs), in which case the function is simply the name of the HOA, and returns the root of the HOA subtree. In Reference Attribute Grammars, the HOAs are evaluated on demand and memoized, so in case the attribute had not already been accessed for other reasons, calling the function will result in the HOA being created before its root is returned.

The FN steps might be useful also for other purposes. They are very versatile as the function may return any node in the AST. One possible use might be to introduce application-specific steps, such as jumping to a particular source file or declaration node. However, we have not explored this possibility, since our main goal has been to provide an algorithm that works out of the box for any language and analysis tool.

4.2. Example node locators

In the Java compiler ExtendJ, the root AST node is of type `Program`. `Program` has a `List`, which in turn contains a number of `CompilationUnit` nodes, each corresponding to a single source file. Most AST nodes for a source file can be identified by first identifying a `CompilationUnit`, and then using a TAL step within that file.

As an example, assume we have the following variable declaration at line 5 in a given source file:

```
int a = 1;
```

An example node locator for the variable declarator (“`a = 1`”) is:

```
[ Child(0),
  Child(131),
  TAL("VariableDeclarator", 10, 5:13 → 5:17) ]
```

Here, `Child(0)` goes from the root AST node (`Program`) to the `List` node. `Child(131)` goes to the 132:nd `CompilationUnit`, which happens to represent the source file. “`10`” is the number of steps from the `CompilationUnit` down to the `VariableDeclarator` node. “`5:13`” is the starting line and column and “`5:17`” is the ending line and column.

For library compilation units, we rely on FN instead, since libraries are implemented using higher-order attributes in ExtendJ.

For example, to identify the `Integer` class, we would use:

```
[ FN("getLibCompilationUnit",
     "java.lang.Integer"),
  TAL("ClassDecl", 2, 0:0 → 0:0) ]
```

Here, the FN step represents the function call

`getLibCompilationUnit("java.lang. Integer")`

on the root node, and results in a `CompilationUnit` node. The TAL step then locates the `ClassDecl` two steps down from the `CompilationUnit`. The text span in this case is $0:0 \to 0:0$, which is expected for AST nodes that do not get created from a normal source file.

4.3. Kinds of node locators

There are two kinds of node locators that can be created: *naive* and *robust*.

Naive A list of steps corresponding to the path from the root down to the target node. For each edge on the path, the corresponding step is either a Child (for a normal child), or an FN (for a higher-order attribute).

Robust A naive locator where sequences of one or more Child steps are replaced with a single TAL step if possible, i.e., if applying it results in exactly the same node as the Child steps would.

The key difference between naive and robust locators is how good they are at locating their target node after the user has made changes to the source document. Naive locators are not good at handling changes, which is why we call them “naive”. Still, naive locators have their uses. Construction and application of naive locators is quite fast, so whenever a locator does not have to be resilient to changes, a naive locator is preferable. Both the input and output of a probe may be a collection of AST node references, which are represented by node locators. When a change happens, the input is used to re-calculate the output. Therefore, only the locators in the input need to be resilient to changes. The output nodes can be naive, which is good for performance.

Main scenarios when Child steps cannot be converted to TAL are when interacting with built-in or rewritten parts of the AST. Built-in AST nodes are often missing position information, i.e., their span is `0:0 → 0:0`, so any two nodes at the same depth and type in the AST will overlap. Rewrites can also create nodes with overlapping position, for example due to multiple rewritten nodes taking the position from a single source node. For example of why overlaps are an issue, consider the expression `f(x, y)`. If both `x` and `y` have the same span, then a TAL cannot be used to reference the `y` node.

There is a third scenario where TAL cannot be used, and that is with ASTs that have content from multiple source files. Since TAL steps only consider line/column and not “file path” or similar, nodes from different files cannot be reliably differentiated. See Section 5.5 for how we solve this problem.

[2] Nodes that are not given explicit spans by a parser should have the span $0:0 \to 0:0$ and are considered to overlap any other span.

4.4. Adapting to changes

As mentioned, the client keeps track of the active probes with their node locators, and adjusts the text spans of the TAL steps as the code is edited. However, the client does not have any knowledge of the syntax, so it only adjusts according to textual changes. The consequence is that the node locator might not fit exactly with the AST of the reparsed code. The flexibility of the TAL step usually makes it possible to find the node, but this also means that there can be a better, more exact node locator that should be used in the future. For this reason, when the server sends updated probe results to the client, it also sends the updated node locator for the probe.

As an example, assume we have the following source code:

```
if (x) {
  a();
}
if (x) {
  b();
}
```

The node locator for the first if-statement contains a TAL step `TAL("IfStatement", . . . , 1:1 → 3:1)`. Suppose now that the user decides to clean up some duplicated code, so they remove the two lines in the middle. The source code now looks like the following:

```
if (x) {
  a();
  b();
}
```

Because of the removed lines, the client adjusts the TAL step to `TAL("IfStatement", . . . , 1:1 → 2:9)`, covering only the first two lines (up to and including the `a()` statement). The client then informs the server that there are changes, and the server then sends back the updated probe result, together with a new more appropriate node locator with a `TAL("IfStatement", . . . , 1:1 → 4:1)`.

4.5. Known limitation: False positives

The proposed design for node locators has a known limitation relating to false positives. In particular, when the code has been edited, it is not always clear which node a user would perceive as the best match. For example, assume we have the following expression:

```
a(b(c))
```

The user adds a probe for `b(c)`. The locator for that probe looks like this:

```
[ TAL("CallExpr", 7, 5:13 → 5:16) ]
```

Then the user changes the source code to:

```
a(c)
```

After the change, the probe will be re-evaluated on `a(c)`, since it is the only AST node that matches `CallExpr` and overlaps with the original TAL position. If the user wanted the probe to keep matching the first argument to `a`, then matching `a(c)` is a false positive. The user would rather match `c`.

One potential solution to this scenario is to make use of subtyping: The call `b(c)` has the AST node type `CallExpr`, and according to the abstract grammar, its parent expects any node of the supertype `Expr` at that position. By using `Expr` instead of `CallExpr` in the TAL step, the new expression `c` would match, solving the user's problem.

Another potential solution is to make the users intent more explicit in the locator, and include a FN step that selects the first argument to `a`. Such a locator could look like this:

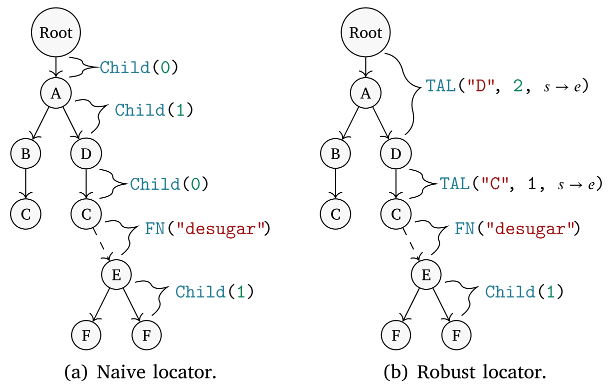


Fig. 12. Naive and robust node locator steps for the bottom-right "F" node.

```
[ TAL("CallExpr", 5, 5:11 → 5:17),
  FN("getArgument", 0) ]
```

Node locators need to balance resilience and risk of false positives when deciding how strictly they should match nodes. We found that being strict with types and permissive with locations seems to work well. Being less strict with types (for example by using subtyping) could potentially introduce more false positives, even if it would help the specific example above. Shorter locators also seem to be more resilient, so extra steps like `FN("getArgument", 0)` should be avoided.

A more robust solution to the problem could be to use multiple kinds of locators simultaneously, and use some heuristic to pick the best result among them.

5. Node locator algorithms

Creating and applying node locators involves a significant amount of traversal in the AST. Doing so efficiently can be a challenge. In this section we present algorithms for node locator construction and application. We also discuss which algorithms have potential performance problems, and how to mitigate them.

5.1. Creating locators

We will present two algorithms for creating node locators; CreateNaiveLocator and CreateRobustLocator.

Algorithm 1: Naive node locator creation

```
1: procedure CreateNaiveLocator(target)
2:     ▷ Create a naive locator for the target AST node.
3:     list ← []
4:     node ← target
5:     while parent(node) ≠ null
6:         list ← [CreateParentStep(node)] + list
7:         node ← parent(node)
8:     return list
9: end procedure
```

CreateNaiveLocator is presented in Algorithm 1. It assumes the existence of a procedure CreateParentStep(n) that returns a Child or FN step which represents how to get from the parent of n to n. A Child step can be used when the child is accessed by index, i.e., for most AST nodes. FN steps can be used for other AST nodes, for example children that are roots of HOA subtrees. CreateNaiveLocator creates a locator consisting only of Child or FN steps by iteratively calling CreateParentStep until it reaches the root of the AST.

CreateRobustLocator is presented in Algorithm 2. It steps through the result of CreateNaiveLocator and tries to identify subsequences of Child steps that can be substituted with a TAL step. To explain

Algorithm 2: Robust node locator creation

```
procedure CreateRobustLocator(target)
    ▷ Creates a robust locator for the target AST node.
    locator ← CreateNaiveLocator(target)
    src ← target
    dst ← target
    res ← []
    for each step in reverse(locator)
        if src ≠ dst and                                  ▷ Ongoing TAL?
          (step is FN or ¬CanExpandTAL(src, dst))
            res ← [CreateTAL(src, dst)] + res
            dst ← src
        if step is FN or ¬CanExpandTAL(src, dst)
            res ← [step] + res
            dst ← parent(src)
        src ← parent(src)
    end for
    if src ≠ dst                                          ▷ Ongoing TAL?
        res ← [CreateTAL(src, dst)] + res
    return res
end procedure

procedure CreateTAL(src, dst)
    ▷ Create a TAL step that can be applied to src to get to dst. src is
      an ancestor of dst in the AST.
    t ← type(dst)
    d ← distance(src, dst)
    s ← span(dst)
    return TAL(t, d, s)
end procedure

procedure CanExpandTAL(src, dst)
    ▷ Check if a TAL step from the parent of src would identify dst.
    expandedSrc ← parent(src)
    tal ← CreateTAL(expandedSrc, dst)
    return ApplyLocator([tal], expandedSrc) == dst
end procedure
```

the algorithm, we will show how a naive locator in Fig. 12(a) is transformed to a robust locator visible in Fig. 12(b). The node labels A, B, …, F indicate node types. CreateRobustLocator traverses the naive locator in reverse order and computes a shorter locator, where some Child subsequences are replaced by a TAL step. To keep track of the current subsequence, it uses two pointers, *src* and *dst* that represent the source and destination of a potential TAL step. When *src* and *dst* point to the same node, it means that no TAL is in progress. Initially, they both point to the target node of the locator, i.e., F in Fig. 12(a). In each loop iteration, CreateRobustLocator does the following in order:

1. If a TAL step is currently being built (*src* ≠ *dst*) and cannot be grown any further, finish building it and move *dst* up to *src* (lines 8 → 11).
2. If we cannot start building a new TAL step, take the incoming (naive) step as-is. Also move *dst* up one step past *src* to avoid starting a new TAL step (lines 12 → 14).
3. Move *src* one step closer to the root of the AST (line 15).

Finally, a TAL step is added from the root of the AST if needed (the final "*src* ≠ *dst*", lines 17 → 18).

The AST in Fig. 12(a) contains multiple steps where a TAL might not be possible to use, depending on whether nodes have unique text spans or not. In the worst case, where all nodes have identical text spans $s \rightarrow e$, there are three such cases:

Algorithm 3: Node locator application

```
procedure ApplyLocator(locator, astRoot)
    n ← astRoot
    for each step in locator do
        n ← ApplyStep(step, n)
        if n == null
            fail
    end for
    return n
end procedure

procedure ApplyStep(step, node)
    case step of
        Child(i)
            return getIthChild(node, i)
        FN(fn, $a_0$, .., $a_n$)
            return invoke(node, fn, $a_0$, .., $a_n$)
        TAL(..)
            return ApplyTALStep(step, node, node)
end procedure

procedure ApplyTALStep(step, node, src)
    if span(node) ≠ 0 : 0 → 0 : 0
      and disjoint(span(node), span(step))
        return null
    if type(node) == type(step)
        best ← node
    else
        best ← null
    for each child in children(node)
        match ← ApplyTALStep(step, child, src)
        if IsBetterMatch(step, src, best, match)
            best ← match
    end for
    return best
end procedure

procedure IsBetterMatch(step, src, lhs, rhs)
    ▷ Check whether rhs is a better match than lhs when applying step
      from src.
    if lhs == null or rhs == null
        return lhs == null
    sl ← abs(span(step) − span(lhs))
    sr ← abs(span(step) − span(rhs))
    if (sl == 0) ≠ (sr == 0)                        ▷ One perfect span match?
        return sr == 0
    dl ← abs(depth(step) − distance(src, lhs))
    dr ← abs(depth(step) − distance(src, rhs))
    if dl ≠ dr                                      ▷ One side closer to ideal depth?
        return dr < dl
    return sr < sl
end procedure
```

1. The bottom-right F node cannot be identified by a TAL since its left sibling would take precedence.
2. FN steps, such as between C and E, always prevent TAL steps.
3. The bottom right C can be identified by a TAL from its parent (D), but not from its grandparent (A). Such a TAL step would instead match the bottom-left C node.

The output of CreateRobustLocator can be seen in Fig. 12(b), where three Child steps are replaced with two TAL steps. If all AST nodes had unique spans, then the two TAL steps could be replaced by a single `TAL("C", 3,` $s \rightarrow e$`)` step. Figs. 12(a) and 12(b) are intentionally constructed to showcase how multiple levels of overlap are handled. In most normal cases a single TAL step can cover a majority of the tree.

5.2. Applying locators

Algorithm 3 shows the algorithm for applying a node locator. The key part is the procedure ApplyTALStep. It finds the best match for a TAL step in the subtree starting with *src*. When multiple potential matches are found, IsBetterMatch is used to determine which node is the best match. In case there are two equally good matches, then whichever one is found first is returned.

The *disjoint* check in ApplyTALStep exists to avoid traversing through subtrees that do not overlap with the TAL step. We assume that parent nodes fully cover all their children.

When comparing types, we currently consider only exact type matches. There are arguments for and against permitting subtypes here, see Section 4.5 for a discussion on this.

IsBetterMatch compares potential matches against an ideal match, which is determined by the span and depth values on the TAL step. Three criterias are considered in descending order of importance:

1. Perfectly matching span.
2. Least deviation from ideal depth.
3. Least deviation from ideal span.

If ApplyTALStep is changed to permit subtyping matches, then IsBetterMatch could also use type comparisons to select the best match. For example, a perfect type match might be less important than perfectly matching the intended span, but more important than the other criteria.

5.3. Optimizing node locator construction

There are cases when the performance of CreateRobustLocator (Algorithm 2) can be a problem. To understand why, we must analyze the worst case time complexity of some of the procedures that are involved. Assume a node locator is being created for a node that is d steps down in the AST from the root, and the full AST contains N nodes.

5.3.1. ApplyTALStep

CreateRobustLocator makes use of the ApplyLocator algorithm which in turn uses the procedure ApplyTALStep. This procedure iterates over every node that overlaps with the span of the TAL step. The time this takes depends on the shape of the AST, and the span information available on its nodes. If the AST is a balanced binary tree and each node has non-overlapping text spans, then this runs in $O(log(N))$ time. In the worst case, however, the AST is shaped like an upside-down T ($\perp$) and each node has overlapping text spans. In this case, almost the entire tree will be visited when applying TAL from any point along the vertical part of the $\perp$. This means that in the worst case, ApplyTALStep runs in $O(N)$ time.

5.3.2. CanExpandTAL

The CanExpandTAL procedure invokes ApplyLocator with a TAL step, so it also runs in $O(N)$ time in the worst case.

5.3.3. CreateRobustLocator

The main loop of CreateRobustLocator runs for d iterations. The first iteration can invoke CanExpandTAL once, and all subsequent iterations can invoke it up to 2 times. In the worst case, this means CanExpandTAL is invoked $(2 * d) - 1$ times. The total worst-case time complexity for CreateRobustLocator is therefore $O(d * N)$. Because $d = N$ in the worst case, the time complexity is quadratic with respect to the size of the AST.

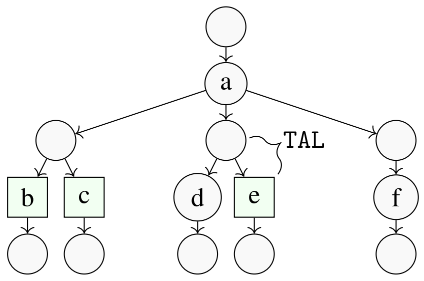


Fig. 13. Visualization of which nodes might be matched by a TAL step if it expands one step. Green squares are the potential matches. All other nodes are impossible to match, which enables the optimizations described in Section 5.3.4.

5.3.4. Achieving linear time

In practice, the performance of CreateRobustLocator is often good enough if implemented as Algorithm 2. The ApplyTALStep procedure skips over entire subtrees if their spans do not overlap with the TAL. In addition, ApplyTALStep does not search through "FN" connections in the tree, which further reduces the number of nodes that have to be visited. Still, performance can noticeably degrade when using locators in a larger context. In Section 8 we show benchmarks where probes update within tens of milliseconds for a project with a hundred thousand lines of code. Of those milliseconds, only a small fraction ($\sim$ 20%) comes from node locator related functionality. To get these numbers we had to optimize CreateRobustLocator. There are two relatively simple and very effective optimizations that can be done.

The first and most important optimization makes sure that nodes are not visited more than once when CreateRobustLocator runs. The process of creating TAL steps involves iteratively applying TAL steps further and further up in the AST, as long as the application results in the expected target node. This can result in ApplyTALStep traversing through the same subtree multiple times. However, only the first time is necessary, and in all subsequent visits to the same subtree the outcome of visiting that tree is already known.

The second optimization uses the knowledge that there is a perfect TAL match at *depth*(tal) steps down in the AST. Any node at a different depth cannot possibly be the best match, so ApplyTALStep can also stop upon reaching a depth further down than the expected perfect depth. Also, a better match can only be before the target node in a depth-first search (DFS), due to the DFS traversal in ApplyTALStep.

The optimizations allow CreateRobustLocator to avoid visiting large parts of the AST when creating or extending a TAL step. Fig. 13 aims to illustrate these optimizations. A TAL step is being created for the target node `e` and is being extended one step upwards in the tree, to include node `a`. First, only nodes on the same depth need to be considered. Then, node `d` can be excluded since it has already been visited (when creating the current TAL step). Node `f` can also be excluded since it is after `e` in a depth-first search. Thus, only the nodes `b`, `c` and `e` need to be considered for a perfect match (`e` is a perfect match since it is the TAL target).

The two optimizations brings the time complexity for CreateRobustLocator down to $O(N)$, even in the worst case of a '$\perp$'-shaped AST with fully overlapping text spans.

5.4. TAL adjustments

At the client side, CodeProber keeps track of the node locators of all probes, and adjusts spans of TAL steps when the users make changes in the editor. Each span is represented by two positions; start and end. The two positions are adjusted independently.

CodeProber uses the Monaco text editor, which reports changes as a range of text being replaced with some new text. The cases of typing or deleting characters correspond to the replaced range or inserted text being empty. These cases are relatively simple to handle. The more special case of text being removed and inserted at the same time is more challenging.

For example, assume that CodeProber has a node locator containing a position for the b inside the expression $a + b$. The user has $c * d$ in their clipboard. They mark b and paste, and the resulting text is $a + c * d$. It is not immediately clear where CodeProber should move the position that previously pointed at b. The edit could be treated as two independent edits in a sequence; first a removal of b, and then insertion of $c * d$. In this case, the removal of b should move the position to the space just after the + sign. The insertion of $c * d$ should then move the position up to the end of the inserted text, i.e the d. Another possibility, which CodeProber uses, is to try to retain the original position whenever simultaneous removals and insertions occur. In this case, it means that the position should end up at c.

Algorithm 4 shows HandleEditorChange which computes a new position based on an original position and a change event. It assumes that there are two functions HandleRemoval and HandleInsertion that adjust a single position based on just a removal or insertion. Based on their outputs it can detect simultaneous insertion and removal and decide whether the updated position should be used, or if the original position should be retained instead.

Algorithm 4: Handle simultaneous removal and insertion

```
1: procedure HandleEditorChange(pos, change)
2:     ▷ Adjust a position (pos) based on an change event (change).
3:     del ← HandleRemoval(pos, change)
4:     ins ← HandleInsertion(del, change)
5:     if pos ∈ removedSpan(change)                  ▷ pos inside removal?
6:       and ins ≠ del                               ▷ Was there an insertion?
7:       and ins > pos                        ▷ Insertion larger than removal?
8:         return pos                         ▷ If yes, retain original position
9:     return ins
10: end procedure
```

5.5. Handling multiple files

Most language tools work with multiple files, for example through imports or passing a list of source files on the command line. However, CodeProber assumes that all relevant information is available in a single AST. This means that the AST can be quite large, since it will typically include content from many files, and not only the edited text within CodeProber. It also means that nodes created from different files may have overlapping line/column spans. If no special measures are taken, this will lead to the following problems:

1. Accuracy of TAL steps will degrade due to the overlapping nodes.
2. Performance of creating and applying locators scales with the size of the AST, so a larger tree will be slower.
3. The list of nodes that is presented when creating a probe (see step 2 of Fig. 4) might include irrelevant nodes from external files, due to span overlaps.
4. Adjustments to TAL steps in external files cannot be reliably done, as CodeProber only has access to change information for the single edited file.

We have chosen to solve the above problems by introducing an optional boolean property `externalFileRoot` for AST nodes. This can be set by the analysis tool when building the AST. When this property is defined on a node, it means that the node represents a file. If set to false, then the node is the root of the edited file inside CodeProber. If set to true, then the file is any external file, for example an imported file. Several algorithms are then enhanced to take the `externalFileRoot` property into account, as will be described in the following subsections.

5.5.1. CreateRobustLocator

The issue of TAL having low accuracy across multiple files can be solved by not using TAL steps above files in the AST. Any node with `externalFileRoot` defined is assumed to represent a file. Therefore, CreateRobustLocator in Algorithm 2 can be enhanced to forbid TAL steps after a node with `externalFileRoot` is found. In more concrete terms, add a local boolean variable *foundFileRoot* to CreateRobustLocator, and update it in the main loop with:

```
1: foundFileRoot ← foundFileRoot
2:     or src implements externalFileRoot
```

Then, add "foundFileRoot **or**" as a prefix to the two instances of "step **is** FN". This stops any ongoing TAL step from being expanded further, and prevents any TAL from being built further up in the AST.

By not using TAL above files in an AST, the performance problems related to multiple files are solved as well. The performance of creating and applying TAL is strongly related to the size of the (sub-)tree where the creation/application starts. By forbidding TAL above files, the size of the (sub-)trees are limited to a single file.

Algorithm 5: Listing nodes that overlap with a position

```
1: procedure ListNodes(node, pos)
2:     ▷ Get a list of all AST nodes from node and down that overlap with
   pos.
3:     if node implements externalFileRoot
4:       and invoke(node, externalFileRoot) == true
5:         return []
6:     if span(node) ≠ (0 : 0 → 0 : 0) and pos ∉ span(node)
7:         return []
8:     res ← [node]
9:     for each child in children(node)
10:        res ← res +ListNodes(child, pos)
11:    end for
12:    return res
13: end procedure
```

5.5.2. Probe creation node list

As was shown in Fig. 4, when the user starts creating a probe, a list of nodes overlapping the clicked position is shown in a menu. Algorithm 5, ListNodes, shows how this list is computed. It is relatively similar to ApplyTALStep in Algorithm 3, in that it recursively traverses through the AST and has an early return condition if the position looked for is outside the span of the currently examined node. To avoid descending into nodes for an irrelevant file, an additional return condition is added, checking if the examined node has the `externalFileRoot` property set, and with the value true. In this case, an empty list is returned.

5.5.3. CreateTAL

To make TAL step adjustments more reliable for external files, we extend TAL with a new boolean field, *external*. This field should be set to true whenever the target node of a TAL, or any of its ancestors, has `externalFileRoot` set to true. In more concrete terms, replace the last line of CreateTAL with:

```
1: e ← IsExternal(dst)
2: return TAL(t, d, s, e)
```

IsExternal recursively searches upwards in the AST for the closest implementation of *externalFileRoot*:

```
1: procedure IsExternal(node)
2:     if node == null
3:         return false
4:     if node implements externalFileRoot
5:         return invoke(node, externalFileRoot)
6:     return IsExternal(parent(node))
7: end procedure
```

CodeProber's client does not adjust the values on a TAL step with *external* set to true.

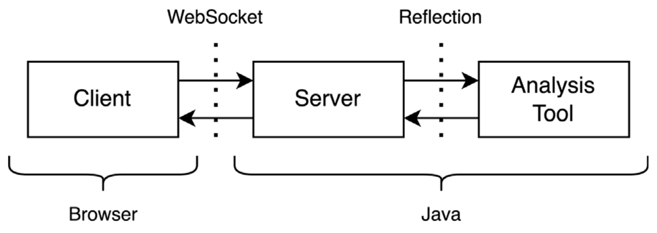


Fig. 14. High level architecture.

6. Implementation

The overall architecture of CodeProber has three components: a client, a server, and an analysis tool, see Fig. 14. The implementation is open source and available on https://github.com/lu-cs-sde/codeprober. This section describes the three components and the API between them, together with rules and recommendations on how the AST produced by the analysis tool should work.

6.1. Overall architecture

In CodeProber, the client is a web page, mostly written in TypeScript, and using the Monaco code editor[3]. The server is written in Java, and can use any analysis tool written using the JastAdd metacompiler (packaged as a jar file, following certain conventions).

For practical purposes, CodeProber packages the client and the server together as a single jar file which takes a path to the analysis tool as its argument. When started, CodeProber opens a local HTTP server that serves the webpage, and a local WebSocket (Melnikov and Fette, 2011) server for all dynamic requests. The user can then simply go to the webpage and start editing and creating probes.

6.2. Client↔server API

CodeProber's client is a browser application, so the options for inter-process communication with the server is quite limited. CodeProber supports HTTP requests and WebSocket, but defaults to using WebSocket for its better performance. There are cases where HTTP requests are preferable, see Section 7.6. The client and server communicate with remote procedure calls (RPC). There are three different request types sent by the client: ListNodes, ListProperties and EvaluateProperty, corresponding to the three steps when the user creates a probe, as in Fig. 4. The client sends the EvaluateProperty request also when the user edits the text, as in Fig. 3.

There is one message sent from the server to the client, Refresh. This is sent when the server detects that the underlying analysis tool has been updated. The client is not expected to respond to this message, but will instead re-evaluate all active probes by sending EvaluateProperty requests.

In all requests sent from the client to the server, the client includes the current editor state, i.e., the full text. The server will then ask the underlying analysis tool to parse this text into a new AST. Sending the full text in each request allows the server to be stateless. For performance, the server can, however, cache the latest text used for parsing, to avoid reparsing if the text has not changed. With this optimization, we have not seen that sending the full text in each request gives any performance problems, even if the text is large. Any imported files can be similarly cached. Additionally, it is possible to buffer changes until a period of inactivity has passed, see Section 7.6.

For responses to requests on node locators, the server includes updated node locators in the response, based on the new AST. The client then uses the new locator in subsequent requests. This way, the node locators on the client side are constantly adapted to the most recent AST, as was discussed in Section 4.

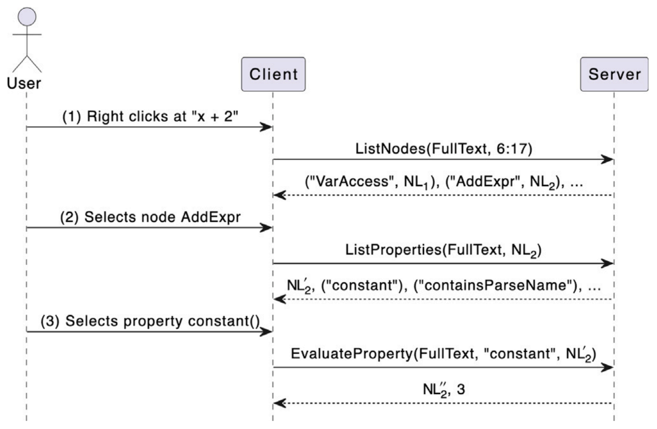

User
Client
Server
(1) Right clicks at "x + 2"
ListNodes(FullText, 6:17)
("VarAccess", NL₁), ("AddExpr", NL₂), ...
(2) Selects node AddExpr
ListProperties(FullText, NL₂)
NL′₂, ("constant"), ("containsParseName"), ...
(3) Selects property constant()
EvaluateProperty(FullText, "constant", NL′₂)
NL″₂, 3

Fig. 15. Sequence diagram for Fig. 4.

Example. Fig. 15 shows a sequence diagram of the messages sent when the user creates the probe in Fig. 4. In the first step ($1 \rightarrow 2$), the user clicks in the text to create a probe. The client then sends the ListNodes request, with the full text and the cursor position as arguments. In response, the server sends a list of node locators, corresponding to the nodes that match the cursor position.

In the second step ($2 \rightarrow 3$), the user selects one of the nodes. The client then sends the request ListProperties, with the full text and the node locator as arguments. In response, the server sends an updated node locator NL'_2, along with a list of property identifiers, each containing a property name and its argument types.

In the third step ($3 \rightarrow 4$), the user selects one of the properties. The client then sends the request EvaluateProperty, with the full text, the node locator NL'_2, the property identifier and any arguments to the property. (If the property has arguments, the client prompts the user to supply them interactively.) In response, the server sends the updated node locator NL''_2, as well as the probe result. Any AST nodes in the result are encoded as node locators that can be used for future ListProperties requests.

For the scenario in Fig. 3, the client will send one EvaluateProperty request for each of the active probes.

6.3. Server↔analysis tool API

The concept of property probes can in theory be used for analysis tools implemented in any language. CodeProber is written in Java and currently requires the analysis tool to run on the JVM. To use CodeProber with analysis tools not running on the JVM, a bridge implementation running on the JVM is needed. In our experiments we have used analysis tools running on the JVM. Most are implemented with the JastAdd metacompiler.

The server communicates with the analysis tool using reflective calls. Heavy use of reflection can have a negative effect on performance, but this is not a problem for CodeProber, as seen in Section 8.

The AST is parsed or reparsed by calling the `main` method on the analysis tool jar file, with the path of a temporary file containing the client state (the full text). The main method should store the parsed AST in a static field in the main class, that should be declared as follows:

```
public static Object CodeProber_root_node;
```

[3] https://microsoft.github.io/monaco-editor/. Accessed February 1, 2024.

Once an AST is produced, the server uses reflection to access and traverse it. The server assumes that the AST follows a certain structure. A number of methods should be available on each AST node for traversal purposes:

1. `getNumChild()` - returns the number of children on this node.
2. `getChild(int)` - returns a child at a given index.
3. `getParent()` - returns the parent node for this AST node, or *null* for the root AST node.
4. `getStart()` / `getEnd()` - returns the start/end position as line/column pairs for this AST node.

When listing available properties, the default implementation is to again use reflection, via *java.lang.Class.getMethods()*. The full list is filtered before being returned to the client. Methods that are non-public are removed. Methods with arguments can only contain arguments of type `int`, `boolean`, `String` or AST node references. Methods with other argument types are removed.

To compute node locators, it must be possible to determine the connection between a parent and child AST node in the form of either a Child or FN step.

Child steps are determined by iterating over all children in the parent node. Using identity comparison, we can detect the index of the child node.

If a node is a higher-order attribute (HOA), an FN step should be used, including the name and the arguments of the HOA. This can be determined thanks to the fact that JastAdd memoizes the arguments to, and results of, all HOA invocations. To construct an FN step for a HOA, CodeProber selects the parent of the HOA, and then iterates through all the parent's HOA memoizations, again using identity comparison to find the name and arguments of the appropriate child HOA.

In case no parent/child connection can be determined, the child node is considered to not be attached to the AST. This causes node locator creations to fail, and the server sends an error code to the client.

6.4. Desirable AST features

To use property probes, some design choices for the analysis tool are desirable to help improve the user experience: good source locations in the AST, on-demand evaluation of properties, and pure properties. We will discuss these in turn.

Source locations. For property probes to work well, it is desirable that the parser captures line and column positions and stores them in the AST nodes at parsing. Also, the positions should honor the AST hierarchy: CodeProber assumes that a node with an explicitly set position has an equal or larger span than all nodes in its subtree. Otherwise, our TAL algorithm might miss the best matching node, since it uses positions to prune subtrees in its search.

In many tools, nodes do get appropriate positions, in order for the tool to be able to report locations of errors and warnings. However, for a given tool, there might be nodes that lack this information. For instance, locations may have been added only for nodes with associated error messages. Another reason might be that the AST has been transformed, without carrying location information over to the transformed parts.

Missing locations will degrade the user experience, as features like highlighting and right clicking to select AST nodes will not work. In the client of CodeProber, a small warning triangle is shown next to each AST location that has its line and column set to zero.

However, CodeProber also tries to compensate for missing location information. It uses a *position recovery strategy* to infer suitable location information in case it is missing. There are multiple supported strategies, and the user can select which (if any) to use. The default strategy is called RecoverZigZag and is shown in Algorithm 6.

Algorithm 6: Default position recovery strategy

```
1:  procedure RecoverZigZag(node)
2:      ▷ Find a replacement position for node from its parent and children.
3:      up ← node
4:      down ← node
5:      while up $\neq$ null or down $\neq$ null
6:          if up $\neq$ null
7:              if span(up) $\neq$ $(0:0 \rightarrow 0:0)$
8:                  return span(up)
9:              up ← parent(up)
10:         if down $\neq$ null
11:             if span(down) $\neq$ $(0:0 \rightarrow 0:0)$
12:                 return span(down)
13:             down ← firstChild(down)
14:     return $(0:0 \rightarrow 0:0)$
15: end procedure
```

RecoverZigZag looks at nearby parent and child nodes, progressively searching further up and down in the AST until an explicitly set location is found, and which is then used as a replacement position. A recovered position usually covers a slightly larger or smaller span than the real span of the AST node. Therefore, position recovery should be seen as a temporary solution, and it is better if the analysis tool is updated so that all AST nodes carry their own position instead.

On-demand evaluation. The user can create probes for any property the AST supports, but usually only a tiny subset of the functionality is ever probed for at the same time. This fits well with on-demand evaluation. Rather than computing all properties up front, it is advantageous if properties are lazy and their values computed only when demanded. On-demand computation is not strictly necessary, but if all potentially probed values are computed up-front, as soon as the source text is edited and the AST is reparsed, re-evaluation of all these values might take a long time and negatively impact the user experience. Of course, if the user does not edit the source text, but only explores properties, the probes can still be very valuable for tools that do up-front computations.

Pure properties. The probed properties should be observationally pure, i.e., without visible side-effects when accessing them. If accessing properties has side-effects, they may behave differently depending on in which order they are invoked, and the benefits of property probes then diminishes. In addition, there is a caching setting in CodeProber that greatly improves performance by reusing the AST whenever possible, to avoid unnecessary reparsing. If properties can cause changes in the AST, then caching is not reliable.

All our evaluations have been performed with JastAdd-based tools, where all property evaluation is on-demand and all properties (attributes) are observationally pure.

6.5. Program representation

In this paper, we have assumed that the program representation is an AST. However, it is sufficient if the representation has a spanning tree, with the traversal interface discussed above, and where source text locations can be attached to the nodes in the spanning tree. In fact, this is the case for JastAdd tools: Many of the JastAdd attributes are node references, so the program representation is actually a graph, but with the AST as the spanning tree.

We have also performed brief experiments with non-JastAdd tools like SpotBugs[4], PMD[5] and WALA[6]. SpotBugs and WALA both perform their analyses on the bytecode level, which is common among analysis tools for Java. Bytecode level tools can also be used with CodeProber,

[4] https://spotbugs.github.io/. Accessed 1 February 2024.
[5] https://pmd.github.io/. Accessed February 1, 2024.
[6] https://github.com/wala/WALA. Accessed February 1, 2024.

but the experience is slightly inferior. This is because the text editor in CodeProber shows source code, but the AST that the user interacts with is that of the bytecode, which is not as intuitive. Still, we were able to explore the properties of those tools, which shows that CodeProber is not necessarily restricted to tools that put analyses directly on top of source code AST's.

7. Case studies

We have performed a number of case studies of CodeProber in order to qualitatively evaluate its usefulness, and find opportunities for improvements. In particular, we have run it on a full Java compiler, on a tool for intraflow analysis of Java, and on compilers for several other smaller languages. We have used the tool in teaching two different courses (compiler construction and program analysis). Furthermore, we have deployed it on a cloud server, so it can be run directly in the browser without the need for installing any local tools.

7.1. Java compiler

During development of CodeProber we continuously tested its functionality on the Java compiler ExtendJ. Based on this experience, we added a few features that we think are useful also for many other analysis tools.

For example, the *position recovery strategy* mentioned in Section 6.4 was added specifically because some node types in ExtendJ do not carry their own position information. The support for *multiple files*, described in Section 5.5, was also added based on our experience from ExtendJ.

During development we also identified and fixed two different caching issues in ExtendJ, which we contributed back to ExtendJ. We hypothesize that these issues had not been discovered before because ExtendJ had not before been used in the live, incremental way that CodeProber uses its underlying analysis tools.

In general, our experience is that using property probes with ExtendJ has been an excellent way of building understanding of the compiler. One common use case for ExtendJ is to write program analysers or experimental extensions to Java. To do this you need to know what AST node types are available, and what properties they contain. This can be accomplished by consulting the official API documentation[7]. However, we soon found ourselves using CodeProber more often than the documentation. For example, if you want to know what properties are available on a *for each* statement, then you can write such a statement, right click on it and see the list of properties. If any property looks interesting, you can click it to immediately see how it works. With the API documentation you need to first find the name of the node (`EnhancedForStmt`) and then you get a list of property names, but these cannot be directly invoked since there is no concrete code attached.

7.2. Flow analysis

IntraJ (Riouak et al., 2021) is an extension to ExtendJ that adds intraprocedural control-flow and dataflow analysis. The main developer of IntraJ used CodeProber in developing new types of flow analysis. Before using CodeProber, the IntraJ developer had used "print debugging" when developing new analysis features:

- Write code for the new feature.
- Add print statement(s) to check that the feature works correctly.
- Iteratively modify the code and print statements until you get the expected behavior.
- Remove the print statement(s).

Now, property probes have replaced most of the "print debugging" steps, since it is much faster and simpler to open/close probes than it is to add/remove print statements and recompile IntraJ.

The IntraJ developer also mentioned that they use the ExtendJ API documentation less, since it often is quicker to explore functionality via the property probes in CodeProber.

One new feature was added specifically for IntraJ, *arrow contributions*. This is a feature that allows property probes to contribute arrows to be drawn between two positions in the source code, overlaying the text. IntraJ uses this feature to visualize the control-flow graph directly in the source code. Previously, the control-flow graph was usually inspected in textual form, which was inconvenient, or using a `dot` visualization of the AST with control-flow edges, which became very large even for a small program. The visualization using arrows is shown in Fig. 10. The idea was inspired by the bug explanations in Clang Static Analyzer[8].

7.3. Compilers for other languages

We tried out CodeProber with compilers for a number of different additional languages: Oberon-0 (Fors and Hedin, 2015) (a very tiny procedural language), Bloqqi (Fors and Hedin, 2016) (a visual language for automation), and SimpliC (a simple C-like language, used in teaching). The experience worked well for all compilers. We did, however, discover that each of the compilers had a few AST nodes that did not carry correct location information. Nodes that produced errors/warnings generally had correct locations. Missing locations were usually attributed to either desugaring or that multiple nodes were created by the same parser production (the parser generator attached location only to the return node of a production). This was no major issue, however, since the *position recovery strategy* developed for ExtendJ worked fine here, as well.

We also used CodeProber with a student implementation of a ChocoPy compiler. ChocoPy is a subset of Python, commonly used for educational purposes (Padhye et al., 2019). Here we had more severe location-related issues. One of the challenges with parsing Python is the indentation sensitivity. The student ChocoPy implementation we used had solved this by making the parser a two-step process; first it transforms all indentation into whitespace-insensitive indentation tokens, and then it parses the transformed source code. AST nodes produced from the transformed sources always had line numbers that matched the original source code, but their columns were usually wrong by a few characters, since the whitespace-insensitive tokens did not match the width of the original indentation. It was possible to create probes, but the user experience was not very good. Of course, these problems could easily have been solved by fixing the student ChocoPy implementation to set proper line and column numbers, but it illustrates the kinds of problems a user can run into.

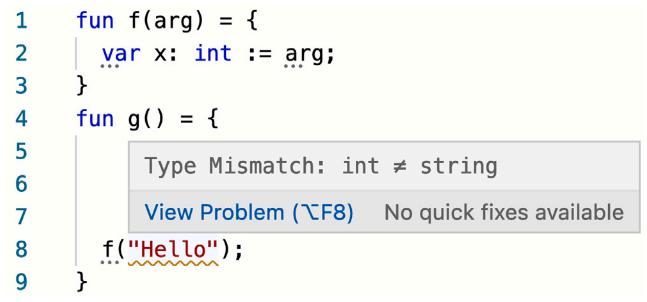
7.4. Compiler course

We used CodeProber in a course on compiler construction which is taken by around 70 students each year. In the course, the students build a compiler for a C-like language, step by step over six lab sessions. The first two labs cover scanner and parser generation. The remaining four labs cover name analysis, type analysis, call graphs, and code generation, all using the JastAdd metacompiler. For most students, this course is their first encounter with terms like "grammar", "abstract syntax tree", etc.

For many years, an AST exploration tool, DrAST (Lindholm et al., 2016), has been used successfully in the course, allowing the students to visualize the AST of a program, and to look at attribute values

[7] http://extendj.org/doc/. Accessed February 1, 2024

[8] https://clang-analyzer.llvm.org/. Accessed February 1, 2024.



```
fun f(arg) = {
  var x: int := arg;
}
fun g() = {
      Type Mismatch: int ≠ string
      View Problem (⌥F8)   No quick fixes available
  f("Hello");
}
```

Fig. 16. CodeProber being used in a program analysis course.

of individual nodes. For the 2022 edition of the course, we introduced CodeProber as an additional opportunity for the students to use, and gave a brief introduction to the tool early in the course. After each of the lab sessions, we performed short informal interviews with the students, to ask about their experiences with both DrAST and CodeProber.

The students' tool preferences fell into three main groups of roughly equal size. Either they used CodeProber, DrAST, or neither of the tools. Very few students used both tools.

The students who preferred using CodeProber generally liked the ease and speed of testing their compilers: Adding/removing code and probes could be done much quicker than, for example, writing test cases. While this was positive feedback, it also highlights one downside we noticed with CodeProber: it can lead to students writing fewer ordinary unit tests. Because it is so quick and convenient to open a probe, the students did not feel motivated to write the tests. They had the opinion that if something breaks in the future, it is easy to just start probing again. While this may work in the short term, having normal regression tests help in the long term. To improve on this, we plan on adding a *test probe* feature in CodeProber, i.e., a mechanism that allows a probe to be saved as a test case. We think such a feature can provide a very convenient way of constructing test cases.

A common point of feedback from those students that preferred DrAST was that they liked being able to see the AST visually. When working with tools like JastAdd, students need to have a good understanding of what the AST looks like, to know where and how to declare attribute equations. Even when the students had understood the concept of an AST, it was helpful to them to see the individual nodes visually for a given example. AST Probes (see e.g. Fig. 8) were added based on this feedback. The implementation is inspired by the DrAST view.

The students who did not use either tool said they did not "see the point" of using either tool. They used more traditional methods to aid the development, like test cases and "print debugging". When students asked for help during the labs, we sometimes asked them to inspect a few properties inside CodeProber. We needed to guide them step-by-step on how to do this. Afterwards, we noticed that some of those students started using CodeProber much more. This tells us that there is a barrier to getting started with CodeProber, and we need to work on improving usability for first-time users. We believe the portion of students using CodeProber will be larger in next iterations if we add some of the planned improvements.

7.5. Program analysis course

The program analysis course included labs on type inference, interval-based dataflow analysis to detect array out of bounds errors, and type analysis using points-to analysis. For these labs a small teaching language, TEAL, is used, and implemented using the JastAdd metacompiler. The core language implementation is provided as part of the course material, so the students only need to implement the analysis

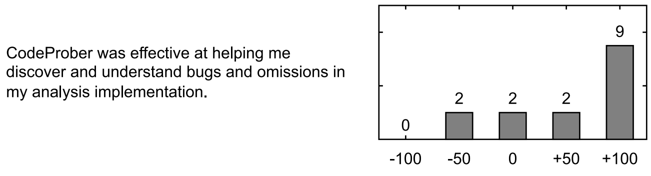


Fig. 17. Result from course evaluation from a program analysis course. +100 is "fully agree", and −100 is "fully disagree".

parts. The course leader forked CodeProber and modified it to be a more special-purpose tool for this course. The fork had a number of small changes, such as adding syntax highlighting for TEAL and hiding some of the options in the tool that were not necessary for the course. However, one of the main changes was the support for *background probes*, which are probes that are always on, and whose results are presented via squiggly lines at various points of interest in the code, and supporting extra hover information. A unique set of background probes was then configured for each lab. Fig. 16 shows an example of this from a lab on type inference. The figure contains several places with hoverable information that are extracted from the students' analysis tool by the background probes. Hovering over the tiny dots (...) will show type equations for the related program element. Yellow squiggly lines appear wherever the type equations conflict and produce a type mismatch. Hovering over the lines, as is done in the figure, shows details about the mismatch. As the students progressed through the lab, more dots and yellow squiggly lines appeared. Whenever type equations or type mismatches seemed wrong, students could resort to normal, manually created probes to investigate why. The preconfigured background probes made it easy for the students to get started, and is definitely something we will use in the future. A few analyses the students wrote needed to run in a loop until some value converged. Implementation mistakes could lead to infinite loops, which manifested in CodeProber as a probe loading forever. This is an area where CodeProber can be improved. For example, each probe can in theory be evaluated in a separate process, and that process can be killed after a certain timeout. The server could also periodically sample and report stack trace information back to the client, to give hints as to where the loops/deadlocks are occurring.

Every student used CodeProber to some degree. It was also through CodeProber that teachers reviewed the student implementations at the end of the labs. However, some students only used the preconfigured background probes, and did not explore any extra properties on their own. We suspect this group of students has some overlap with those who used neither DrAST nor CodeProber in the compiler course.

It was not originally planned for CodeProber to become a special purpose tool like this, but the general feedback from students was positive: In an anonymous course evaluation questionnaire completed by 15 students, 9 agreed strongly with the statement that CodeProber was effective in helping them discover and understand bugs and omissions in their analysis implementations (see Fig. 17). To support use cases like this, CodeProber will merge in some of the fork's changes, and add more configuration support so that it can be adapted to future use cases without needing to fork the entire repository.

7.6. Cloud server

The normal way of running CodeProber is to run the server on the local machine, and browse to localhost to run the client. We were also interested in hosting the CodeProber server on a cloud server so that users can run the client directly on the web, without installing any local software. This can be very useful for demonstrations and also for providing playgrounds for users before deciding to install it on their own computer.

As an experiment, we decided to adapt CodeProber to run in GitHub Codespaces. Codespaces is a service that hosts development environments in the cloud, and allows you to connect to them directly in the browser. GitHub Codespaces was made generally available in 2022.[9] Anybody with a GitHub account can try it for free, which makes it a good candidate for running demonstrations. A few minor changes were made to CodeProber to make it run well in Codespaces. The most notable ones were to delay requests and to make WebSocket optional.

7.6.1. Delayed requests

We noticed that during heavy use, the requests to Codespaces would sometimes fail. When inspecting our server logs we could not see any trace of the request that failed. Our guess is that Codespaces has some hidden throttling limits. To overcome this limitation we added a small delay to probe updates when running in Codespaces. Instead of immediately updating when the user makes a change, the client will wait for a period of inactivity before sending update requests to the server. The delay defaults to 0.5 s. After adding this delay we have not had any issue with random request failures.

7.6.2. Optional WebSocket

The WebSocket connection between CodeProber's client and server is normally able to stay connected indefinitely. When running in Codespaces however, it automatically disconnects after a few minutes of inactivity. The exact inactivity time varies, but seems to be around 1 to 5 min. During normal use, CodeProber might be idle in the background for several minutes while the user is working on their tool, and only occasionally will the user come back to inspect something in CodeProber. This usage pattern does not work well with the automatic WebSocket disconnections.

To overcome this issue we added support for sending WebSocket-related requests as HTTP PUT requests instead. We replaced server-initiated messages (e.g. Refresh in Section 6.2) with long polling. The result is a slightly slower time per request, since each request has to establish a new connection. The upside is that this allowed CodeProber to stay idle in the background for a long time without disconnection issues.

7.7. Summary

The case studies show that property probes are useful for a variety of analysis tools and different languages, both for development and in teaching. By running CodeProber for a full Java compiler, we made sure it works for real program analysis development. By running it in courses, we found that students appreciated the tool, and used it voluntarily to solve problems, which we take as an indication of its usefulness. Furthermore, by trying it out in many different scenarios, we have found several opportunities for adding new useful support. Examples include the position recovery strategy (fixing missing line and column information), support for multiple files (necessary for big projects), arrow diagnostics (visualizing graphs on top of the code), AST probes (visualizing the AST in a probe), and background probes (that are always active, and manifested directly in the code as squiggly lines). We also successfully adapted CodeProber to run on a cloud server, allowing users to try it out without needing to install any software.

[9] https://github.blog/2022-11-10-whats-new-with-codespaces-from-github-universe-2022/

8. Performance evaluation

In the previous section we saw examples of CodeProber being used for a few different tools and in teaching scenarios. These examples show that CodeProber works well in practice, at least in smaller contexts. In this section we investigate the performance of CodeProber for larger projects, to ensure that the response time is low enough for interactive use also in these cases. We are interested in finding out the administrative overhead of creating and evaluating probes, how it relates to other overhead like parsing, and how these overheads scale with project size.

8.1. Methodology

The processing time it takes to use property probes can be divided into three main parts:

- Parsing
- Property evaluation
- Probe administration

The processing times for parsing and property evaluation depend on the underlying analysis tool, which is ExtendJ in our benchmarks. As mentioned earlier, ExtendJ is a full Java compiler, implemented using the JastAdd metacompiler, which uses on-demand evaluation for properties. Different properties may naturally take different time to evaluate. Since we are interested in investigating the probe overhead rather than property evaluation, we have chosen simple properties with constant evaluation time in our setup.

To evaluate properties, the code needs to be parsed into an AST, and reparsing is needed whenever the user edits the code. Initially, all source files on the source path will be parsed, including the one edited in CodeProber. Additional imported class files are parsed on-demand, depending on what properties are evaluated. ExtendJ has support for incremental parsing at the file level, so when the user edits code, only the edited file is reparsed, and ASTs for other files on the source path are reused. As part of the parsing cost, we also count the administrative cost of flushing memoized properties—their values might be inconsistent since the AST has changed. (There is an incremental attribute evaluation mode in JastAdd, but it carries its own overhead, and is not used in our experiments.)

For property probes, all time-consuming work happens on the server side, so this is what we measure in the probe administration part. A headless client is used for measurements, and it runs on the same machine as the server in order to avoid any network latency in the data.

The probe administration part contains all the server-side functionality that is required to support property probes. This includes listing nodes that overlap with the user's cursor, creating and applying node locators, serializing probe results to send them to the client, etc.

We have run measurements on both a high-end benchmark machine and on a normal development laptop. The results we report are from the benchmark machine, in order to get as noise-free results as possible. The results from the laptop were roughly 1.5 times slower than those from the benchmark machine.

The benchmark machine ran Java 11 on Ubuntu 21.20 with an Intel i7-11700K CPU and 128 GB DDR4 RAM. The machine was configured with a minimal amount of background services to reduce noise in the data. The laptop ran Java 17 on Mac OS 12.2.1, an Apple M1 Pro CPU and 16 GB LPDDR5 RAM.

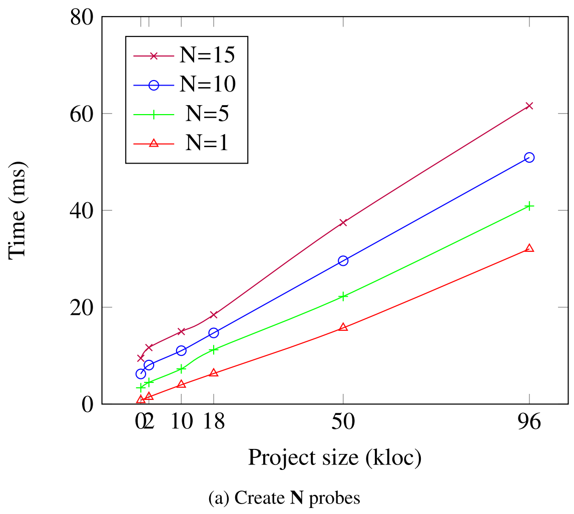
Benchmarking is done with three variables: project configuration **P**, action type **T**, and number of actions **N**.

P is one of six project configurations. The minimum configuration is a single file containing five lines of code. The largest configuration is the source code of Apache FOP[10], which contains over 900 source files and 96 K lines of code.

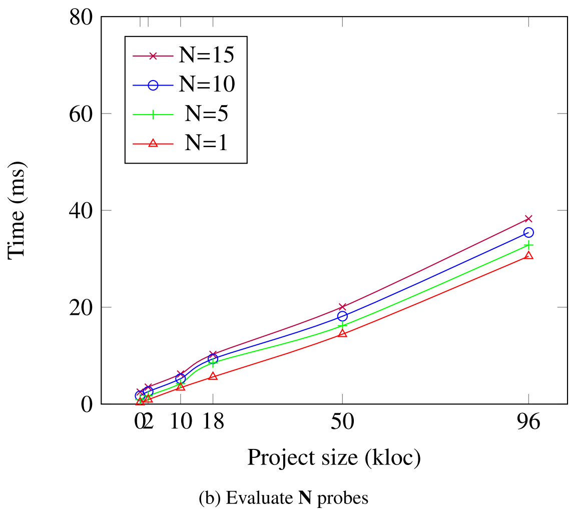
[10] https://xmlgraphics.apache.org/fop/. Accessed February 1, 2024.

Table 1
Performance measurements, all times in milliseconds.
The data for creating and evaluating probes is plotted in Fig. 18.

Project name	Size	Time to create N probes (ms)				Time to evaluate N probes (ms)				Full parse (ms)	
	LOC	N = 1	N = 5	N = 10	N = 15	N = 1	N = 5	N = 10	N = 15	Steady state	Startup
Mini	5	0.8	3.4	6.2	9.5	0.3	0.9	1.7	2.5	0.1	128.5
Probe Server	2 K	1.5	4.5	8.1	11.7	0.9	1.7	2.6	3.5	9.0	198.2
Commons Codec	10 K	4.0	7.3	11.0	15.0	3.4	4.2	5.2	6.2	44.1	290.6
NetBeans	18 K	6.3	11.2	14.7	18.4	5.6	8.5	9.3	10.3	57.7	345.7
PMD	50 K	15.7	22.2	29.6	37.5	14.4	16.2	18.1	20.1	155.1	493.2
FOP	96 K	32.0	40.9	50.9	61.6	30.5	32.8	35.4	38.3	348.4	695.1



(a) Create **N** probes



(b) Evaluate **N** probes

Fig. 18. Time to create and evaluate probes.

The action type **T** is either creating or evaluating probes. The number of actions **N** is 1, 5, 10 or 15.

Whenever CodeProber performs more than one action for the same source code, it will reuse the AST. Therefore, the parsing cost only needs to be paid once per source code version. The cost for a subsequent action$_k$ ($k > 1$) is therefore only the property evaluation time and probe administration.

For each combination of **P**, **T**, and **N**, we performed the following sequence:

1. Simulate a change to the source code.
2. Perform **N** actions of type **T**.

This sequence was performed in a loop until steady state had been achieved. After that we performed the sequence an additional 5000 times, and recorded the average time to finish the **N** actions.

We also measured full parsing time, since this is relevant both when the user starts up CodeProber, and in case the user reconfigures the analyzed project, in which case a full reparse of all source files is performed. For this reason, we measured full parsing both as a start-up cost (without JVM warmup) and as steady state (after warmup of the JVM).

8.2. *Results*

All results are shown in Table 1. Furthermore, the time for creating and evaluating probes are plotted in Figs. 18(a) and 18(b). We will now discuss these results in more detail.

8.2.1. *Creating a probe*

Creating a probe involves two operations:

1. List all AST nodes overlapping with the user's cursor.
2. List all properties available on a given AST node.

The two operations correspond to the ListNodes and ListProperties requests in Section 6.2. The result of these actions is a probe with a so far empty result. The actual evaluation of the probe is triggered by the EvaluateProperty request which is benchmarked separately.

If a user creates a probe based on an existing probe result, the first operation will be skipped. In our benchmarks we always measure both operations, which gives us a worst case time for creating a probe. The results can be seen in Table 1, and are plotted in Fig. 18(a).

8.2.2. *Evaluating a probe*

A probe is evaluated for one of two reasons:

- either a new probe was created,
- or some underlying data changed, and existing probes must be re-evaluated

Evaluating a probe corresponds to the EvaluateProperty request in Section 6.2. In the scenario where a user plays around with code to see how properties update, the probes will be evaluated significantly more often than they are created. In another scenario, where the user keeps the code fixed, and only explores new properties by creating new probes, there will be a single evaluation each time a probe is created.

Since the property evaluation cost is kept very small in our benchmarks, the measurements should consist almost entirely of parsing time and probe administration time.

The results can be seen in Table 1, and plotted in Fig. 18(b).

8.2.3. *Full parse time*

The results shown in Figs. 18(a) and 18(b) are using incremental parsing. The costs shown include a single re-parse of the edited file, regardless of how many probes were created or evaluated. A full parse is sometimes needed, such as when initially starting CodeProber, or when changing project configuration while CodeProber is running. We have benchmarked the full parse in two scenarios. Once where the parsing code had reached steady state, and once where the parsing code had just been loaded into the JVM ("Startup"). Full parse is measured for all project configurations **P**. The time for the full parse (steady state and start up) can be seen in Table 1.

8.3. Discussion

J. Nielsen defined three time limits concerning responsiveness (Nielsen, 1993). According to that definition, responding in less than 0.1 s is enough to appear instant, responding in less than 1 s is good enough to not interrupt the user's flow of thought, and responding in less than 10 s is the limit for keeping users' attention so they do not start on other tasks.

As can be seen from Table 1, all overheads for creating a new probe or reevaluating one or many probes is well below 0.1 ms. For example, creating a probe in the largest project (FOP) takes 32 ms, and reevaluating 15 probes after an edit takes 38.3 ms, so they are all within Nielsen's category of appearing as instant.

From the plots in Figs. 18(a) and 18(b), we can see that the time difference between creating or evaluating 5, 10, or 15 probes is fairly constant for a given project. For example, for FOP, the time to evaluate 5, 10, and 15 probes is 32.8, 35.4, and 38.3 ms respectively. This means that the administration time per probe is only around (38.3–32.8)/10 = 0.55 ms. The bulk of the total time is for reparsing the edited file and flushing memoized properties, e.g., around 32.8-0.55*10 = 27.3 ms for FOP. This means that creating many probes is not a problem for performance. It will rather be the screen size that limits how many active probes the user would like to have.

We can also see that the time grows linearly with project size. The main reason for this is not the probe administration, but the parsing and the flushing of memoized properties. The parsing is proportional to the size of the edited file, which should be rather similar in all projects. The flushing of memoized properties is proportional to the size of the AST of the complete project, including all source files. This is because of the way JastAdd implements memoization (as fields in AST node objects). This might be possible to optimize in JastAdd, e.g., by storing memoized properties in a way that is faster to flush, or by using incremental flushing. While the growth is linear, it is still fairly slow, and fairly large projects can be handled without problems.

The benchmarks show results for properties that are very quick to compute. Naturally, in case the user probes a property that takes a long time to compute, the response time will be correspondingly longer. However, if the underlying analysis tool is demand-driven, like the JastAdd-generated tools we have used, the response time can still be very short, even for fairly advanced computations. For example, we have experimented with probes showing generated bytecode for a method and the arrow diagnostics showing control-flow, and they do update directly when editing the code.

The measurements of the full parse is an effect of the underlying analysis tool, rather than CodeProber. However, we wanted to include these numbers to show that it is not a factor that gives any problems. As can be seen in Table 1, the longest time for a full parse (that happens initially at start up), takes 695.1 ms for the largest project (FOP, 97 kLOC), so less than 1 s.

Overall, property probes have shown to be very helpful in exploring how an analysis works, and for implementing and fixing features. The approach fits very well for analysis tools that use on-demand evaluation for individual properties, like the JastAdd-based tools we have tried it on. However, we think the approach can be very useful also for tools that do up-front evaluation, as long as the results can be tied to an AST with source text locations. In this case, properties can be explored interactively, although the user will of course have to wait for a possibly lengthy reanalysis if the underlying source text is edited.

9. Related work

CodeProber allows interactive exploration of properties based on the source code. Earlier tools for debugging and exploring attribute grammars include, for example, Noosa (Sloane, 1999), DrAST (Lindholm et al., 2016), Aki (Ikezoe et al., 2000), and EvDebugger (Rodríguez-Cerezo et al., 2014). They all have ways of showing the syntax tree and attribute values, but none of them have any concept of probes that are updated after changes to the source text.

Noosa is a special-purpose interactive debugger for compilers implemented in the Eli attribute grammar system. It supports, e.g., visualization of the AST, display of attributes of the AST, linking between source text and AST, monitoring the stream of abstract events during data-driven attribute evaluation, and setting breakpoints relating to such events.

DrAST is an interactive tool for visualizing JastAdd ASTs and inspecting AST node properties. It introduces a filtering language to collapse subtrees in order to reduce the visual complexity of the AST, and to specify which attributes to show directly in the tree for certain node types, possibly conditionally. Individual attributes can also be inspected.

Aki is a visual debugger for attribute grammars, supporting algorithmic and slice-based debugging of attributes. Based on attribute dependencies, it can systematically query the user about correctness of attribute values, in order to pinpoint the source of an error. Aki can present the syntax tree for a source program and its attribute values. Aki's debugging techniques are developed for traditional attribute grammars where attributes are evaluated in a data-driven manner. It would be interesting future work to develop similar techniques for RAGs and demand-driven evaluation and integrate them into CodeProber.

EvDebugger supports creating and debugging attribute grammar implementations. The main goal of the tool is to support students in learning attribute grammars, and in particular to illustrate the attribute evaluation process. EvDebugger has support for showing a syntax tree, and stepping through an attribute evaluation process to understand in what order attributes are evaluated. It is based on traditional attribute grammars where evaluation is data-driven, according to static dependencies in the grammar.

While CodeProber is not primarily aimed at teaching attribute grammar evaluation algorithms, it would be interesting to add support for tracing and stepping through a demand-driven evaluation, to help students understand how such an evaluation works, when memoization happens, etc. Additionally, it would be very interesting future work to add support for showing dynamic dependencies between properties, and letting the user navigate this graph to explore property values. This would be very related to the algorithmic/slicing debugging methods mentioned above, allowing a user to explore why a certain property has a certain value.

The Language Server Protocol (LSP)[11] is a widely supported protocol for interaction between editors and language implementations. It supports features like code completion, refactoring, validation, and more. Some of CodeProber's features could be implemented as a language server. For example, evaluating properties that produce diagnostics ("squiggly lines") and presenting them over LSP is possible. However, many features are not possible to replicate within LSP. This includes custom UI such as the probe windows and arrows rendered on top of the code. In addition, node locators depend on the user's input actions to adjust TAL steps, and that information isn't easily accessible over LSP. There is interesting future work in trying to port as much functionality as possible over to LSP.

[11] https://microsoft.github.io/language-server-protocol/. Accessed February 1, 2024.

The concept of node locators has relations to *origin tracking* (Van Deursen et al., 1993). This is a set of techniques for identifying where a node came from after tree rewrites. This is useful, for example, when generating error messages for transformed trees. Origin tracking has also been integrated with attribute grammars with higher-order attributes (Williams and Wyk, 2014) (HOAs), which might be useful for improving locations for HOAs used in our CodeProber.

Node locators also have connections to *edit scripts*. Edit scripts describe differences between two versions of a source file, either in textual or AST form. This can be used to generate detailed program diffs, or track nodes across multiple versions of a source file. However, existing techniques, like GumTree (Falleri et al., 2014), IJM (Frick et al., 2018), MTDIFF (Dotzler and Philippsen, 2016), and TrueDiff (Erdweg et al., 2021), are focused on detecting differences between two files, without any knowledge of the actual input sequence that transformed one file to another. Our node locators require that we have input information available while editing. This is a limitation, but it also makes the algorithm much simpler. It might be possible to derive input information using edit scripts, and thus make it easier to integrate property probes with, for example, LSP.

Property probes in CodeProber can be viewed as being on liveness level 3 out of 6 according to Tanimoto (2013). Probes are automatically updated when either the input source file or the analysis tool have been changed, but do not predict user actions.

Property probes can also be compared to *watch* expressions found in many debuggers. Watch expressions typically only run while a debugging session is running, and the expressions are evaluated in the context of the current debugging session state. Property probes, on the other hand, are always active, and are evaluated without any state (except the source file contents that were used to initially construct the AST).

Beller, Spruit, Spinellis and Zaidman found that “developers spend surprisingly little time in the debugger” (Beller et al., 2018), citing complexity of debuggers as a potential reason. Many developers they surveyed preferred using “print debugging” instead, despite its limitations. This indicates to us that there is a need to develop new ways of exploring/debugging programs. It might be easier to develop debugging tools for particular use cases, like for exploring partial program analysis results. This paper represents one such tool.

Erdweg et al. define *language workbenches* as “tools that support the efficient definition, reuse and composition of languages and their IDEs” (Erdweg et al., 2013). They define a feature model according to which a language workbench must support notation, semantics, and an editor for the defined language. Optional features include testing and debugging of the language definition. For the editor, optional features include support for semantic services like reference resolution (i.e., name binding), semantic completion, error markings, live translation, etc. In relation to language workbenches, CodeProber supports many of these optional features since it can be used for debugging and exploratory testing of the language specification, and it can be used for prototyping semantic language services. So while CodeProber is not a language workbench by itself, it could be used as a component of one.

10. Conclusion

We have presented the concept of property probes, an interactive mechanism for exploring program analyses in terms of the source code.

To support probes to be updated after edits, we introduced node locators with three kinds of steps: Child, TAL, and FN, and illustrated how they are used to robustly map between source code and the nodes in the program representation, and to handle synthetic nodes that have no representation in the source code.

We have developed CodeProber to support property probes, and discussed its client–server architecture and implementation. To validate our work, we successfully applied CodeProber to a number of tools for different languages and analyses, all based on Reference Attribute Grammars. We have also used CodeProber in two courses, and received positive feedback from students. This initial testing has already shown the utility of the tool. We are now using the tool extensively in our own work on program analysis. We have also shown through experiments that the overhead of using probes is very small, even if the analyzed project is large, giving latencies in the interactive tool that are far below the recommended limit of 0.1 s.

CRediT authorship contribution statement

Anton Risberg Alaküla: Conceptualization, Methodology, Software, Investigation, Writing – original draft, Visualization. **Görel Hedin:** Conceptualization, Methodology, Investigation, Writing – review & editing, Supervision, Project administration. **Niklas Fors:** Conceptualization, Methodology, Investigation, Writing – review & editing, Supervision. **Adrian Pop:** Conceptualization, Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Our code is available on Github at https://github.com/lu-cs-sde/codeprober. A snapshot of our code at the time that this article was published is available on Zenodo at https://zenodo.org/doi/10.5281/zenodo.10607231.

Acknowledgments

This work was funded by ELLIIT – Excellence Center at Linköping-Lund on Information Technology and by the Swedish Foundation for Strategic Research. We would like to thank Idriss Riouak, main developer of IntraJ, for continuously testing and giving feedback during the development of CodeProber. We would also like to thank Christoph Reichenbach, course leader for the program analysis course where CodeProber was used, for his feedback and his contributions to CodeProber.

References

Beller, Moritz, Spruit, Niels, Spinellis, Diomidis, Zaidman, Andy, 2018. On the dichotomy of debugging behavior among programmers. In: Proceedings of the 40th International Conference on Software Engineering. pp. 572–583. http://dx.doi.org/10.1145/3180155.3180175.

Dotzler, Georg, Philippsen, Michael, 2016. Move-optimized source code tree differencing. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 660–671. http://dx.doi.org/10.1145/2970276.2970315.

Ekman, Torbjörn, Hedin, Görel, 2007a. The jastadd extensible java compiler. SIGPLAN Not. 42 (10), 1–18. http://dx.doi.org/10.1145/1297105.1297029.

Ekman, Torbjörn, Hedin, Görel, 2007b. The JastAdd system - modular extensible compiler construction. Sci. Comput. Program. 69 (1–3), 14–26. http://dx.doi.org/10.1016/j.scico.2007.02.003.

Erdweg, Sebastian, Szabó, Tamás, Pacak, André, 2021. Concise, type-safe, and efficient structural diffing. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 406–419. http://dx.doi.org/10.1145/3453483.3454052.

Erdweg, Sebastian, Van Der Storm, Tijs, Völter, Markus, Boersma, Meinte, Bosman, Remi, Cook, William R, Gerritsen, Albert, Hulshout, Angelo, Kelly, Steven, Loh, Alex, et al., 2013. The state of the art in language workbenches: Conclusions from the language workbench challenge. In: Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, in, USA, October 26-28, 2013. Proceedings 6. Springer, pp. 197–217. http://dx.doi.org/10.1007/978-3-319-02654-1_11.

Falleri, Jean-Rémy, Morandat, Floréal, Blanc, Xavier, Martinez, Matias, Monperrus, Martin, 2014. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 313–324. http://dx.doi.org/10.1145/2642937.2642982.

Fors, Niklas, Hedin, Görel, 2015. A JastAdd implementation of oberon-0. Sci. Comput. Program. 114, 74–84. http://dx.doi.org/10.1016/j.scico.2015.02.002.

Fors, Niklas, Hedin, Görel, 2016. Bloqqi: Modular feature-based block diagram programming. In: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. In: Onward! 2016, Association for Computing Machinery, New York, NY, USA, pp. 57–73. http://dx.doi.org/10.1145/2986012.2986026.

Frick, Veit, Grassauer, Thomas, Beck, Fabian, Pinzger, Martin, 2018. Generating accurate and compact edit scripts using tree differencing. In: 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 264–274. http://dx.doi.org/10.1109/ICSME.2018.00036.

Hedin, Görel, 2000. Reference attributed grammars. Informatica (Slovenia) 24 (3), 301–317.

Ikezoe, Yohei, Sasaki, Akira, Ohshima, Yoshiki, Wakita, Ken, Sassa, Masataka, 2000. Systematic debugging of attribute grammars. In: Ducassé, Mireille (Ed.), Proceedings of the Fourth International Workshop on Automated Debugging. AADEBUG 2000, Munich, Germany, August 28-30th, 2000, URL https://arxiv.org/abs/cs/0011029.

Lindholm, Joel, Thorsberg, Johan, Hedin, Görel, 2016. Drast: an inspection tool for attributed syntax trees (tool demo). In: van der Storm, Tijs, Balland, Emilie, Varró, Dániel (Eds.), Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. Amsterdam, the Netherlands, October 31 - November 1, 2016, ACM, pp. 176–180, URL http://dl.acm.org/citation.cfm?id=2997378.

Melnikov, Alexey, Fette, Ian, 2011. The WebSocket Protocol. RFC Editor, http://dx.doi.org/10.17487/RFC6455, Request for Comments 6455 URL https://www.rfc-editor.org/info/rfc6455.

Nielsen, Jakob, 1993. Response times: the three important limits. In: Usability Engineering. Academic Press.

Padhye, Rohan, Sen, Koushik, Hilfinger, Paul N., 2019. Chocopy: A programming language for compilers courses. In: Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E. pp. 41–45. http://dx.doi.org/10.1145/3358711.3361627.

Riouak, Idriss, Reichenbach, Christoph, Hedin, Görel, Fors, Niklas, 2021. A precise framework for source-level control-flow analysis. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 1–11. http://dx.doi.org/10.1109/SCAM52516.2021.00009.

Risberg Alaküla, Anton, Hedin, Görel, Fors, Niklas, Pop, Adrian, 2022. Property probes: Source code based exploration of program analysis results. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering. pp. 148–160. http://dx.doi.org/10.1145/3567512.3567525.

Rodríguez-Cerezo, Daniel, Henriques, Pedro Rangel, Sierra, José-Luis, 2014. Attribute grammars made easier: EvDebugger a visual debugger for attribute grammars. In: 2014 International Symposium on Computers in Education. SIIE, pp. 23–28. http://dx.doi.org/10.1109/SIIE.2014.7017699.

Sloane, Anthony M., 1999. Debugging eli-generated compilers with noosa. In: Jähnichen, Stefan (Ed.), Compiler Construction, 8th International Conference, CC'99, Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, the Netherlands, 22-28 March, 1999, Proceedings. In: Lecture Notes in Computer Science, vol. 1575, Springer, pp. 17–31. http://dx.doi.org/10.1007/978-3-540-49051-7_2.

Tanimoto, Steven L., 2013. A perspective on the evolution of live programming. In: Burg, Brian, Kuhn, Adrian, Parnin, Chris (Eds.), Proceedings of the 1st International Workshop on Live Programming. LIVE 2013, San Francisco, California, USA, May 19, 2013, IEEE Computer Society, pp. 31–34. http://dx.doi.org/10.1109/LIVE.2013.6617346.

Van Deursen, Arie, Klint, Paul, Tip, Frank, 1993. Origin tracking. J. Symbolic Comput. 15 (5–6), 523–545. http://dx.doi.org/10.1016/S0747-7171(06)80004-0.

Vogt, Harald, Swierstra, S. Doaitse, Kuiper, Matthijs F., 1989. Higher-order attribute grammars. In: Wexelblat, Richard L. (Ed.), Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation. PLDI, Portland, Oregon, USA, June 21-23, 1989, ACM, pp. 131–145. http://dx.doi.org/10.1145/74818.74830.

Williams, Kevin, Wyk, Eric Van, 2014. Origin tracking in attribute grammars. In: Combemale, Benoît, Pearce, David J., Barais, Olivier, Vinju, Jurgen J. (Eds.), Software Language Engineering - 7th International Conference, SLE 2014, VäSteråS, Sweden, September 15-16, 2014. Proceedings. In: Lecture Notes in Computer Science, vol. 8706, Springer, pp. 282–301. http://dx.doi.org/10.1007/978-3-319-11245-9_16.

Anton Risberg Alaküla is a PhD student in Computer Science at Lund University, Sweden. His research interests include programming languages and cloud computing. He received his MSc. in Computer Science and Engineering in 2014 from Lund University, and spent 7 years in industry before starting his PhD.

Görel Hedin is a professor at Lund University, Sweden, where she heads the Software Development and Environments group. Her research is focused on the generation of language tools like compilers, visual program editors, and source code analyzers. She has developed reference attribute grammars as a technique for generating extensible language tools. She leads the development of the open source metacompiler JastAdd that is the main tool for reference attribute grammars, and which is used in both academia and industry.

Niklas Fors is an Associate Senior Lecturer at Lund University, Sweden. His primary research interests are tooling for software languages, program analysis and reference attribute grammars. In his PhD thesis, he presented a feature-based diagram language Bloqqi for improving code reuse in automation programming. He also teaches introductory programming for engineering students.

Adrian Pop is a research engineer at Linköping University, Sweden. His primary research interests are in programming languages (design, implementation, debugging, etc.) and integrated development tools. He is one of the main contributors and the technical coordinator of the open-source OpenModelica compiler targeting the equation-based object-oriented language Modelica for modeling and simulation of cyber-physical systems.