ISSN 2079-3316 **PROGRAM SYSTEMS: THEORY AND APPLICATIONS** **vol. 13**, No 1(52), pp. 35–62
Research Article THEORETICAL COMPUTER SCIENCE

UDC 004.4'23+004.415+004.02
10.25209/2079-3316-2022-13-1-35-62

Robust algorithmic binding to arbitrary fragment of program code

Alexey V. **Goloveshkin**✉, Stanislav S. **Mikhalkovich**
Southern Federal University, Rostov-on-Don, Russia,
alexeyvale@gmail.com✉ (*learn more about the authors on p. 62*)

Abstract. When solving a task, a programmer actively interacts with a finite set of code fragments. The information about their locations is important for quick navigation, for other developers, and as a kind of documentation. Integrated development environments (IDEs) provide tools for marking code fragments with labels, displaying lists of labels, and using these labels for quick navigation. However, they often lose the correspondence between the label and the marked place when the code is edited, in particular when changes are made outside the IDE.

In previous works, the authors propose a tool to be integrated into various IDEs for "binding" to large syntactic entities of a program and building a markup that is robust to code editing. The description of the marked element is built on the basis of the abstract syntax tree (AST) of the program. Later it is used to algorithmically search for the element in an edited code. The search has a success rate from 99 to 100%.

This article aims at robust algorithmic binding to an *arbitrary* section of the code. For binding to a single-line code fragment, we propose an extension of the model describing the marked fragment, and an additional search algorithm. We also propose an algorithm for embedding nodes corresponding to multi-line fragments in an AST. We show that the correctness of the AST is not violated by these embeddings. Bindings to randomly selected lines were made in the code of three large C# projects. Manual check of these lines search results in the edited code has confirmed that the bindings are robust to code editing.

Key words and phrases: program markup, algorithmic binding to program code, software development, abstract syntax tree, code similarity measurement

2020 *Mathematics Subject Classification*: 68U99; 68W05

For citation: Goloveshkin A. V., Mikhalkovich S. S. *Robust algorithmic binding to arbitrary fragment of program code* // Program Systems: Theory and Applications, 2022, **13**:1(52), pp. 35–62. `http://psta.psiras.ru/read/psta2022_1_35-62.pdf`

Эта статья по-русски: *http://psta.psiras.ru/read/psta2022_1_3-33.pdf*

Introduction

Navigating through the code is one of the main activities of a programmer when working on some task. It is intensively performed at all stages of work: from the initial exploring of the code to the final review of the changes made [1, 2]. The most active transitions take place within a finite set of code fragments which are of particular interest in the context of the task being solved. Preserving the information about locations of these fragments for quick navigation between them, and sharing this information with other developers are the problems programmers solve with some "analogue" things and tools that are not directly intended for it [3, 4]. The mechanisms offered by integrated development environments (IDEs) for marking code fragments — associating some labels with them and using these labels for quick access to code — are inconvenient and do not allow one to link additional high-level semantic information, related to the problem domain, to the marked fragments. More importantly, the markup quickly becomes irrelevant to the current state of the code: the connection between the label and the marked fragment can be lost due to code editing. As a consequence, programmers rarely use built-in IDE features for preserving information about important code fragments [5, 6].

In our research [7–9], models, algorithms, and tools that allow the programmer to mark code fragments of interest (to *bind* to them), to accompany labels with detailed comments and organize them as a hierarchical structure of a custom depth, to save and load the markup are proposed. The labels are bound to the code *algorithmically*: when the programmer marks a certain fragment, the program is parsed and a model describing the given fragment is constructed, based on the obtained abstract syntax tree (AST) (see Section 2). When navigating to some previously marked fragment is needed, this model is used for algorithmic search for the marked fragment in the current version of the program. The algorithms restore the correspondence between the labels and the fragments (*rebind* the labels) even if the fragment or its surroundings are edited. Thus, the markup is *robust* to code editing.

Figure 1 shows a fragment of the Visual Studio window. On the left, there is our markup panel integrated into the IDE. A visible part of the markup opened in the panel consists of 8 labels grouped according to some domain semantics.

The models and algorithms proposed in [9] are designed to bind to large-scale and medium-scale syntactic entities of a program, for example, classes and class members for a C# program . It is often necessary to mark not the whole method or class, but a specific line of code or several

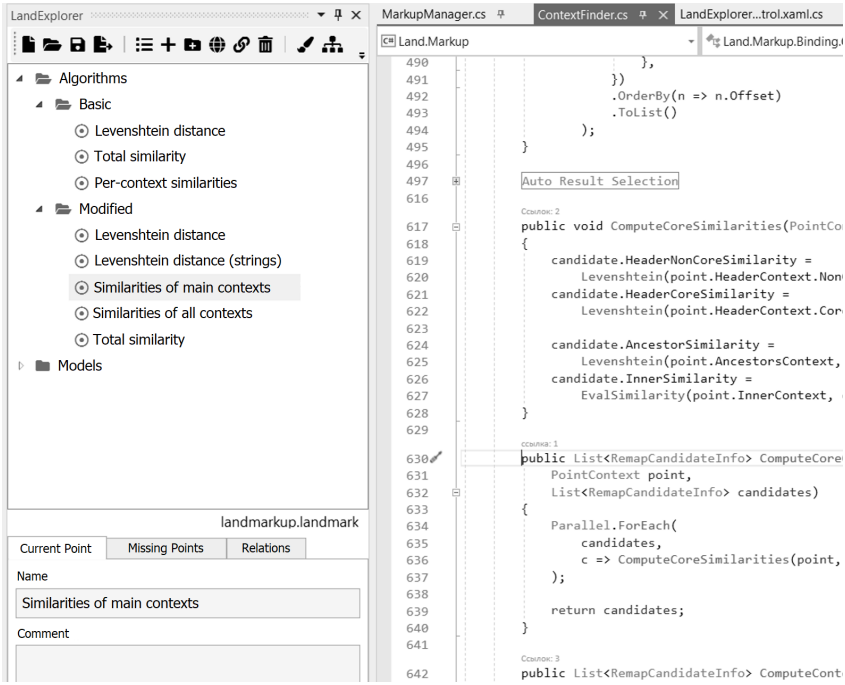


Figure 1. Fragment of the Visual Studio IDE window with a built-in markup panel.

adjacent lines. This task requires the development of additional algorithms, as well as the modification of the models we have previously created. The problem of binding to multi-line code fragments was raised by us in [10]. It was noted that binding to several adjacent lines can be performed in the same way as binding to a syntactic entity, but requires that a node corresponding to this multi-line fragment exists in the AST of the program. In the current paper, this idea is further developed.

The main contributions of the present research are:

(1) an extension of the model previously used for binding to syntactic entities and a modification of the rebinding algorithm, which provide robust binding to a single line of code;

(2) a formal concept of custom block — an auxiliary entity that helps to bind to multi-line code fragments; an algorithm that embeds nodes corresponding to custom blocks into an AST while maintaining the correctness of an AST;

(3) experimental results demonstrating the successful application of

the described models and algorithms for robust binding to code fragments.

The remainder of the paper is organized as follows. Section 1 provides an overview of the research related to the binding and rebinding problems. Section 2 briefly outlines some results previously obtained by the authors, which are the basis for the current research. Section 3 provides models and algorithms developed to make the markup tool capable of binding to an arbitrary code fragment, both single-line and multi-line. In Section 4, experiments are conducted to evaluate the robustness of the code markup made with the described models and algorithms.

1. Related work

1.1. Binding to code

The problem of "memorizing" some characteristics by which a certain place in the code can be found is well-known. In A.L. Fuksman's research [**11**], to obtain an integrated program by adding extension functions to some basis, it is required to unambiguously identify points of the basis where fragments of the extension functions need to be embedded. This identification is supposed to be done by text coordinates — a line number and a column number. In the modern paradigms that are also based on the idea of dividing a program into a basis and extension functions being added to it [**12**–**14**], it is proposed to identify places where extensions should be inserted by class names, method names, signatures, possibly using patterns. All of the methods above are largely unreliable when the code is being edited, and cannot be used for binding in the markup task.

Another approach to finding the right place in the codebase is to frame this place directly in the code using *pseudo-comments* — comments of a special form, processed by the language compiler in the way ordinary comments are processed, but having a special meaning for some external tool. Pseudo-comments are intensively used in various software development and maintenance systems [**15**, **16**]. This approach is robust to any code editing, but it stops working if pseudo-comments themselves are accidentally changed. It is also unpleasant that pseudo-comments "litter" the code with purely technical information.

The approach developed in this work is based on the idea of representing a marked fragment as a set of special structures called *contexts* and storing this representation separately from the code [**18**, **19**]. In [**9**], we propose context models that capture enough information to successfully search for large-scale and medium-scale syntactic entities later in the edited

program. In the present research we extend this technique to arbitrary code fragments.

1.2. Code search

There are problems related to the problem of robust binding to code: searching for code clones [20–22], plagiarism detection [23, 24], code recommendation [25]. Like robust binding, they belong to the wider scope of software evolution analysis. Their common subproblem is to evaluate the similarity of programs. However, each problem has its own specifics and additional assumptions imposed on the codebases compared. The problem of robust binding to code has the following distinctive features:

- it is guaranteed that two versions of the same codebase are compared;
- for each fragment marked in the initial version of the code, there is only one match in the current version of the code, or there is no match at all.

Separate studies are devoted to restoring the current code editing scenario from the available initial and current version of the codebase [26–29], as well as predicting further edits [30]. To achieve good results in this field, a detailed analysis of the program is required using a full parser of the corresponding language, and taking into account some features of the particular language may be important in further comparison. In our approach, we rely on a lightweight analysis of the program structure and then process the lightweight abstract syntax tree. We parse the program down to the syntactic entities that we are capable of binding, and work with the plain text at a greater depth. The models and algorithms we use for binding and rebinding are language independent and can be applied with minimal cost to any structured text.

It can also be noted that the algorithm we propose for finding marked single-line fragments is conceptually close to the LHDiff [31] algorithm, which is developed to track a line of code based on the line content and surroundings. However, unlike [31], we do not believe that the version of the code that existed at the time the line was “memorized” is always available. Even the use of a version control system cannot guarantee such availability, since the need to bind to something may arise when the code is in some intermediate state between the commits. We save a limited amount of information describing the marked line and its surroundings and rely only on this stored information during the subsequent search. Before searching, we also narrow the search scope down to an enclosing syntactic entity.

2. Preliminaries

2.1. Lightweight parsing

One of the main requirements when solving the robust binding problem is the language independence of models and algorithms. First, the markup tool is supposed to be embedded in different IDEs for different languages. Secondly, several programming languages are often used in the same project, and the programmer needs to preserve the information about code fragments written in different languages in the same markup. The requirement of language independence is also applied to the format in which abstract syntax trees processed by the tool must be represented.

The second requirement for an AST is its lightweightness. The contexts describing the marked fragment are developed under the assumption that binding is done on the elements of a coarse-grained program structure — large-scaled and medium-scaled syntactic entities. The definitions of such entities contain a fairly considerable amount of information useful for further search. Large pieces of the program, for example, all method bodies, can be left unparsed and not structured in the tree. At their level, the program is treated as a plain text. Lightweight ASTs are easier to construct and faster to process.

To add support for another language, an existing production parser of the language can be used. In this case, the main difficulty is to find out how elements suitable for binding are structured in an AST built by the parser: this information is vitally important for the correct construction of models describing the marked element. To collect the information, one has to investigate the complete language specification (if available). In [32, 33], it is noted that this approach is very time-consuming. In addition, it is necessary to implement an AST converter that prunes the tree, cutting off unnecessary details, and converts the tree into a unified language-independent format. A more promising approach is to implement a lightweight grammar where only syntactic entities suitable for binding are described in accordance with one's own understanding of the language syntax, and the structure of the rest of the program is omitted. From this grammar, a lightweight parser can be generated. Such grammars are also called *island grammars* [32]. In the island grammars approach, the entities described in detail are supposed to be important in the context of the task for which the grammar is developed. Such entities are called *islands*. Other areas of the program are called *water*. In [7, 8], we further developed the concept of island grammars and modified the LL(1) and LR(1) parsing algorithms to make them able to process such grammars. We also created LanD parser generator which is capable to

```
enum = common 'enum' name Any '{' Any '}' ';'?
class_struct_interface = common CLASS_STRUCT_INTERFACE name Any '{' entity* '}' ';'?
method = common type name arguments Any (init_expression? ';' | block)
field = common type name ('[' Any ']')? init_value? (',' name ('[' Any ']')? init_value?)* ';'
property = common type name (block (init_value ';')? | init_expression ';')
water_entity = AnyInclude('delegate', 'operator', 'this') (block | ';')+

common = entity_attribute* modifier*
entity_attribute = '[' Any ']'
arguments = '(' Any ')'
block = '{' Any '}'
```

Figure 2. Fragment of the lightweight LR(1) grammar of C#.

generate lightweight parsers, and a special language for describing island grammars for this generator. With LanD, we implemented lightweight parsers for a large number of languages to use in the markup tool. Figure 2 demonstrates a fragment of the lightweight LR(1) grammar of C#. There is a special token `Any` used to indicate areas whose structure is unimportant and should not be analysed and presented in an AST.

2.2. Binding to large-scale syntactic entities

In our recent research [9], we proposed models and algorithms for robust binding to syntactic entities of the program. Each marked entity a is described by a tuple of the following form:

$$\mathbf{BindingPoint_a} = (Type_a, H_a, I_a, S_a, N_a, C_a) \ .$$

Herein, $Type_a$ is a nonterminal grammar symbol corresponding to a. Since binding is based on the analysis of the AST, binding to a actually means binding to the corresponding tree node. In general discussions of tuple components, the lower index referring to the particular entity may be omitted further in the paper.

H_a is the *header* context stored as a list of quads of the form $(Type, Priority, ComparisonMode, Words)$, one per a leaf child[1] of a. Figure 3 shows an example of a method header and a header context built for the method. *Type* is a grammar symbol corresponding to the header element, *Priority* is a non-negative real number that reflects the importance of matching the element when comparing two headers. $ComparisonMode \in \{\text{“}Distance\text{”}, \text{“}ExactMatch\text{”}\}$ defines how to calculate the similarity of two elements of the type *Type*. For example, when

[1]Speaking of trees, by the children of some node we mean the immediate descendants of this node, and by the node parent we mean the immediate ancestor.

```
public override void Visit(BinOpNode binop) { … }
```

H_a = (

(MODIFIER,	0.1,	"ExactMatch",	((1, "public")),
(MODIFIER,	0.1,	"ExactMatch",	((1, "override")),
(type,	0.3,	"Distance",	((1, "void")),
(name,	0.8,	"Distance",	((1, "Visit")),
(arguments,	0.5,	"Distance",	((0.1, "("), (1, "Bin"), (1, "Op"), (1, "Node"), (1, "binop"), (0.1, ")")))

)

FIGURE 3. Header context built for a method.

comparing method names (`name`), an edit distance should be calculated, and when comparing modifiers (`MODIFIER`), it only makes sense to check for a strict match. *Words* is a list constructed by dividing the text corresponding to the header element into alphanumeric "words" and other characters. The list item is a pair (*Priority*, *Text*), where *Priority* is set to 1 for an alphanumeric text and equals 0.1 for other characters. The operators Core(H_a) and NotCore(H_a) are also defined for H_a, introducing an additional grouping of header elements according to their impact on the correct rebinding. Operators return non-overlapping lists of H_a elements.

I_a is the *inner* context stored as a triple of the form (*Text*, *Hash*, *Length*) further referred as **TextOrHash**. *Text* is a text of the inner part of the element (the text corresponding to a, minus whitespaces and fragments corresponding to the leaf children of a). *Hash* is a fuzzy hash [**34**] constructed for this text. If the length of the text exceeds a certain value set with the binding algorithm parameter, only the fuzzy hash is stored, and the *Text* component remains empty. For short texts, the hash cannot be constructed due to technical limitations of the applied fuzzy hashing algorithm, and only the text itself is saved. The length of the text is always stored in *Length*.

S_a is the *scope* context, stored as a list of $(Type, H)$ pairs. Each element corresponds to one of the entities enclosing a (one of the ancestors of a), and is represented by its type and header.

N_a is the *neighbours* context, stored as a pair $(Before, After)$, where both components are of the form $(All, Nearest)$. *Before* describes the neighbours preceding a, and *After* describes the neighbours following a. *All* is a **TextOrHash** triple, built for the concatenated text of all siblings of a (all entities suitable for binding that have a common parent with a). *Nearest* is a **BindingPoint** built for the nearest (in terms of location in the text) neighbour of a, that has the type $Type_a$. For example, if a is a method, All(Before(a)) and All(After(a)) contain

information about all class members located before and after a, respectively. In Nearest(Before(a)), there is information about the nearest preceding method, possibly located in another class. Nearest(After(a)) is built in a similar way.

C_a is the *closest* context. It is a list of no more than $n_C \in \mathbb{N} \cup \{0\}$ **BindingPoint** tuples corresponding to the most similar entities of the type $Type_a$, presented in the program at the moment of binding to a. For example, if a is an overloaded method, its other implementations are memorized in C_a.

When the programmer requests a transition to the previously marked entity a (clicks on the corresponding element in the markup panel), rebinding is performed: a is searched in the current version of the code. An AST of the current version of the program is built and a list of candidates is formed — an array of **BindingPoint** tuples corresponding to the entities of the $Type_a$ type, presented in the code. Then, for each candidate c, a context-wise comparison of $\mathbf{BindingPoint}_a$ and $\mathbf{BindingPoint}_c$ is performed and a vector of distances $\mathbf{d_{ac}}$ is calculated:

$$\begin{aligned}\mathbf{d_{ac}} =&(\mathrm{Dist}(\mathrm{Core}(H_a), \mathrm{Core}(H_c)), \mathrm{Dist}(\mathrm{NotCore}(H_a), \mathrm{NotCore}(H_c)),\\ &\mathrm{Dist}(I_a, I_c), \mathrm{Dist}(S_a, S_c), \mathrm{Dist}(\mathrm{All}(N_a), \mathrm{All}(N_c)),\\ &\mathrm{Dist}(\mathrm{Nearest}(N_a), \mathrm{Nearest}(N_c))).\end{aligned}$$

Distances are real numbers from the range $[0, 1]$. For each of the contexts, the distance evaluates the difference between the context describing a and the corresponding context describing c. The distance vector is scalarly multiplied by the vector of weights $\mathbf{w_a}$. Weights are real numbers reflecting the importance of each of the contexts for finding the marked element. The vectors of weights are constructed independently for every searched element using a heuristic algorithm. The final score is computed as follows:

$$\mathrm{Dist}(a, c) = \frac{\mathbf{d_{ac}} \cdot \mathbf{w_a}}{\sum\limits_{w \in \mathbf{w_a}} w}.$$

$\mathrm{Dist}(a, c) \in [0, 1]$ is the distance between the initial version of the entity and the currently available candidate. Then the candidates are sorted by increasing distance. The best candidate c_1 is the one with the minimum distance[2]. If c_1 and the next candidate c_2 satisfy the condition $\mathrm{Dist}(a, c_2) \neq 0 \wedge \mathrm{Dist}(a, c_1) \cdot 2 \leq \mathrm{Dist}(a, c_2)$, an automatic rebinding and transition to c_1 in the editor window occurs, otherwise the user of the markup tool is provided with the ordered list of candidates, and the correct match is selected manually.

[2]Hereinafter, array indexing is one-based.

3. Binding to arbitrary code fragment

3.1. Binding to single line

Binding to a single line is performed through binding to an AST node and storing some additional information. The previously introduced **BindingPoint** model is extended with an additional *line* context $L = (HadSame, Inner, Outer)$. It consists of a line duplication flag, an inner context, and an outer context, respectively. *Inner* is a **TextOrHash** triple, built for the text of the marked line. $Outer = (Before, After)$ is a pair of **TextOrHash** triples built for the concatenated text of all preceding and all succeeding lines, respectively, which are located together with the marked line within the same smallest enclosing syntactic entity that can be presented as a **BindingPoint**. For example, when binding to a line inside a method, *Before* contains information about all preceding lines belonging to the same method, and *After* contains information about all subsequent ones. *HadSame* is equal to 1 if, when binding, there are other lines inside the smallest enclosing syntactic entity whose contents are identical to the content of the marked one. Otherwise, it is equal to 0.

The binding is carried out in two steps. In the first step, a search is made for the smallest entity a that encloses the line of interest. This entity is memorized as $\mathbf{BindingPoint}_a$ with an empty L_a context. In the second step, L_a is constructed to describe the line. Note that our implementation is optimized to avoid duplication of contexts describing the a entity when binding to other lines within it.

The rebinding process also consists of two steps. First, the entity a is searched for in the current version of the code, then, if there is a successful rebinding of a to some c, Algorithm 1 is started. The text corresponding to c is split into candidate lines. For each candidate line, the context L_c describing it and the vector of component-wise distances between L_a and L_c are calculated (lines 2–6 of Algorithm 1). Then, depending on whether there were lines identical to the marked one at the time of binding, and whether there are candidate lines similar to the searched one that are identical or almost identical in their contents, the weights of the components *Inner* and *Outer* are heuristically adjusted (lines 7–10 of Algorithm 1). The $MaxW$ and $MinW$ parameters specify the maximum and minimum weight of the context. The results given in Section 4 are obtained with $MaxW = 1$ and $MinW = 0.25$. For each candidate line, the total distance is calculated as the scalar product of the vector of component-wise distances and the vector of weights and then normalized to the range $[0, 1]$. The candidate with the smallest distance is considered the current version of the searched line.

Algorithm 1 Single line rebinding.

function Rebind(a, c)
  $lines \leftarrow$ GetLines(c)
  $lineContexts \leftarrow \{line \in lines \mid \text{GetLineContext}(line)\}$
  for all $L_c \in lineContexts$ **do**
    $distances[L_c] \leftarrow (\text{Dist}(\text{Inner}(L_a), \text{Inner}(L_c)), \text{Dist}(\text{Outer}(L_a), \text{Outer}(L_c)))$
  end for
  $minInDist \leftarrow \text{Min}_{L_c \in lineContexts}(distances[L_c][1])$
  $confusionFlag \leftarrow \text{HadSame}(L_a) \vee |\{L_c \in lineContexts \mid !\,\text{CheckGap}(minInDist, distances[L_c][1])\}| > 1$
  $w^{\text{Inner}(L)} \leftarrow confusionFlag\,?\,MinW : MaxW$
  $w^{\text{Outer}(L)} \leftarrow confusionFlag\,?\,MaxW : MinW$
  for all $L_c \in lineContexts$ **do**
    $totalDistances[L_c] \leftarrow \frac{distances[L_c] \cdot \left(w^{\text{Inner}(L)}, w^{\text{Outer}(L)}\right)}{\left(w^{\text{Inner}(L)} + w^{\text{Outer}(L)}\right)}$
  end for
  $orderedCandidates \leftarrow \text{OrderByAsc}\,(lineContexts, totalDistances)$
  return $orderedCandidates[1]$
end function

function CheckGap(v_1, v_2)
  return $v_2 \neq 0 \wedge v_1 \cdot 2 \leq v_2$
end function

The distance between two **TextOrHash** triples u and v is evaluated using the similarity function $\text{Sim}(u, v)$ given in [9], which is interconnected with the distance function by the formula $\text{Dist}(u, v) = 1 - \text{Sim}(u, v)$. Here

$$\text{Sim}(u,v) = \begin{cases} \theta\Big(1 - \text{Dist}(\text{Text}(u), \text{Text}(v)), \frac{L_1}{L_2}\Big), \\ \qquad \forall k \in \{u, v\}, \text{Length}(k) \leq L_2 \\ \theta\Big(1 - \text{TlshDist}(\text{Hash}(u), \text{Hash}(v)), \frac{L_1}{L_2}\Big), \\ \qquad \forall k \in \{u, v\}, \text{Length}(k) \geq L_1 \\ 0, \quad \text{otherwise}, \end{cases}$$

where the distance between texts is calculated as the Levenshtein distance [35] normalized to the range $[0, 1]$, and TlshDist is a function that returns the normalized distance between two fuzzy hashes. θ is a threshold function that returns the first argument if its value is greater than or equal to the second argument, otherwise the second argument is returned. L_1

and L_2 are parameters of the binding algorithm. L_1 is the minimum length of the text for which it is possible to calculate a fuzzy hash; L_2 is the maximum length of the text to store it in the *Text* component. In our experiments, $L_1 = 25$ in accordance with the technical limitations of the hashing algorithm used, and $L_2 = 100$.

For *Outer*, the distance is calculated by the formula

$$\begin{aligned}\mathrm{Dist}(\mathrm{Outer}(L_a), \mathrm{Outer}(L_c)) = \\ (\mathrm{Dist}(\mathrm{Before}(\mathrm{Outer}(L_a)), \mathrm{Before}(\mathrm{Outer}(L_c))) \\ + \mathrm{Dist}(\mathrm{After}(\mathrm{Outer}(L_a)), \mathrm{After}(\mathrm{Outer}(L_c))))/2\,.\end{aligned}$$

3.2. Multi-line binding

If it is necessary to bind to a multi-line code fragment as a whole (such a fragment can consist of only a few lines or be quite lengthy, including several syntactic entities), the borders of the fragment being marked must be indicated explicitly so that it can be detected at the stage of parsing the program and the corresponding node can be placed in the AST. After that, the models and algorithms described in 2.2 can be used, since such a bordered fragment can be considered as a large-scale syntactic entity. Note that binding to multi-line fragments is not completely arbitrary. Due to the fact that the node corresponding to the multi-line area needs to be added to the AST, this area must satisfy certain requirements, which are discussed later in this section.

3.2.1. *Detecting bordered multi-line fragment at parsing stage*

The first option for recognizing a multi-line fragment, which is bordered in some way, at the parsing stage is to modify the rules of the lightweight grammar of the target programming language. However, as we noted in [**10**], this approach has some serious drawbacks. The most important one is the complication of the grammar. One needs the ability to bind to multi-line fragments in different places of the program. For example, inside a method, as well as at the level of class members. The content of the fragment depends on the specific place in which it is located. To describe a bindable multi-line area, each possible place requires a separate nonterminal symbol and a separate alternative added to the existing grammar rule to make that symbol reachable.

We adhere to a fundamentally different approach. The only thing that needs to be provided for the lightweight parser is the description of language constructs that frame multi-line fragments marked by the programmer. For this, a special configuration section `CustomBlock` is added to the grammar written in the LanD language. Figure 4 shows variants

```
%CustomBlock {
    start("//+")
    end("//-")
    basetoken COMMENT
}
```

(*a*) borders are pseudo-comments

```
%CustomBlock {
    start("#region")
    end("#endregion")
    basetoken DIRECTIVE
}
```

(*b*) borders are region borders

Figure 4. Configuration of binding to multi-line fragments for a C# program

of configuration for C#. The `basetoken` option specifies a token that can be interpreted as an opening and closing border of the multi-line area being marked. The `start` option sets a prefix that the value of the token specified in `basetoken` must have to be considered an opening border. The `end` option has similar meaning for a closing border. The descriptive part of the border — everything that comes after `start` and `end` in the text of the corresponding token — can contain any text according to the needs of a programmer who marks up the code. Thus, borders have the same significance as ordinary lexemes of the type `basetoken`. Note that, in order to avoid collisions between syntactic entities presented in a program and bordered multi-line fragments, the token used as `basetoken` must correspond to something that is not taken into consideration when parsing, such as a comment or a preprocessor directive.

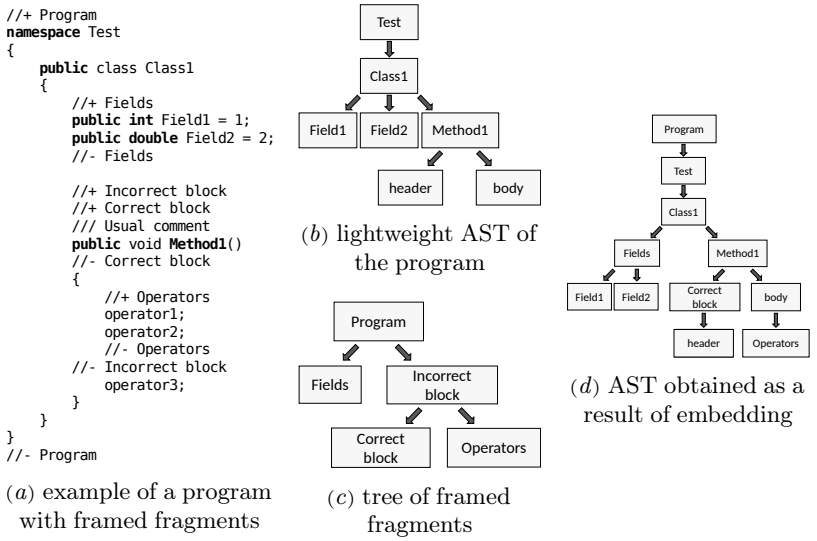
Figure 4a demonstrates the configuration for the case where multi-line fragments are bordered with pseudo-comments. Figure 5a shows the program text in which several multi-line fragments are bordered in this way. Figure 4b offers another configuration — marked fragments can be enclosed in regions, so the developer can not only bind to them, but also better structure the code in the editor window.

Definition 3.1. *Framed* fragment is a continuous fragment of code, consisting of zero or more lines, bordered in accordance with the description provided in the `CustomBlock` section.

When parsing, opening and closing borders of framed fragments are tracked, and in addition to an AST, in which framed fragments are not presented, a forest is built in which each node corresponds to a certain framed area. Then this forest is embedded in the AST. We describe this process in detail below.

3.2.2. *Representing framed fragments in AST*

Figure 5 shows an example of a program and structures that are built by our lightweight parser. In the demonstrated case, the forest of framed fragments consists of one tree, since there is a fragment that encloses the



(*a*) example of a program with framed fragments

(*b*) lightweight AST of the program

(*c*) tree of framed fragments

(*d*) AST obtained as a result of embedding

FIGURE 5. Embedding the tree of framed fragments in the AST

entire program. At the end of the parsing, the forest is embedded in the AST. The result is the AST, in which there are nodes corresponding to the framed fragments (Figure 5d). One can bind to framed fragments, whose nodes are successfully embedded in the AST.

We additionally regulate the appearance of the borders of a framed fragment. An attempt to embed the fragment is made only if its opening and closing borders, with the exception of the prefixes specified by the options `start` and `end`, match, and the matching part is not empty. As a result, the markup tool recognizes potentially incorrect edits of the program that violate the integrity of several marked areas at once. The tool can inform the programmer about this problem. An example of such editing is shown in Figure 6.

In addition to the limitation above, we impose a number of conditions on the framed fragment that are required for successful embedding in the AST. These conditions ensure that the framed fragment is consistent with the syntactic structure of the program. All of them are presented in Definition 3.5.

Let a and b be the nodes of one or different trees built on the text of the same program. We define several relations on the set of nodes of these trees.

```
namespace Test
{
    public class Class1
    {
        public void Method1()
        {
            //+ First fragment
            operator1;
            operator2;
            //- First fragment
            operator3;
            //+ Second fragment
            operator4;
            operator5;
            operator6;
            //- Second fragment
        }
    }
}
```

(*a*) initial version with two framed fragments

```
namespace Test
{
    public class Class1
    {
        public void Method1()
        {
            //+ First fragment
            operator1;
            operator5;
            operator6;
            //- Second fragment
        }
    }
}
```

(*b*) current version

FIGURE 6. Potentially incorrect program editing

Definition 3.2. We say the node a is *nested* in the node b, and denote this fact as $a \subseteq b$ if and only if the fragment of the program text corresponding to the node a is strictly nested in the fragment of the program text corresponding to node b, or matches it.

Similarly, strict nesting of nodes is defined, denoted as $a \subset b$, and textual match of nodes is introduced, denoted as $\overset{\mathrm{T}}{=}$. Note that $a \subseteq b \equiv a \subset b \vee a \overset{\mathrm{T}}{=} b$. A text match and a regular match are not the same thing, as two distinct nodes of the same or different trees may match textually.

Definition 3.3. We say the node a *intersects* the node b, and denote this fact as $a \sqcap b$ if and only if there is a non-empty fragment of the program which is nested both in the fragment corresponding to a and in the fragment corresponding to b.

Definition 3.4. We say the node a *strictly intersects* the node b, and denote this fact as $a \,!\sqcap\, b$ if and only if $a \sqcap b$ and none of the nodes from the given pair is nested in the other.

Definition 3.5. A framed fragment is called a *custom block* if and only if its opening and closing borders, with the exception of the prefixes specified by the options `start` and `end`, match, and the matching part is not empty, and the corresponding node f from the tree belonging to the forest of framed fragments satisfies one of the *embedding conditions*:

(*1*) $r \subseteq f$;

```
1|
2|  //+ Fragment 2
3|
4|  // Copyright (c) Alexey Goloveshkin
5|  // This code is distributed under
6|
7|  //+ Fragment 1
8|  namespace Test
9|  {
10|   public class Class1
11|   {
12|      public int Field1 = 1;
13|      public double Field2 = 2;
14|   }
15|  }
16|  //- Fragment 1
17|
18|  //- Fragment 2
```

(*a*) covering the entire program

```
1|  namespace Test
2|  {
3|    public class Class1
4|    {
5|      public int Field1 = 1;
6|
7|      //+ Fragment 2
8|      /// <summary>
9|      //// Method 1
10|     /// </summary>
11|     /// <param name="n">Int number</param>
12|     //+ Fragment 1
13|     public void Method1(int n)
14|     //- Fragment 1
15|     //- Fragment 2
16|     {
17|       if (n > 0)
18|       {
19|         System.Console.WriteLine("N > 0");
20|       }
21|     }
22|   }
23|  }
```

(*b*) covering the method header

FIGURE 7. Examples of custom blocks

(*2*) an AST node n exists, such that for any AST node n' such that n' is an ancestor of n or is n itself it holds that $f \mathbin{!\sqcap} n'$ or $f \subset n'$, and for any AST node n'' such that $n'' \subseteq f$ it holds that n'' is a descendant of n.

The first embedding condition allows us to add f to the AST as a new root of the tree. Figure 7a shows two custom blocks, the nodes of which can be embedded in the AST of the program as a root and its child. Note that the text fragment corresponding to the custom block can be significantly larger than the fragment corresponding to the AST root, since empty lines and comments are ignored by the parser, but can be within the custom block area. In the example above, the AST root corresponds to the text from line 8 to line 15 before embedding the custom blocks. The second condition describes an AST node n to whose children it is possible to add a node f that does not satisfy the first condition. The definition of custom block permits not only strict nesting of f in potential ancestors, but also strict intersection with them. Therefore, a marked fragment can include areas that are skipped by the parser, just as is allowed by the first embedding condition. Figure 7b shows two custom blocks going beyond the method `Method1`. Figure 5 demonstrates how such a custom block (`Valid block`) is embedded in an AST. Obviously, in the case shown, the text area that corresponds to the parent node `Method1` is expanded.

Checking the second condition and embedding the nodes that satisfy it is carried out by Algorithm 2, which is run recursively during the pre-order

traversal of the AST with the `Visit` method. In the algorithm, the input parameter n is a currently visited AST node. A list of roots of all trees from the forest of framed fragments is passed as the *fragments* parameter when visiting the AST root. The fulfilment of the condition 3.5.2 is checked incrementally while going down the AST. Simultaneously with going down the AST, the algorithm goes down the trees from the forest. When visiting some n node, roots of the forest subtrees whose nodes can potentially be embedded in an AST subtree rooted in n are passed in *fragments*. Stepping down some tree from the forest means the previously considered node of this tree is accepted as fully satisfying the condition 3.5.2 and is embedded in AST, or is discarded due to violation of the condition. In Algorithm 2, the function Children(a) is used to denote a sequence of children of node a, where a can be an AST node as well as a node from the forest of framed fragments.

At the first stage (lines 4–26 of Algorithm 2), for each node f from *fragments*, it is checked whether it can be embedded in the AST as a child of n. If f strictly intersects with some of the children of n or is nested in a child, such an embedding cannot be done. Otherwise, the fragment corresponding to f is treated as a custom block, and all children of n that are nested in f or textually match f are selected, forming some (possibly empty) subsequence of the sequence Children(n) (line 6 of Algorithm 2). Its elements become new children of f, and f itself is added instead of this subsequence to the children of n. If there are no children of n nested in f, f is childless and is inserted between the children of n such that the text of the preceding child precedes the text of f and the text of the following child follows the text of f. The InsertChild procedure used in the algorithm takes as its first argument the node to whose children, at the position specified by the third argument, the second argument must be added. After the loop, the old children of every node f inserted into the AST, which are the elements of the trees of framed fragments, are inserted into *fragments* (lines 27–31 of Algorithm 2) instead of the corresponding f.

At the second stage (lines 32–36 of Algorithm 2), framed fragments are selected that strictly intersect with exactly one child of n or are strictly nested in such a child (line 33 of Algorithm 2). They are possibly custom blocks. The list of the nodes corresponding to such fragments is passed as the second argument to the recursive call of the Visit method for that child. Finally, the only nodes remaining in *fragments* are those that definitely cannot be embedded in the AST. However, this does not mean that it is not possible to embed some of their descendants. All the nodes remaining in *fragments* are replaced with the sequences of their children (line 37 of Algorithm 2), and then the whole process is repeated.

Algorithm 2 Embedding nodes corresponding to custom blocks in an AST.

```
1: procedure Visit(n, fragments)
2:     while |fragments| > 0 do
3:         inserted ← ∅
4:         for all f ∈ fragments do
5:             if {c ∈ Children(n) | f !⊓ c ∨ f ⊂ c} = ∅ then
6:                 innerChildren ← [c ∈ Children(n) | c ⊆ f]
7:                 if |innerChildren| > 0 then
8:                     index ← IndexOf(innerChildren[1], Children(n))
9:                 else
10:                    index ← 0
11:                    for i ← 1, |Children(n)| do
12:                        if Children(n)[i].StartOffset > f.EndOffset then
13:                            index ← i
14:                            break
15:                        end if
16:                    end for
17:                    if index = 0 then
18:                        index ← |Children(n)| + 1
19:                    end if
20:                end if
21:                inserted ← inserted ∪ {f};   fCopy ← f
22:                SetChildren(fCopy, innerChildren)
23:                SetChildren(n, children \ innerChildren)
24:                InsertChild(n, fCopy, index)
25:            end if
26:        end for
27:        for all f ∈ inserted do
28:            index ← IndexOf(f, fragments)
29:            RemoveAt(fragments, index)
30:            InsertAt(fragments, Children(f), index)
31:        end for
32:        for all c ∈ Children(n) do
33:            innerFragments ←
    {f ∈ fragments | f !⊓ c ∧ ∀c′ ∈ Children(n) \ {c}, ¬(c′ ⊓ f) ∨ f ⊂ c}
34:            fragments ← fragments \ innerFragments
35:            Visit(c, innerFragments)
36:        end for
37:        fragments ← ⋃_{f∈fragments} Children(f)
38:    end while
39: end procedure
```

For an abstract syntax tree containing m nodes and a forest of framed fragments containing l nodes, the worst-case complexity of the algorithm is $\mathcal{O}(m \cdot l)$.

Obviously, Algorithm 2 adds any node f corresponding to condition 3.5.2 to the AST as a child of some node n. When visiting the ancestors of n, the node f is passed (directly or as a part of a subtree whose root is in *fragments*) to recursive calls of the Visit method. When n is visited, the node f itself, or one of its ancestors from the framed fragment tree, is embedded in the AST either as enclosing one or more children of n, or as not intersecting with any of them. If not f itself is embedded at the current step, but its parent is, as the AST traversal continues with regard to the changes caused by that embedding, f will be embedded when the node corresponding to its parent is visited recursively.

In Figure 5a, the framed fragment `Incorrect block` is not a custom block, since the node corresponding to it is nested in all AST nodes in the path from the root to `Method1`, but at the level of the children of the node `Method1` it includes the header node and at the same time strictly intersects with the method body. As a result, `Incorrect block` is not represented in the resulting AST shown in Figure 5*d*.

The markup tool automatically checks the embedding conditions if the programmer wants to bind to some multi-line fragment. It is not possible to create the framed fragment `Invalid block` shown in Figure 5*a* using the markup tool. When trying to bind to the corresponding code area, only binding to the enclosing class `Class1` is offered.

3.2.3. *Correctness of custom block definition*

The custom block concept is introduced in such a way that when embedding a custom block into an AST, the correctness of an AST is preserved. By such correctness, we mean consistency with the fundamental properties of an AST. For any nodes n_1 and n_2 that belong to the AST and are distinct from each other,

(*1*) n_1 is an ancestor of $n_2 \implies n_2 \subseteq n_1$;

(*2*) $n_1 \sqcap n_2 \iff n_1$ is an ancestor of n_2 or n_2 is an ancestor of n_1.

Proposition 3.1. Let the node f corresponding to the custom block be embedded in the AST. Then for any nodes n_1 and n_2 belonging to this AST and being distinct from each other and from f,

(1) n_1 is an ancestor of n_2 if and only if n_1 was an ancestor of n_2 before embedding f.

(2) n_1 and n_2 are connected by the relation $\subseteq$ if and only if they were connected by this relation before embedding f.

(3) n_1 and n_2 are connected by the relation $\sqcap$ if and only if they were connected by this relation before embedding f.

PROOF. Embedding nodes that match condition 3.5.1 does not affect the edges within the initially built AST and does not change the text areas corresponding to the nodes that were present in the initially built AST. For nodes embedded in the AST in accordance with condition 3.5.2, the validity of all parts of the proposition follows from Algorithm 2. □

Note that the nodes n_1 and n_2, connected by the relation $\subset$ before embedding f in the AST, can be connected by the relation $\overset{\mathrm{T}}{=}$ after embedding, if f became their descendant. As noted earlier, code areas corresponding to AST nodes that are the ancestors of f can be expanded as a result of embedding f, as they must cover all descendants.

Proposition 3.2. Let the node f corresponding to the custom block be embedded in the AST. Then for any node n' belonging to this AST and being distinct from f,

(1) n' is an ancestor of $f \implies f \subseteq n'$;

(2) f is an ancestor of $n' \implies n' \subseteq f$;

(3) $n' \sqcap f \iff n'$ is an ancestor of f or f is an ancestor of n'.

PROOF. Node f, which is embedded due to condition 3.5.1, has no ancestors, and all nodes of the initial AST are nested in f and become its descendants. Therefore, all parts of the statement are satisfied. For nodes embedded in the AST in accordance with condition 3.5.2, the validity of all parts of the statement follows from Algorithm 2. □

Statement 3.1 means that embedding the custom block node in the AST does not violate the relations between the nodes of the AST that existed before embedding. Statement 3.2 means that f itself correctly participates in relations with the initial AST nodes.

Table 1. Projects analyzed.

	Initial version date	Current version date	Commits	Files changed
asp	03.02.2020	04.07.2020	3551	1398
pabc	25.05.2018	26.05.2020	1462	335
ros	03.02.2021	30.04.2021	3459	1749

4. Experiments

Since binding to multi-line code fragments is carried out in the same way as binding to syntactic entities, the results of the experiments given in [9] can be considered to describe the robustness of custom blocks rebinding. For classes, methods, fields and properties, it is demonstrated that marked and later edited elements are successfully found in 98,93–100% of cases.

To check the robustness of binding to single-line fragments, we perform binding to randomly selected lines in the codebases of three large open source projects: *ASP.NET Core* (*asp*), *PascalABC.NET* (*pabc*), *Roslyn* (*ros*). Table 1 provides more detailed information for each of them. The main language of all projects is C#.

For each project, some version of the codebase is chosen as the initial one. Then we move a certain number of commits ahead to some version that is considered the current one. In the initial version of each project, a certain number of methods (300 for *asp*, 100 for *pabc*, and 300 for *ros*) is randomly selected that are edited in the current version and successfully rebound by our markup tool. It is additionally checked that there are two or more lines in the body of the method. Then binding is performed to two randomly selected lines in each method. The expected result is a successful rebinding of all lines that still exist in the current version of the method. The prerequisite consisting in the successful rebinding of the methods themselves makes it possible to eliminate the impact of the previously tested algorithm, which rebinds large syntactic entities, on the results of the experiment.

Table 2 shows the results of the experiment. For each project, the marked lines are divided into two groups. “Good” ones are all lines except empty lines and lines with a single operator bracket (`{` or `}`). “Bad” lines are ones with a single operator bracket or empty. The first subcolumn for each project contains statistics for “good” lines, the second one is for

Table 2. Results of the experiment.

	asp		*pabc*		*ros*	
Total	433	167	155	45	435	165
Rebound, right	410	151	147	32	414	144
Rebound, wrong	2	4	2	11	8	12
Place found, right	18	9	4	1	10	6
Place found, wrong	3	3	2	1	3	3
Success, %	99,51	97,42	98,66	74,42	98,1	92,31

"bad" lines. Empty lines and lines with operator brackets are isolated into a separate group, since they are almost always not unique within a method, so their rebinding can be significantly less successful than for other lines (at the same time, the probability that in a real situation the programmer decides to bind to them is also lower).

The "Total" row of the table shows the total number of marked lines, "Rebound, correct" shows the number of cases when the previously marked line is still present in the code and is correctly rebound, the row "Rebound, incorrect" indicates the number of lines for which relevant versions are still present in the code, but rebinding to incorrect candidates occurs. The line "Location found, correct" indicates the number of cases when the searched line is deleted and rebinding is performed to a place no more than 1–2 lines apart from the former location of the deleted line. In "Location found, incorrect", cases are considered when the searched line is deleted and rebinding occures to a line located more than 1–2 lines apart from the initial place. In the "Success" row, the percentage of successfully rebound lines from the total number of marked lines that are not deleted is calculated.

A significant difference between the results of rebinding "good" and "bad" lines is observed, especially for *pabc*, where the success rate of rebinding the "bad" lines is 74,42%, while for the rest of the lines, the success rate is 98,66%. Detailed analysis shows that the outer context plays a key role in successful rebinding for "bad" lines, so "bad" lines located close to each other cause misrebinding because their surroundings are almost identical. For "bad" lines, it also turns out to be significant that the outer context similarity score is decreased when editing the method signature. Because of the lightweightness of the parsing, for the line located in the method body, the outer context captures not just the content of the body, but the entire

```
var symbol = semanticModel.GetDeclaredSymbol(ancestor, cancellationToken);
if (symbol != null)
{
    if (symbol.Locations.Contains(location))
    {
        return symbol;
    }
    // We found some symbol, but it defined something else. We're not going to have a higher node defining _another_ …
    return null;
}
```

(*a*) original version of the code

```
var symbol = semanticModel.GetDeclaredSymbol(ancestor, cancellationToken);
if (symbol != null)
{
    // The token may be part of a larger name (for example, `int` in `public static operator int[](Goo g);`.
    // So check if the symbol's location encompasses the span of the token we're asking about.
    if (symbol.Locations.Any(loc => loc.SourceTree == location.SourceTree && loc.SourceSpan.Contains(location.SourceSpan)))
        return symbol;
    // We found some symbol, but it defined something else. We're not going to have a higher node defining _another_ …
    return null;
}
```

(*b*) current version of the code

Figure 8. Example of an incorrect rebinding of a line in *ros*

text of the method except for the line itself. More precise handling of such situations would increase the percentage of successful rebindings.

As for "good" (informative) lines, their rebinding fails primarily due to intensive editing of the lines themselves. Figures 8 and 9 show two out of eight errors occurred for the "good" lines in *ros*. In a) and b) in each figure, the initially marked line and the line to which the rebinding is performed, respectively, are highlighted in gray. A line taken into a frame with a transparent background is the one rebinding to which would be correct.

It is worth noting that sometimes deleting a line from a method does not mean that the line completely disappears. In every project, cases exist when some part of the method is extracted into a separate method, or a number of arguments is added to the signature and an overloaded version of the method matching the old signature appears. At the level of syntactic entities, in this case, rebinding is performed to a method with the signature matching the initial one. However, this method may not contain some lines that are expected to be there, as they were taken out or even never were there. These cases are counted in the "Location found" row. The correct place for rebinding is considered to be the call to the method in which the searched line is actually located. Obviously, rebinding can be more accurate in such situations if the absence of a good candidate line affects the rebinding of the enclosing entity.

```
using (var token = _asyncListener.BeginAsyncOperation(nameof(FindImplementingMembersAsync)))
{
    // Let the presented know we're starting a search.
    var context = presenter.StartSearch(
        EditorFeaturesResources.Navigating, supportsReferences: true);
    using (Logger.LogBlock(
        FunctionId.CommandHandler_FindAllReference,
        KeyValueLogMessage.Create(LogType.UserAction, m => m["type"] = "streaming"),
        context.CancellationToken))
    {
```

(*a*) original version of the code

```
using var token = _asyncListener.BeginAsyncOperation(nameof(FindImplementingMembersAsync));

// Let the presented know we're starting a search.  We pass in no cancellation token here as this
// operation itself is fire-and-forget and the user won't cancel the operation through us (though
// the window itself can cancel the operation if it is taken over for another find operation.
var (context, cancellationToken) = presenter.StartSearch(EditorFeaturesResources.Navigating, supportsReferences: true);
using (Logger.LogBlock(
    FunctionId.CommandHandler_FindAllReference,
    KeyValueLogMessage.Create(LogType.UserAction, m => m["type"] = "streaming"),
    cancellationToken))
{
```

(*b*) current version of the code

FIGURE 9. Example of an incorrect rebinding of a line in *ros*

Conclusion

When working on a project, programmers need a tool that allows them to mark code sections that are important for the current task, use previously created markup to quickly resume work on some other task, and share this information with their colleagues. The key problems when creating such a tool are binding to code — memorizing some place in it — and rebinding — finding a previously memorized place in the current version of the code. Robustness to editing is the crucial property of binding to code. It means that rebinding should be successful no matter how much the fragment of interest is edited.

In the paper, we propose the continuation of our research on binding to code based on binding to nodes of a lightweight abstract syntax tree of a program.

Previously created models and algorithms, which are designed for robust binding to large syntactic entities of a program, are extended to provide robust binding to arbitrary single-line fragment nested in some syntactic entity. For binding to a multi-line fragment, the necessary formalization and the algorithm that checks the correctness of the multi-line fragment selection with regard to the syntactic structure of the program, and then embeds the node, corresponding to this fragment, in the AST, are proposed. The experiments demonstrate that rebindings of previously marked arbitrary code lines are successful in the vast majority of the cases.

For future work, we consider improving the robustness of binding to single-line fragments by more complex examination of the contexts describing the line, and tighter integration of the markup tool with IDEs and taking into account the current code editing scenarios when rebinding.

Authors contribution: Alexey V. Goloveshkin — 85% (software, validation, formal analysis, investigation, resources, data curation, writing (review & editing), visualization, supervision); Stanislav S. Mikhalkovich — 15% (methodology, visualization, project administration, funding acquisition).

References

[1] M. Schröer, R. Koschke. "Recording, visualising and understanding developer programming behaviour", 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (11 May 2021, Honolulu, HI, USA), 2021, pp. 561–566. ↑36

[2] A. Ciborowska, A. Chakarov, R. Pandita. "Contemporary COBOL: developers' perspectives on defects and defect location", 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (27 Sept.–1 Oct. 2021, Luxembourg), 2021, pp. 227–238. ↑36

[3] C. Parnin, S. Rugaber. "Resumption strategies for interrupted programming tasks", *Software Quality Journal*, **19**:1 (2011), pp. 5–34. ↑36

[4] O. Badreddin, R. Khandoker, A. Forward, O. Masmali, T. C. Lethbridge. "A decade of software design and modeling: a survey to uncover trends of the practice", *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS'18 (14–19 October 2018, Copenhagen, Denmark), ACM, New York, NY, 2018, ISBN 978-1-4503-4949-9, pp. 245–255. ↑36

[5] M.-A. Storey, J. Ryall, J. Singer, D. Myers, L.-T. Cheng, M. Muller. "How software developers use tagging to support reminding and refinding", *IEEE Trans. Softw. Eng.*, **35**:4 (2009), pp. 470–483. ↑36

[6] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, A. van Deursen. "Collective code bookmarks for program comprehension", 2011 IEEE 19th International Conference on Program Comprehension (22–24 June 2011, Kingston, ON, Canada), 2011, pp. 101–110. ↑36

[7] A. V. Goloveshkin, S. S. Mikhalkovich. "Tolerant parsing with a special kind of "Any" symbol: the algorithm and practical application", *Proc. ISP RAS*, **30**:4 (2018), pp. 7–28. ↑36, 40

[8] A. V. Goloveshkin. "Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded "Any" symbol", *Proc. ISP RAS*, **31**:3 (2019), pp. 7–28. ↑36, 40

[9] A. Goloveshkin, S. Mikhalkovich. “Using improved context-based code description for robust algorithmic binding to changing code”, *Procedia Computer Science*, **193** (2021), pp. 239–249. ↑36, 38, 41, 45, 55

[10] A. V. Goloveshkin, S. S. Mikhalkovich. “Binding to an arbitrary section of the program in the task of markup of program code”, *Sovremennyye informatsionnyye tekhnologii: tendentsii i perspektivy razvitiya*, Materialy XXVI nauchnoy konferentsii (18-–19 aprelya 2019 g., Yuzhnyy federal’nyy universitet, Rostov-na-Donu, Rossiya), Yuzhnyy federal’nyy universitet, Rostov-na-Donu–Taganrog, 2019, ISBN 978-5-9275-3139-4, pp. 86–89 (In Russian). ↑37, 46

[11] A. L. Fuksman. *Technological Aspects of Software System Design*, Statistika, M., 1979 (In Russian). ↑38

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. “An overview of AspectJ”, *ECOOP 2001 — Object-Oriented Programming*, 15th European Conference (18–22 June 2001, Budapest, Hungary), LNCS, vol. **2072**, Springer-Verlag, Berlin, 2001, ISBN 978-3-540-45337-6, pp. 327–354. ↑38

[13] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella. “Delta-oriented programming of software product lines”, *Software Product Lines: Going Beyond*, 14th International Conference SPLC 2010 (13–17 September 2010, Jeju Island, South Korea), LNCS, vol. **6287**, Springer-Verlag, Berlin, 2010, ISBN 978-3-642-15578-9, pp. 77–91. ↑38

[14] S. Apel, C. Kästner, C. Lengauer. “Language-independent and automated software composition: the FeatureHouse experience”, *IEEE Trans. Softw. Eng.*, **39**:1 (2013), pp. 63–79. ↑38

[15] S. Kanai. *Customize code generation using IBM Rational Rhapsody for C++*, IBM Support. ↑38

[16] S. Yu. Kalinin, I. A. Kolokolov, A. N. Litvinenko. “Applying AOP concepts to the development of extendable applications”, *Izvestiya Yuzhnogo federal’nogo universiteta. Tekhnicheskiye nauki*, **103**:2 (2010), pp. 58–68 (In Russian). ↑38

[17] M. S. Malevannyy, S. S. Mikhalkovich. “Programming environment support for navigating through aspects of program code”, *Sovremennyye informatsionnyye tekhnologii: tendentsii i perspektivy razvitiya*, Materialy konferentsii (17–18 aprelya 2014 g., Yuzhnyy federal’nyy universitet, Rostov-na-Donu, Rossiya), Yuzhnyy federal’nyy universitet, Rostov-na-Donu, 2014, ISBN 978-5-9275-1227-0, pp. 275–276 (In Russian). ↑

[18] M. S. Malevannyy, S. S. Mikhalkovich. “Context-based model for concern markup of a source code”, *Proc. ISP RAS*, **28**:2 (2016), pp. 63–78. ↑38

[19] M. S. Malevannyy, S. S. Mikhalkovich. “Robust binding to syntactic elements in a changing code”, *Proceedings of the 12th Central and Eastern European Software Engineering Conference in Russia*, CEE-SECR’16 (28–29

October 2016, Moscow, Russia), ACM, New York, NY, USA, 2016, ISBN 978-1-4503-4884-3, 8 pp. ↑38

[20] C. K. Roy, J. R. Cordy, R. Koschke. "Comparison and evaluation of code clone detection techniques and tools: a qualitative approach", *Science of Computer Programming*, **74**:7 (2009), pp. 470–495. ↑39

[21] D. Rattan, R. Bhatia, M. Singh. "Software clone detection: a systematic review", *Information and Software Technology*, **55**:7 (2013), pp. 1165–1199. ↑39

[22] A. Walker, T. Cerny, E. Song. "Open-source tools and benchmarks for code-clone detection: past, present, and future trends", *SIGAPP Appl. Comput. Rev.*, **19**:4 (2019), pp. 28–39. ↑39

[23] M. Novak, M. Joy, D. Kermek. "Source-code similarity detection and detection tools used in academia: a systematic review", *ACM Trans. Comput. Educ.*, **19**:3 (2019), 27, 37 pp. ↑39

[24] H. Cheers, Y. Lin, S. P. Smith. "Evaluating the robustness of source code plagiarism detection tools to pervasive plagiarism-hiding modifications", *Empirical Software Engineering*, **26** (2021), 83, 62 pp. ↑39

[25] S. Luan, D. Yang, C. Barnaby, K. Sen, S. Chandra. "Aroma: code recommendation via structural code search", *Proc. ACM Program. Lang.*, **3**:OOPSLA (2019), 152, 28 pp. ↑39

[26] G. Dotzler, M. Philippsen. "Move-optimized source code tree differencing", *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE'16 (3–7 September 2016, Singapore), ACM, New York, NY, USA, 2016, ISBN 978-1-4503-3845-5, pp. 660–671. ↑39

[27] V. Frick, T. Grassauer, F. Beck, M. Pinzger. "Generating accurate and compact edit scripts using tree differencing", 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) (23–29 Sept. 2018, Madrid, Spain), pp. 264–274. ↑39

[28] M. Kim, D. Notkin, D. Grossman, G. Wilson. "Identifying and summarizing systematic code changes via rule inference", *IEEE Transactions on Software Engineering*, **39**:1 (2013), pp. 45–62. ↑39

[29] C. Ragkhitwetsagul, J. Krinke, D. Clark. "A comparison of code similarity analysers", *Empirical Software Engineering*, **23**:4 (2018), pp. 2464–2519. ↑39

[30] S. Brody, U. Alon, E. Yahav. "A structural model for contextual code changes", *Proc. ACM Program. Lang.*, **4**:OOPSLA (2020), 215, 28 pp. ↑39

[31] M. Asaduzzaman, C. K. Roy, K. A. Schneider, M. D. Penta. "LHDiff: a language-independent hybrid approach for tracking source code lines", *2013 IEEE International Conference on Software Maintenance* (22–28 Sept. 2013, Eindhoven, Netherlands), 2013, ISBN 978-0-7695-4981-1, pp. 230–239. ↑39

[32] L. Moonen. "Generating robust parsers using island grammars", *Proceedings of the 8th Working Conference on Reverse Engineering* (2–5 Oct. 2001,

Stuttgart, Germany), IEEE Computer Society, Washington, DC, USA, 2001, ISBN 0-7695-1303-4, pp. 13–22. ↑40

[33] M. G. J. van den Brand, M. P. A. Sellink, C. Verhoef. "Obtaining a COBOL grammar from legacy code for reengineering purposes", *Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications* (25–26 September 1997, Amsterdam, The Netherlands), eWiC, BCS Learning & Development Ltd., Swindon, UK, 1997, ISBN 3-540-76228-0, pp. 1–16. ↑40

[34] J. Oliver, C. Cheng, Y. Chen. "TLSH – a locality sensitive hash", *Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop*, CTC'13 (21–22 Nov. 2013, Sydney, NSW, Australia), IEEE Computer Society, Washington, DC, USA, 2013, pp. 7–13. ↑42

[35] V. I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals", *Dokl. Akad. Nauk SSSR*, **163**:4 (1965), pp. 845–848 (In Russian). ↑45

Received 28.12.2021;
approved after reviewing 08.02.2022;
accepted for publication 19.02.2022.

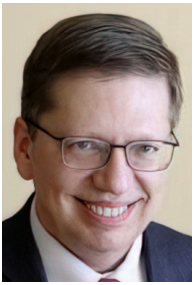
Recommended by *And. V. Klimov*

Information about the authors:

Alexey V. Goloveshkin

MSc in Computer Science and Information Technologies. Research interests: programming languages, analysis of software evolution, software engineering. Area of interest: programming languages, software evolution analysis, software engineering.

0000-0001-6947-0594
e-mail: alexeyvale@gmail.com



Stanislav S. Mikhalkovich

PhD, Associate Professor, Head of the Department of Informatics and Computational Experiment at I.I. Vorovich Institute for Mathematics, Mechanics, and Computer Science of Southern Federal University, head of the PascalABC.NET project.

0000-0003-0373-3886
e-mail: miks@sfedu.ru