STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LX**, Number 1, 2015

CREATING AN EFFICIENT AND INCREMENTAL IDE FOR TTCN-3

KRISTÓF SZABADOS

Abstract. In this article we present methods and algorithms for constructing an efficient IDE in the sense that the processing costs of reanalyzing source code after change is minimal. Moreover, we show that these methods and algorithms can be designed in a way that they support iterative realization, hence, they fit better to the current trends of iterative software development life-cycle. We also show how these algorithms can be built into an existing system and we show measurements on performance benefits. The proposed methods were validated in the telecommunication area for compiling TTCN-3 code.

1. Introduction

Nowadays developing a modern Integrated Development Environment (IDE) has two major requirements: (1) it has to be responsive and (2) developed in a lean iterative process.
(1) The common way of checking source code for errors is to have a compiler parse all of the files and run a semantic analysis on the produced data. In case of modern IDEs this is not feasible, full analysis takes too long on large projects. An IDE is expected to analyze the project automatically and to report possible errors instantaneously, irrespective of the size.
(2) The current software development trends favor iterative development in order to reduce cost. Instead of large development efforts supporting complex situations, the aim is to fullfill the needs of the user with a lean solution.

This paper is organized as follows. In Section 2 we present earlier works related to this subject. In Section 3 we introduce the datamodel and its

Received by the editors: April 25, 2014.

2010 *Mathematics Subject Classification.* 68N20, 68W40.

1998 *CR Categories and Descriptors.* D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *Program editors*; D.2.6 [**Software Engineering**]: Programming Environments – *Integrated environments*; D.3.4 [**Programming Languages**]: Processors – *Incremental compilers*.

Key words and phrases. Parsing, Integrated Development Environment, Iterative Development.

operations. Section 4 describes the measurement environment. Section 5 presents the incremental parsing algorithm. Section 6 shows an outlook on how these methods can be used to speed up the semantic checking. Finally, Section 7 gives an analysis of the algorithm's performance, and Section 8 summarizes the results of this paper.

2. Related work

Several articles have already been published on incremental parsing. In programming languages recognition, languages are typically described by LL(k)[1], LL(*)[2], LR(0)[3], LR(1), and LALR(1)[4] grammars. Research on their incremental parsing focused mostly on LL [12, 7, 8] and LR parsing [3, 11, 5, 6, 10] methods. All techniques parse the input text, building the first parse tree, and update it according to the changes in the input. An important point in each case is finding the minimal structural units that are affected by the modifications ([3] for LR(0) and [6] for LR(k) grammars). Incremental parsers use "re-parse points" where parts of the semantic information could be re-parsed from the input. These occur close to the update location [7]. Others proposed approaches for doing incremental parsing with repairing errors coming from earlier processing as they get resolved later [1]. Wagner et al. [11] describes a method which does not use error recovery, but utilizes the history of modifications instead. There are also approaches applying neural networks (see [2] e.g. uses the Perceptron algorithm). Huang et al. [4] suggested dynamic programming to enhance incremental processing.

Most of these articles propose parsed syntax trees to enhance the re-parsing speed. Their aim is to minimize the parsing cost starting from any point in the tree. To reach this the authors produce parser generators which create incremental parsers from the Backus-Naur Form (BNF) of their target language. In this setup the tree is usually uniform and completely represents the given text in the file. The algorithms recommended by the authors work on the same way for each node, can modify any node in the tree and can generate full results in a single execution.

We propose a different approach. Instead of using a Parse Tree that represents an identical copy of the text in a tree shaped form, we use an Abstract Semantic Tree (AST). In an AST the semantic processing can change the

[1]LL is a top-down parser class for a subset of the context-free grammars. An LL parser is called an LL(k) parser if it uses k tokens of lookahead

[2]An LL parser is LL(*) if it uses the minimum lookahead per input sequence [9].

[3]LR is a bottom-up parser class, reading the input in one direction (typically left to right), producing a reversed rightmost derivation

[4]LALR parsers are simplified canonical LR parsers, with reduced language recognition power, and significantly reduced memory requirements

form and order of the information, for example order them alphanumerically, transform for efficiency or extend with external information if required. The iterative development approach allows us to implement a tool that generates the solution stepwise, i.e., it can be developed incrementally until the cost of further developments overweight its benefits.

We analyzed how the incremental parsing can be used to speed up semantic checking. A process missing from the articles mentioned earlier.

3. The model

In this section we introduce the model we used to represent data and modifications.

The Titan IDE, where these algorithms are implemented, supports Testing and Test Control Notation - 3 (TTCN-3)[5] and Abstract Syntax Notation One (ASN.1) files [13].

3.1. **Data representation.**

Our representation has one node for every module (TTCN-3 compilation unit) serving as the local root node of the module. These local root nodes form a list of nodes, representing the complete set of semantic information. Modules have subnodes for each top level definition, and definitions can also have subnodes in a recursive manner. This continues till the leaf nodes, which represent some semantic terminals.

The AST stores location information in attributes. The region of each node is textually enclosing the regions of all its subnodes. This means that for any point in the module, there is a node in the AST, whose textual location contains that point and has the shortest region of such nodes.

To provide performance benefits the semantic analysis can reorder the elements of the AST, while keeping their structural information. Where ordering is not defined by the language, the items are stored in associative containers.

3.2. **The model of modifications.**

Our assumption is that a user can change the text within a file at any location. Indirectly applying the following modifications to the AST:

- Creating and inserting new subtrees into the AST.
- Deleting subtrees of the AST.
- Merge or divide nodes in the AST[6].
- Editing text with no semantic value, does not change the semantic state, but locations might need to be adapted.

[5]TTCN-3 [14] is an imperative programming language with testing related extensions and with syntax and semantics close to imperative languages like C.

[6]Insertion of terminals can separate a semantic node into several new nodes. Deleting the closing and opening sings can merge nodes into a new node.

Changes might also introduce syntactical errors that corrupt parts or the whole AST. It is also possible that previous syntactical errors were corrected with a modification of the AST.

We assumed that most of the time users are editing consecutively, working in logical units. This means that most of the time the same or related nodes are changed in the AST.

3.3. **Re-parse points.** To change the AST in a consistent manner, to handle parsing of the minimal amount of text, and to contain the negative effects of syntax errors, special nodes have to be located in the AST. A re-parse point is a node in the AST, whose textual location can be re-parsed as a consistent entity, based on the information known about the AST and the modification as a precondition. These AST nodes form a subtree in the BNF of the language. When the same text representing these nodes is parsed again, the AST has exactly the same semantic meaning.

4. The measurement environment

In this section we introduce our measurement environment, and our measurements of the full file parsing method.

For performance measurments we used 8 projects (Table 1) of different sizes and 2 different execution modes:

(1) Client mode simulates the program starting up in an unoptimized environment. Java VM[7] is started in "client" mode. We measured the first executions of the algorithms.
(2) Server mode simulates an optimized environment. Java VM is started with the "-server"[15] flag[8]. The algorithms were run 5 – 10 times using exactly the same operations before measuring.

To see if the projects and execution modes will give us measurable differences we measured the first syntax checking, which has to read all text in the project. We measured (Figure 1) a clear correlation between the execution time and the amount of text to be processed.

[7]Virtual Machine

[8]uses the most aggressive performance optimizations

Table 1. Projects analyzed

project index	1	2	3	4	5	6	7	8
number of modules	4	39	65	68	118	204	567	828
thousand lines of code	28	19	52	66	66	436	1.174	826

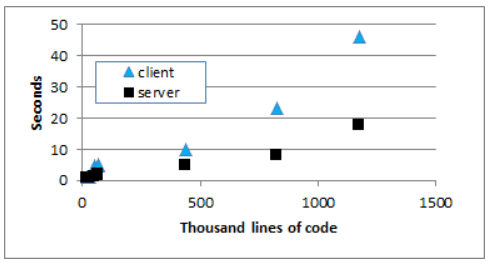

FIGURE 1. Full syntax checking performance by lines

5. THE INCREMENTAL PARSING ALGORITHM

In this section we present the methods of the incremental processing algorithm and provide performance measurements.

5.1. **Collecting changes.** In Procedure 1 the processing of syntactic changes and semantic checking is separated. The consecutive changes are merged (line 2) first to reduce the number of longer running checks. The elements of the merged list are processed by invoking a syntax check on them. When all changes are syntactically processed, a single semantic check (line 5) is executed.

Our tool collects changes and does calculations as a background thread to not burden the user interface with heavy processes. Our prototype tool also waits 1 second after the user has finished working on the text to start the background processing. This was seen to be efficient in practice. While the user enters or deletes a consecutive list of characters, the background processing does not start.

5.2. **Refreshing the AST.** The generic form of finding the affected nodes is a recursive algorithm (Procedure 2), invoked with the node to be processed, and the change that happened to the file.

The algorithm iterates on the child nodes of its actual parameter node. The locations of children following the change are updated (line 6). When a change is located inside the child, the algorithm is invoked on it recursively (line 8). If that fails to contain the change, the algorithm reparses the child node itself (line 11). Should that fail, or the change be outside the child nodes, the algorithm backtracks to the previous level of recursion by returning failure.

5.3. **Processing an affected node.** We created specialized versions of the generic algorithm for processing certain types of nodes efficiently. To see how these algorithms interact, see the example in appendix B.

5.3.1. *Processing the file/module root.* The module root[9] has no parent node to regress to, if it could not handle the change the whole file has to be re-parsed (Procedure 3).

5.3.2. *Processing semantic lists.* Semantic lists[10] are elements of the same root type and follow each other in the text. In these structures ordering is not assumed, any number of elements can appear or disappear after a change, and the semantic analyzer can reorder these elements.

The algorithm (Procedure 4) uses two variables, left and right (boundary), to store the locations of safety boundaries. The invariant of this region is that nodes outside of it are not affected by the change and the procedure minimizes this region. It also stores whether the change was already enveloped in a child node: the variable enveloped is initialized to false.

This algorithm differs from the generic in that, first the smallest interval to be parsed is measured by checking the nodes against the changed region (lines 5 – 13). Nodes falling in this region are removed (lines 14 – 18) and the region is re-parsed (line 18) integrating any new node found. If there were no errors, the locations of the nodes following the change location are updated.

5.3.3. *Processing non-list semantic structures with elements.* In semantic structures containing semantically different sub-nodes[11], the ordering and existence of its sub-nodes is fixed[12] and they can not appear or disappear without causing syntactic errors[13]. The children of such nodes are processed in the order of their appearance in the textual representation of the node (Procedure 2).

5.3.4. *Processing semantic structures with no elements.* In terminating nodes, without semantic branches either the whole node has to be re-parsed or it can not be re-parsed, in those cases negative result has to be reported.

---

[9]In our implementation this is the TTCN3Module class, which represents the whole of a TTCN-3 module.

[10]In our implemetation the Definitions class represents the list of all definitions on the module level (types, templates, constants, module parameters, functions, altsteps, testcases). The class StatementBlock represents the consecutive list of statements inside functions, altsteps, testcases, statement blocks and the control part.

[11]In our implementation any semantic node that has children, but does not fit in the first 2 categories. For example: the Def_Function class representing a function definition.

[12]Some of the sub-nodes are required to create a syntactically and semantically correct node, while some can be optional. As an example a function might have a name, formal parameter list, return clause, body (a statement block).

[13]In a function definition if any of the name, formalparameter list (as the whole entity), the *runs on* and *system* clauses, or the statement block(as the whole entity) are damaged, the whole function definition is incomplete.

6. Effect on the semantic checking

The incremental parsing algorithms allows improvements on the performance of the semantic checking as well. Decreasing the amount of changes on AST makes caching of previously calculated results possible. The introduction of version handling for the semantic nodes can provide both safety and performance gain. This can be done by assigning a time-stamp, holding the time of the last semantic analysis to semantic nodes. After the first parsing, all nodes are time-stamp uninitialized, and nodes are checked semantically. Following this check, the nodes can have the time-stamp of the actual semantic check cycle.

A node that was already semantically checked needs to be re-checked only when the semantic properties of the node, it's contents or a referenced node has been changed. As the dependency hierarchy of the modules is known from the previous semantic evaluation and does not change, it can be used to find which modules reference changed modules directly or indirectly.

7. Performance Analysis

7.1. **Searching for the correct node.** Our AST is a tree, where non-leaf nodes can have different number of child nodes. The root of this tree is the semantic node representing the module. The depth of this tree can be estimated with $O(\log_M n)$, where M denotes the branching factor (the children an internal node has in average) and n denotes the number of nodes. This AST has a high branching factor, making the tree's height very small. In modern programming methodology an embedding of more than 7 levels is usually considered a serious design fault, in practice the depth of the tree is usually less than 10.

When the algorithm explores the AST from the root, to the location that closest approximates the damaged region, it is traversing this B-tree, which can be estimated to take $O(\log_M n)$ steps. In the best case, this is the final location, and it is able to process the changes there. This means the processing of $\frac{S}{M^k}$ characters, where S denotes the size of the whole file and k denotes the number of levels descended.

Sometimes it is not able to handle the change on the lowest level, hence the amount of work done by the tool can be estimated with

$$\sum_{i=\log_M n}^{f} \frac{S}{M^i}$$

where f denotes the level where the change could be handled.

In the worst case, when the whole module becomes corrupted by the changes, this means that slightly more work is done as a normal parse does. In this case it had to parse not only the whole file but also had to parse smaller parts of it in order to decide that they are not able to contain the damage done. In the best case only the lowest level of the tree has to be parsed. As the amount of characters to re-parse decreases significantly on each level of the tree, the effective work becomes very small. In the average case, the algorithm finds the node that may contain the damage somewhere in between the corner cases. In some cases the whole file does not need to be re-parsed, which means that most probably it doesn't have to parse more than S/M characters. In such a case the execution time of the program will decrease by some power of M.

Since users usually transform syntactically correct text into an other syntactically correct text by very small changes, a valid assumption could be that the average execution time is close to the one estimated for the best case.

7.2. **Memory usage.** The described algorithms use only local variables needed to keep information like numbers for the left and right boundary, data that is already available in the AST, or can be calculated.

When the re-parsing of the damaged region is done, a part of the AST is rebuilt. Once this new AST part is inserted into the semantic database, the old version is removed.

7.3. **Implementation.** The implementation can be done iteratively. Traversing the tree only to the definition level in the AST already provides a speedup. In this case, when the algorithm finds that the damaged area is completely enveloped by a definition, it re-parses the whole definition. Limiting the amount of text to be processed from the whole file to a single definition decreases the original execution time to $O(s/m)$.

Once the algorithm reaches the level of statements and references, the amount of work has already been decreased to a minimal level. In practice, when general programming design style guides are followed, this should be no more than a single line of text. Decreasing the amount of work to below one line of characters, might not provide any significant benefits[14].

Implementation is not limited to procedural methods, our implementation in *Eclipse Titan* is done using Object-Orientet constructs.

[14]For example the list of formal parameters in the FormalParameterList class could also be treated like a list, but as there are usually only a few formal parameters in a list we choose to not make them incremental yet.

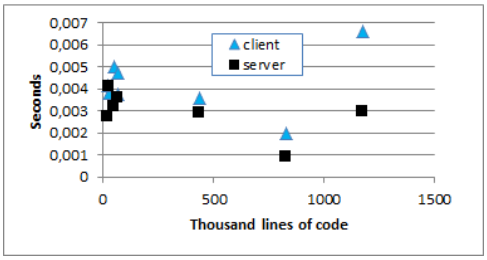

FIGURE 2. Incremental syntax checking performance by lines

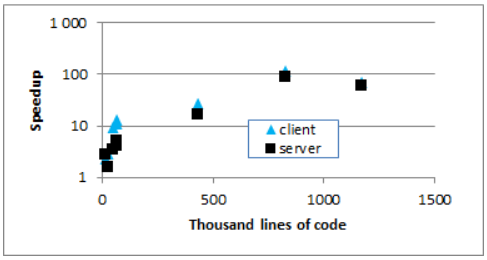

FIGURE 3. Incremental syntax check speedup

7.4. **Measurements.** To trigger the incremental syntax checking, we entered a single space in a randomly choosen module, without changing the semantics. In this setup: (1) the location to be parsed is determined, (2) the incremental parser is invoked, (3) some text is processed and the location informations are adjusted. The amount of characters to be parsed is minimal, minimizing the text processing overhead of the system.

In all cases the execution time of the incremental syntax analysis stays below $7 \cdot 10^{-3}$ seconds in client, and $3 \cdot 10^{-3}$ seconds in server mode (figure 2). In some cases the execution time goes as low as $9,23 \cdot 10^{-4}$ seconds in server mode.

Figure 3 displays the speedup measured showing how much this method improves the performance.

8. CONCLUSION

In this article we presented an algorithm that makes incremental parsing of TTCN-3 files possible, reparsing needs only as much text as necessary to analyze the change. We also demonstrated how this algorithm can be integrated into an existing system. We showed an approach to enhance an existing semantic checking system in order to take better use of incremental parsing.

The measurements shows that the algorithm yields reduced parsing times. With the original method, small modifications triggered the analysis of the whole file, lasting up to seconds. With the proposed algorithm the execution of exactly the same test was hardly noticeable for human users.

9. Further work

It is desirable to continue this work, investigating the efficient processing of the semantic changes. The algorithms described only change states of semantic entities, whose textual representation changed. This should enable efficiently processing of the semantic changes.

It would also be desirable to extend this analysis with supporting other languages. We believe that the algorithms described can be used for efficient processing of other file formats. This still needs to be checked in practice.

10. Acknowledgements

The authors would like to thank the Test Competence Center of Ericsson Hungary for supporting this research and open sourcing the Titan project. The Titan project contains the implementations of the algorithms and is accesible as *Eclipse Titan* here: https://projects.eclipse.org/proposals/titan

References

[1] S. P. Abney, *Rapid incremental parsing with repair* in Proceedings of the 6th New OED Conference: Electronic Text Research, University of Waterloo, Waterloo, Ontario, 1990, pp. 19.

[2] M. Collins and B. Roark, *Incremental parsing with the perceptron algorithm* in Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics, ser. ACL 04., Association for Computational Linguistics, Stroudsburg, PA, USA, 2004. articleno. 111, http://dx.doi.org/10.3115/1218955.1218970

[3] C. Ghezzi and D. Mandrioli, *Augmenting parsers to support incrementality* Journal of the ACM, vol. 27, no. 3, ACM, New York, NY, USA, Jul. 1980., pp. 564579, http://doi.acm.org/10.1145/322203.322215

[4] L. Huang and K. Sagae, *Dynamic programming for linear-time incremental parsing* in Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, ser. ACL 10., Association for Computational Linguistics, Stroudsburg, PA, USA, 2010, pp. 10771086. http://dl.acm.org/citation.cfm?id=1858681.1858791

[5] F. Jalili and J. H. Gallier, *Building friendly parsers* in Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ser. POPL 82. New York, NY, USA, ACM, 1982, pp. 196206. http://doi.acm.org/10.1145/582153.582175

[6] J.-M. Larchevêque, *Optimal incremental parsing* ACM Transactions on Programming Languages and Systems, vol. 17, no. 1, ACM, New York, NY, USA, pp. 115, Jan. 1995. http://doi.acm.org/10.1145/200994.200996

[7] G. Linden, *Incremental updates in structured documents*, Ph.D. dissertation, (1993), Department of Computer Science, University of Helsinki, https://helda.helsinki.fi/bitstream/handle/10138/21469/abstract.pdf?sequence=2

[8] A. M. Murching, Y. V. Prasad, and Y. N. Srikant, *Incremental recursive descent parsing* Computer Languages, vol. 15, no. 4, Pergamon Press, Inc., Tarrytown, NY, USA Oct. 1990, pp. 193204, http://dx.doi.org/10.1016/0096-0551(90)90020-P

[9] T. Par, K. S. Fisher: *LL(*): the foundation of the ANTLR parser generator* in Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, ser. PLDI' 11., ACM New York, USA, 2011, pp. 425-436, ISBN: 978-1-4503-0663-8 doi: 10.1145/1993498.1993548

[10] L. Petrone, *Reusing batch parsers as incremental parsers* in Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science. London, UK, Springer-Verlag, 1995, pp. 111123. http://dl.acm.org/citation.cfm?id=646833.708027

[11] T. A. Wagner and S. L. Graham, *History-sensitive error recovery* in In preparation. 24/9/1997 17:26 PAGE PROOFS master, 1997.

[12] Li, Warren X., *A Simple and Efficient Incremental LL(1) parsing*, in proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM '95 (1995), Springer-Verlag, London, pp. 399–404, http://dl.acm.org/citation.cfm?id=647005.712013

[13] ITU, *Information technology Abstract Syntax Notation One (ASN.1): Specification of basic notation*, International Telecommunication Union, 07 2002. http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf

[14] ETSI, *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, European Telecommunications Standards Institute, 04 2012. http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.04.01_60/es_20187301v040401p.pdf

[15] Oracle Inc. *Java tuning white paper*, (2005), http://www.oracle.com/technetwork/java/tuning-139912.html

Appendix A. Procedures

Procedure 1 Change processing

```
1: procedure BackgroundThread(List of changes)
2:     List merged ← Merge(changes);
3:     for all change element of merged do
4:         Syntax_check(GetModule(edited_file), change);
5:     Semantic_check(getProject(module));
6: end procedure
```

Procedure 2 The simplified algorithm skeleton

```
1: procedure GenericUpdate(Node node, Interval change)
2:     for all child element of node do
3:         if child.end < change.start then
4:             continue;
5:         else if child.start > change.end then
6:             RecursiveUpdateLocation(child);
7:         else if (child.start < change.start) and (child.end > change.end)
   then
8:             result ← GenericUpdate(child, change);
9:             if result = true then
10:                UpdateLocation(child);
11:            else if Reparse(child) = false then
12:                return false;
13:        else
14:            return false;
15:    return true;
16: end procedure
```

Procedure 3 The algorithm for module level

```
1: procedure Update(Module module, Interval change)
2:     if (module.start < change.start) and (module.end > change.end) then
3:         result ← Update(module, change);
4:         if result = true then
5:             UpdateLocation(module);
6:             return ;
7:     ParseFile();
8: end procedure
```

Procedure 4 The algorithm for semantic lists

```
1: procedure UPDATE(ListNode node, Interval change)
2:     Number left ← node.start;
3:     Number right ← node.end - 1;
4:     Boolean enveloped ← false;
5:     for all child element of node do
6:         if child.start > change.start then
7:             rightBound ← Min(child.start-1, right);
8:         else if child.end < change.start then
9:             leftBound ← Max(child.end, left);
10:        else if (child.start < change.start) and (child.end > change.end)
   then
11:            result ← Update(child, change);
12:            if result = true then
13:                enveloped ← true;
14:    if ¬ enveloped then
15:        for all child element of node do
16:            if (child.start > left) and (child.end < right) then
17:                Remove(child);
18:        enveloped ← Reparse(node, left, right);
19:    if ¬ enveloped then
20:        return false;
21:    else
22:        for all child element of node do
23:            if (child.end > right) then
24:                RecursiveUpdateLocation(child);
25:        return true;
26: end procedure
```

```
1  module Example {
2    ...
3    function demonstration (in integer p_value) runs on
         demo_component_CT
4    {
5      ...
6      for (var integer i:= 0; i <= 10; i := i+1) {
7        ...
8        j := j + i;
9        ...
10     }
11     ...
12   }
13   ...
14 }
```

Listing 1. Example TTCN-3 code

Appendix B. Example operation

The example code (Listing 1) contains one function, with one loop and one statement shown, among any number of other definitions and statements.

If the addition (line 8) is changed to a substraction, the algorithm is called with the module node (line 1) and the location of the changed chaarcter. The module level (Procedure 3) forwards processing to the list of definitions, which determines (Procedure 4) the definition to be checked (line 3). The function definition checks (Procedure 2) it's parts and determines, that the statement-block might be able to handle the change. The statementblock searches the list of statements (Procedure 4) and finds that the loop statement (line 6) might handle the change. The for loop checks it's parts (Procedure 2) and finds that it's statementblock needs to be searched. In the statementblock of the for loop, the statement (line 8) containing the modification is found (Procedure 4).

At this point the tool needs to parse only a single line. Implementing the algorithms to reduce this region further can be a business decision, based on how much the next step in the implementation costs, and how big of an improvement it would bring.

Ericsson Telecommunications Hungary, H-1117 Budapest, Irinyi J. u. 4-20, Hungary

E-mail address: Kristof.Szabados@ericsson.com