Generator of automated tools for program instrumentation

Mikhail Onischuck

Peter the Great St.Petersburg Polytechnic University
Saint-Petersburg, Russian Federation
orcid.org/0000-0001-5359-0161

Vladimir Itsykson

Peter the Great St.Petersburg Polytechnic University
Saint-Petersburg, Russian Federation
orcid.org/0000-0003-0276-4517

***Abstract*. Instrumentation is one of the methods used in dynamic program analysis for assessing software performance. This paper proposes a technology for constructing software instrumentation tools for different programming languages. The instrumentation functions are described within this approach using several grammar-based DSLs. The obtained instrumentation toolkit is the result of generating a new system based on formal descriptions of the instrumentation process. The tracing functions are embedded into the original program using a TXL utility; a conversion program is also generated for this utility. The developed prototype was tested on 4 large projects written in different programming languages: Java, Python, C++ and Object Pascal. The tests confirmed the efficiency of the approach and the applicability of the developed prototype.**

Keywords — instrumentation, instrumentation tools generation, formal language grammar, TXL

I. Introduction

Controlling software quality is a major challenge for the software industry. IT companies spend tremendous efforts and resources on diverse methods and practices for software quality assurance. The starting stage generally consists in software quality analysis, checking whether the software is compliant with the specifications and searching for errors. Instrumentation is a mechanism used for software analysis. *Instrumentation* is commonly understood as inserting additional code into the source program, which allows detecting and monitoring the parameters characterizing the software performance, with options for debugging and troubleshooting [1].

There are different approaches to program instrumentation. *Manual* instrumentation implies that the developers independently log snippets, relying on their understanding of the program logic. Depending on the goals set in *automated* instrumentation, the developers use some tool for inserting the logging code into the source program based on certain rules. Unfortunately, most of the existing tools come with limitations restricting the options for code instrumentation. What is more, such tools are generally hard-coded to only work with a specific programming language, even though instrumentation may be required for programs in any language.

In this paper, we propose a universal approach to solving the instrumentation problem incorporating a declarative framework using the grammar of the target programming language, serving to generate automated systems for software instrumentation.

The paper is organized as follows. Section 2 describes the technology we have developed for generating instrumentation systems. Section 3 considers a prototype instrumentation tool generator. Section 4 presents the results of testing the approach on real software projects. Section 5 analyzes the existing solutions in automated instrumentation. The paper is concluded by summarizing the results and outlining the directions for further development.

II. Technology for generating instrumentation tools

In general, depending on the programming language and the runtime environment, either the source code or some intermediate representation (e.g., bytecode) can be instrumented. For example, [2] considers source code instrumentation in C++ for verifying potential vulnerabilities that can threaten the system security. A technology developed in [3] introduces an instrumentation framework based on analysis and processing of an intermediate representation containing some form of an abstract syntax tree. In contrast, [4] and [5] discuss bytecode instrumentation for a Java virtual machine.

Because our study deals with a universal instrumentation system independent of the programming language and runtime environment, the only way to organize this system would have to be source-code instrumentation.

Instrumentation consists in converting the source program into a new one, with special code added to it which enables tracing once the program is executed. This means that the task of instrumentation is reduced to converting (transforming) one source code into another.

A. Methodology

Since one of the requirements to the approach developed is the option to instrument programs written in different languages, it should incorporate a mechanism for generating instrumentation systems for different target languages. An obvious solution is to use the grammar of the target language as input for the instrumentation system. Since standard grammars are not originally designed for instrumentation, a grammar markup mechanism should be developed, adding semantics to the grammar for subsequent use in instrumentation rules. We achieve this by utilizing grammar annotations, created once for each target programming language. The actual instrumentation rules are developed by the user solving a specific applied problem.

The general schematic for the approach developed is shown in Fig. 1.

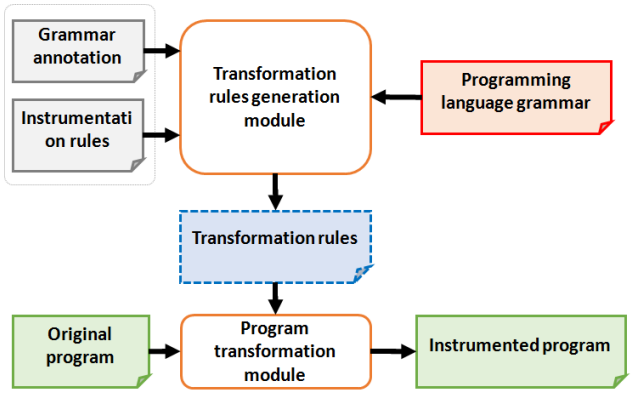

Fig. 1. General schematic of the generator.

The module for generating the transformation rules takes as input the grammar of the target language, the grammar annotation for instrumentation, and the instrumentation rules; the module then generates transformation rules based on these data, feeding them to the input of the program transformation module together with the original source code. The program transformation module in turn generates an instrumented program semantically equivalent to the original one with the added instrumentation code.

B. *Representation and processing of program source code*

Source code transformation has been examined in great detail; a wide range of tools have been developed that can effectively solve this task using some form of a parse tree or a concrete syntax tree (CST) for internal representation. The nodes of this tree represent syntactic structures and individual elements adopted in the target programming language, so *making changes* to the processed code (its transformation) can be distinguished as a separate stage of instrumentation (see Fig. 1). Such systems require a description of transformations at the level of individual parse tree nodes (and/or sets of nodes) to operate. While it can be difficult to understand this description or correlate it with the problem solved by the user, an additional complication is that it should strictly follow the grammar of the language used. These drawbacks lead us to identify another significant stage of instrumentation that is *preparing for changes* (generating a description of the transformations). The description of the changes can be simplified by introducing some primary representation with limited functions but at the same time universal enough to be used for processing source code in different programming languages. Because there are no generally accepted standards for grammar formatting and structuring, the above requirement means that the instructions describing the *purpose* of changes (instrumentation rules) should be separated from the instructions describing repeating *primitive operations* specific to the given grammar and *the syntax elements used* (grammar annotation, Fig. 1).

Taking CST as their internal representation, the systems for source code transformation process syntactic structures by moving from higher (file, module, expression set) to lower levels (string literals, digits). The order in which CST nodes are visited and whether the same node can be revisited depends solely on the specifics of a particular tool and the required transformations. Therefore, the instrumentation task can be solved using a method for step-by-step recursive descent down the parse tree to be modified by inserting additional code (the top-down one-pass method).

Let us define a workspace where instrumentation will be done, i.e., the *working context* that is some subset of CST nodes given as $Y = \{g \in G \mid A(g)\}$, where $A(g)$ is the condition (logical statement about the properties) that the element g belongs to some subset Y, while G is the entire set of nodes of a particular parse tree. At the same time, using a CST means using a large number of intermediate nodes in accordance with the grammar rules used constructing it. Consequently, information should be collected during the top-down traversal of the parse tree (a), to be subsequently used for (b) making a decision whether a node belongs to the instrumentation context before actually inserting the code.

Thus, we have formulated a method for processing parse tree nodes and a mechanism for controlling the workspace of this process.

C. *Transformation of program source code*

The source code of programs written in a wide range of programming languages can be processed either by developing specific tool finely tailored to the individual languages, or taking an existing tool or language (with the appropriate runtime environment), such as, for example, “The Meta-Environment” [6], Stratego/XT [7], Rascal [8], TXL [9] or similar. We have chosen the TXL language to confirm the applicability of the approach.

TXL is a domain-specific language designed to support source analysis and processing through rule-based structural transformation [9]. The FreeTXL compiler/interpreter (referred to as the TXL utility from now on) is the official runtime environment for programs in the TXL language.

The TXL language is relatively simple and pure; furthermore, the TXL utility offers such benefits as binary executables available for various software platforms and easy integration (providing XML output). For these reasons, we decided to use the TXL utility for transformation over the source code which is in turn produced by the instrumentation system generator. In view of this, the generated 'instrumentation system' is understood in this study as the *description of transformations* in the TXL language together with the grammar of the target programming language and the TXL runtime environment.

The *where* clause is used in TXL to constrain the composition for transformation rules and functions [10], while sequences of such expressions are combined by means of conjunction. On the other hand, applying logical disjunction to comparison operators from the start within this syntactic structure allows introducing expressions in conjunctive normal form (CNF), obtained from predicates constructed by users to describe instrumentation contexts.

Due to the functional nature of the TXL language, we decided to use the arguments/parameters of the chain of functions to pass the nodes collected during the traversal down the parse tree. In this case, the “chain” is a particular solution to the instrumentation problem, accounting for the above one-pass method for CST traversal.

D. Grammar annotation

It can be a challenge to extract the directed acyclic graph (DAG) representing the nested syntactic hierarchy from the grammar of an arbitrary programming language. This obstacle can be overcome by either manual grammar marking up or by constructing heuristics sufficient for the purpose of defining the scope of the instrumentation on the parse tree. For this purpose, we constructed an XML-based format describing grammar annotations, including such information as:

- description of the *syntactic structures* of the target programming language that are the most significant for the end user in accordance with the grammar, also including:
  - a text template describing the type of structure;
  - instrumentation points combined with a simplified instrumentation algorithm;
- directed acyclic *graph of the hierarchy* of nesting syntactic structures;
- *points of interest* describing the text data to be retrieved from the parse tree nodes;
- auxiliary user-defined functions in the TXL language.

E. User description of the instrumentation process

The prototype uses a declarative domain-specific language (DSL) to describe user-defined rules; it is based on the languages of such projects as Annotation File Utilities [11] and AspectJ [12]. Fig. 2 shows an example of the rules described using the developed language from the system's end user perspective, aimed at logging the first if-statement executing in the context of the "main" method contained in the "Main" class.

```
use fragment "logging/dependencies"
use fragment "logging/fields"
use fragment "logging/init"
use fragment "logging/message"

context class:
  { poi:class_name = "Main" }

context method:
  { poi:method_name = "main" }

context class_and_method:
  class & method

rules:
  print_usefull_message_to_the_log:
    @@ -> [first] imports # all:
      add:
        logging_dependencies

    @class -> [first] body # all:
      add:
        logging_fields

    @class_and_method -> [first] if # before:
      make:
        msg <- $pointcut + " " + $node + " block in " + $file + " class, in " + $method_name + " method";
      add:
        logging_message(msg)
```

Fig. 2. End user description of the instrumentation rules.

The following main components comprise the user description of the set of rules are (the number of the list item corresponds to the circled number in the figure):

1) listing the source code *fragments* used in this set of rules and their relative file paths;

2) listing the instrumentation *contexts* of interest to the user (both *simple*, i.e., a set of statements about the properties of a syntactic structure, and *composite*, i.e., several contexts joined by first-order logical operators);

3) grouping the instrumentation steps as *named rules*;

4) *refining the instrumentation context* using programming language keywords, modifiers (enclosed in square brackets) and text patterns, if any are required for the task to be solved by the user;

5) setting specific *instrumentation points* by their identifiers;

6) creating *user-defined variables* from text elements and constant values;

7) a *namelist of fragments* to be inserted simultaneously into the same place, *specifying the parameters*, if any are required according to the text of the snippet used.

The main benefits of the DSL developed is that it provides two methods for describing instrumentation contexts (item *2*) and an option for successively refining the context specified (items *4*, *5*). This way, the user can considerably limit the workspace for transformations while the implementation details remain concealed. Nevertheless, the transformation descriptions output by the generator can be used as the initial step for more complex instrumentation routines.

F. Tool generation procedure

The *input* artifacts for the “generator and transformation tool” system are the following:

- source code of the program to be instrumented;
- grammar description of the target language that the source code is written in;
- grammar annotation;
- description of user-defined instrumentation rules;
- source code snippets (i.e. “fragments”) in the target programming language to be inserted;
- additional startup and runtime environment parameters.

As an *intermediate* output, the generator provides the instructions for the transformations to be performed with a specific input file with the source code in accordance with the grammar given for the target programming language.

The *output* artifact is the source code in the target programming language, which has been subjected to the required transformations.

The developed prototype automatically generates the transformation instructions as a set of interconnected TXL functions in the following order:

1) load, parse and check the dependences of the source code fragments used in accordance with the rules described by the end user;

2) calculate the maximum distances from the root node for each node of the DAG representing the key structures of the target language in accordance with the grammar annotation provided;

3) build wrappers over standard comparison operators;

4) build functions for implementing the tasks assigned to the points of interest;

5) build functions checking whether CST nodes belong to contexts;

6) build function chains in accordance with instrumentation rules allowing for the user's requirements;

7) build auxiliary TXL functions;

8) build user-defined functions;

9) build the *main* TXL function and apply policies for additional user-defined functions;

10) update the states of functions that are chain elements;

11) generate TXL instructions for the required transformations and call the TXL utility.

The general structure of a typical chain of calls to programmer-defined domain-specific functions:

1) *C-functions* (*collect*) are *rule*-type TXL functions, designed to accumulate information from the parse tree nodes. This information is later used to assess whether the node belongs to the chosen instrumentation context. Such functions operate by calling the next function from the chain and passing it all the values of the arguments that were received by the current function, together with the node considered.

2) *F-function* (*filter*) is a function designed to filter CST nodes relative to contexts described by the end user by calling the auxiliary function for assessing whether the node belongs to the context and passing it the collected nodes.

3) *R-functions* (*refine*) are functions designed to refine the context to a limited subset of some required syntactic structures of the target programming language in accordance with one of the implemented modifiers:

a) *"first"* searches and processes only the *first* encountered node from the current subtree in accordance with the type specified in the grammar annotation;

b) *"all"* searches and processes *all nodes* in accordance with the type specified in the annotation;

c) *"level"* searches and processes the nodes located at the *same* (first) nesting level. A schematic example illustrating how this modifier works is given in Fig. 3: different nesting levels of syntactic structures are shown from the bottom-up, a sequence of structures from the standpoint of source code is shown from left to right; the colors correspond to different types of CST nodes (nodes of the required type are colored in red; nodes to be processed are colored in dark red); the numbers indicate the order in which the pass is performed in the TXL environment.

4) *I-function* (*instrument*) are functions designed directly for instrumentation in accordance with the patterns (search and replace) specified in the annotation and the operation algorithm.

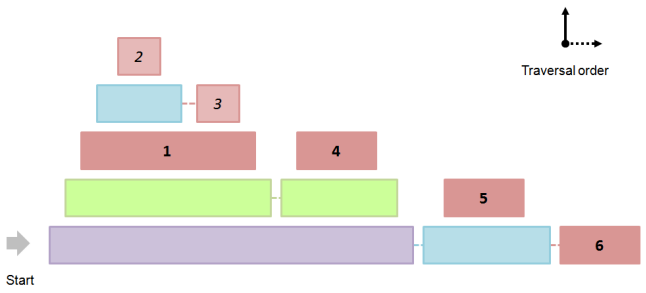

Fig. 3. Operation of level modifier visualized.

Function chains should be generated for each individual expression containing a context refinement and the keyword *"add"* (see Fig. 2) together with a list of code fragments, as a separate rule (group of refinements) in accordance with the description order.

III. Prototype implementation

To test the efficiency of the above approach, we constructed a prototype instrumentation tools generator combining the generator application built based on the TinyXML2 [13] and Boost [14] libraries in C++ for the purpose of parsing grammar annotations along with user descriptions and interaction with the TXL utility, respectively, and the application itself. Fig. 4 shows the general schematic for the prototype together with the artifacts necessary to solve the problem posed: in accordance with the initial model (Fig. 1), the generator utility is a module producing an intermediate (optionally cached) description of instrumentation instructions in a format that can be used by the second part of the two, i.e., the transformation system. The generator utility acts as a module producing the transformation rules in this case, while the TXL utility performs the function of the program transformation module. The colors of the arrows in the figure correspond to different frequencies of analysis and processing of artifacts by the generator and the TXL utility (orange is more frequent, purple is less frequent), the colors of the input artifacts characterize the degree to which it is difficult for the user to create them (red is very difficult, blue is moderately difficult, green is easy).

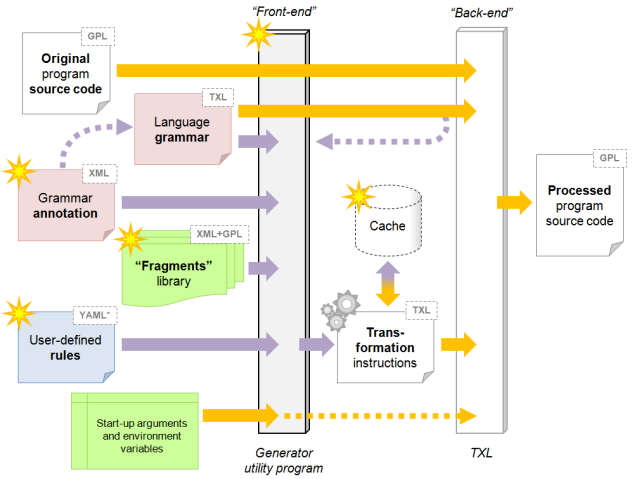

Fig. 4. General schematic of the prototype.

IV. Prototype testing

The applicability of the developed instrumentation method and the functionality of the implemented prototype were verified using several industrial open source projects. The projects and the source code samples were chosen based on the capabilities of the TXL grammars available at the time of this study [15]; notably, some elements of the grammars were slightly modified to better suit the described instrumentation approach (increased separation of syntactic structures). As a result, four projects were chosen for the experiments, written in four different programming languages:

- AspectJ [12] is a system and DSL designed to implement aspect-oriented programming principles within the Java language. Version 1.9.5 was chosen for the experiment.
- Keras library [16] is an add-on library for high-level processing and construction of deep learning neural network models for the Python language. Version 2.3.1 was chosen for the experiment.
- Boost library [14] is a multifunctional modular library for building software products using the C++ language. Version 1.72.0 was chosen for the experiment.
- Lazarus IDE [17] is a graphical cross-platform environment for rapid application development in the Object Pascal language and its dialects. Version 2.0.8 was chosen for the experiment.

Different syntactic structures of the languages were instrumented in the experiments, such as import sections, class bodies, methods, branch operators, and loops, in particular using such means as different nesting levels of programming language structures. The source codes of the performed experiments are available in [18].

The approach was found to be efficient the most for instrumenting an AspectJ project written in Java. As for other projects, we found both minor limitations associated with multivariate forms of some syntactic structures (for example, the import statement in Python), and major difficulties when the descriptive capabilities of the grammars of languages provided by the TXL developers and/or the community were found to be insufficient. In particular, the Object Pascal grammar covered about 80% of the code base of the Lazarus IDE project at the time of the study, while the C++ grammar (specifically designed for the older version of the C++ standard) covered only 21% of the Boost library base. Simplified language grammars should be further refined for the developed prototype to be used industrially, providing full support for the standards of these languages.

We can conclude from our findings that the proposed approach and the developed prototype generator are largely applicable for the tasks described. There are certain limitations because existing grammars of the programming languages are imperfect; moreover, the prototype constructed has some drawbacks yet to be eliminated.

V. Comparison with counterparts

There is a wide range of different software systems offering options for automating the instrumentation process to some degree. Examples of such systems include tools for test coverage analysis (*GCC Gcov* [19], *Froglogic Squish Coco Coverage Scanner* [20]), code analysis (*Testwell CTC++ Preprocessor* [21], *Bullseye Coverage* [22]), tracing and statistics calculations (*Google Web Tracing Framework* [23]), vulnerability assessment (see [2] and [4]).

Each of these projects only works with a very *small subset* of programming languages, and they need to be further developed and adapted to work with new languages. For open source projects, this can be done by forking the main code base but this takes a lot of time and resources. Commercial tools (Testwell CTC++, Bullseye Coverage and Froglogic Squish Coco) can only be expanded by the developers.

Compared to such instrumentation methods as, for example, bytecode processing [5], application of the aspect-oriented programming paradigm [24], or reduction to a single intermediate representation with subsequent reconstruction [3], the main difference of the approach proposed our study is that the user can flexibly control the instrumentation process and adapt the instrumentation for other programming languages by specifying instrumentation rules in terms of the target programming language, also independently creating and annotating grammars in terms of parsing texts in formal languages.

VI. Conclusion

The study presents an approach to software instrumentation, with a prototype developed for a generator for automated instrumentation tools, for which the DSL of instrumentation rules and the format for writing annotations for formal grammars were described. We tested the prototype on several industrial open-source projects, confirming that it was functioning properly and that the approach could be applied successfully.

If a program is represented as a text in some formal language with a developed grammar describing the structures of this language, this approach can be considered sufficiently universal for solving the instrumentation problem. This, however, implies that the capabilities of such a generator mainly depend on the capabilities of the transformation system applied and the grammar of the target language, which was confirmed experimentally. High performance is the most crucial factor for program execution: it can be achieved by making the program as close as possible to a machine-generated semi-structured format, removing most of the information that does not improve this indicator (i.e., information about the original structure of the program). All of this limits the potential applications of the approach to an environment with access to structural information about the program.

The technique can be further developed by expanding the DSLs constructed, which can allow overcoming the existing context constraints, and conducting more focused experimental studies for a wider range of programming languages.

VII. References

[1] Ломакина Л., Вигура А. Тестирование программных систем на основе компьютерной алгебры. Международный журнал экспериментального образования. 2016, 10-1, 132–134.

[2] Li H. et al. Automated source code instrumentation for verifying potential vulnerabilities. IFIP International Conference on ICT Systems Security and Privacy Protection. Springer, Cham. 2016, 211-226.

[3] Chittimalli P. K., Shah V. GEMS: a generic model based source code instrumentation framework. IEEE Fifth International Conference on Software Testing, Verification and Validation. 2012, 909-914.

[4] Chander A., Mitchell J. C., Shin I. Mobile code security by Java bytecode instrumentation. Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01. 2001, T.2., 27-40.

[5] Binder W., Hulaas J., Moret P. Advanced Java bytecode instrumentation. Proceedings of the 5th international symposium on Principles and practice of programming in Java. 2007, 135-144.

[6] The Meta-Environment. Centrum Wiskunde & Informatica [online]. [viewed 13.02.2021]. Available from: http://www.meta-environment.org/

[7] Stratego Program Transformation Language [online]. [viewed 13.02.2021]. Available from: http://strategoxt.org/

[8] Rascal MPL - About. Centrum Wiskunde & Informatica [online]. [viewed 13.02.2021]. Available from: https://www.rascal-mpl.org/about/

[9] About TXL. Queen's University and the Txl Project [online]. [viewed 10.02.2021]. Available from: http://txl.ca/txl-abouttxl.html

[10] Cordy J. Excerpts from the TXL cookbook. International Summer School on Generative and Transformational Techniques in Software Engineering. Springer. 2009, 27–91.

[11] Annotation File Format Specification. Веб-сайт проекта The Checker Framework [online]. [viewed 10.02.2021]. Available from: https://checkerframework.org/annotation-file-utilities/annotation-file-format.html

[12] The AspectJ Project. The Eclipse Foundation [online]. [viewed 10.02.2021]. Available from: http://www.eclipse.org/aspectj/

[13] Lee Thomason. TinyXML2 is a simple, small, efficient, C++ XML parser that can be easily integrated into other programs [online]. [viewed 10.02.2021]. Available from: https://github.com/leethomason/tinyxml2

[14] Boost C++ Libraries. Boost.org [online]. [viewed 10.02.2021]. Available from: https://www.boost.org/

[15] TXL World. Queen's University and the Txl Project [online]. [viewed 10.02.2021]. Available from: https://txl.ca/txl-resources.html

[16] Keras Team. Keras: the Python deep learning API [online]. [viewed 10.02.2021]. Available from: https://keras.io/

[17] Lazarus and Free Pascal Team. Lazarus Homepage [online]. [viewed 10.02.2021]. Available from: https://www.lazarus-ide.org/

[18] Onischuck Mikhail. Source Code Instrumentation System [online]. [viewed 14.02.2021]. Available from: https://github.com/dog-m/SCIS/tree/master/example/experiments_with_production_level_projects

[19] Gcov (Using the GNU Compiler Collection (GCC)). Free Software Foundation, Inc [online]. [viewed 10.02.2021]. Available from: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[20] Squish Coco 4.3.2. Froglogic GmbH [online]. [viewed 10.02.2021]. Available from: https://doc.froglogic.com/squish-coco/latest/squishcoco.pdf

[21] Testwell CTC++ Description. Testwell [online]. [viewed 10.02.2021]. Available from: http://www.testwell.fi/ctcdesc.html

[22] BullseyeCoverage Measurement Technique. Bullseye Testing Technology [online]. [viewed 10.02.2021]. Available from: https://bullseye.com/ measurementTechnique.html

[23] Instrumenting Code with tracing-framework by Google. Google Inc [online]. [viewed 10.02.2021]. Available from: https://google.github.io/tracing-framework/instrumenting-code.html

[24] Mahrenholz D., Spinczyk O., Schroder-Preikschat W. Program instrumentation for debugging and monitoring with AspectC++. Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002. 2002, 249-256.