Journal of Siberian Federal University. Engineering & Technologies, 2018, 11(7), 801-810
~ ~ ~

УДК 004.43

# Static Analysis Technique for Searching Natural Semantic Defects of Program Objects and Software Implementation of This Approach Based on the LLVM Compiler Infrastructure and the Clang Frontend

**Dmitry S. Victorov,**
**Evgeny N. Zhidkov and Roman E. Zhidkov***
*Military Academy of Aero-Space Defense*
*named after the Marshal of Soviet Union G.K. Zhukov*
*50 Zhigareva Str., Tver, 170022, Russia*

Received 04.09.2018, received in revised form 27.09.2018, accepted 12.10.2018

*The article proposes a static analysis technique for searching functional software defects. Development is caused by the need to create high quality domestic military automated system software in conditions of temporary and financial constraints. The use of static analysis makes it possible to automate the process of error detection and it allows to localizing defects simply. The priority object of inspections should be special software for newly created automated systems because it is exclusively, scale and logical complexity. An approach based on controlling the implementation of principle of dimensional homogeneity of the physical equations presented in the software. For this purpose, program objects identifiers are assigned an invariable attribute in the form of a physical dimension vector of interpreted value. The set of all vectors forms a semantic domain used in verifying the relationships between program objects by analogy with the analysis of dependencies in the abstract interpretation. The technique uses a program internal representation for calculations of derivate vectors on the basis of operations with initial program objects. The software implementations uses the LLVM compiler and Clang frontend to translate code in C, C++, Objective-C. Clang API internal representation in the form of abstract syntax tree is modified to store data about program objects vectors.*

*Keywords: automated systems, software, program object, verification, static analysis, functional defects, abstract syntax tree, LLVM, Clang.*

Citation: Victorov D.S., Zhidkov E.N., Zhidkov R.E. Static analysis technique for searching natural semantic defects of program objects and software implementation of this approach based on the LLVM compiler infrastructure and the Clang frontend, J. Sib. Fed. Univ. Eng. technol., 2018, 11(7), 801-810. DOI: 10.17516/1999-494X-0095.



* Corresponding author E-mail address: r-zhdkv@yandex.ru

# Методика статического анализа для поиска дефектов естественной семантики программных объектов и ее программная реализация на базе инфраструктуры компилятора LLVM и фронтенда Clang

**Д.С. Викторов, Е.Н. Жидков, Р.Е. Жидков**
*Военная академия воздушно-космической обороны*
*им. Маршала Советского Союза Г.К. Жукова*
*Россия, 170022, Тверь, ул. Жигарева, 50*

*В статье описывается методика статического анализа для поиска функциональных дефектов программ, разработка которой обусловлена необходимостью создания качественного отечественного программного обеспечения автоматизированных систем военного назначения в условиях временных и финансовых ограничений процесса разработки. Использование статического анализа позволит автоматизировать процесс выявления дефектов и упростит их локализацию в рамках мероприятий процесса верификации, при этом приоритетным объектом проверок должно быть специальное программное обеспечение вновь создаваемых автоматизированных систем ввиду своей эксклюзивности, масштабности и логической сложности. Предлагается подход, основанный на контроле выполнения принципа размерной однородности физических уравнений, представленных в программе, для чего идентификаторам программных объектов назначается неизменный признак в виде вектора физической размерности интерпретируемой величины. Совокупность всех векторов образует семантический домен, используемый при проверке соотношений между программными объектами по аналогии с анализом зависимостей при абстрактной интерпретации. Для проверки используется внутреннее представление программы, по которому происходят вычисления производных векторов на основании операций с исходными программными объектами. Программная реализация предлагаемой методики основывается на компиляторе LLVM и фронтенде Clang для трансляции кода на языках C, C++, Objective-C. Получаемое с помощью API Clang абстрактное синтаксическое дерево модифицируется для хранения данных о векторах программных объектов.*

*Ключевые слова: автоматизированные системы, программное обеспечение, программный объект, верификация, статический анализ, функциональные дефекты, абстрактное синтаксическое дерево, LLVM, Clang.*

## Ведение

При разработке современных автоматизированных систем (АС) происходит процесс непрерывного расширения функциональных возможностей, осуществляемый в основном за счет усложнения логики и увеличения размера используемого специального программного обеспечения (СПО), ввиду чего огромное внимание уделяется вопросам повышения его качества, которое в значительной степени зависит от возможного содержания программных дефектов. Серьезность сложившейся в программной инженерии ситуации подтверждается фактом увеличения доли затрат на выявление и устранение дефектов до 80 % от общей стоимости программного обеспечения (ПО) [1].

Статический анализ (СА), применяемый в рамках верификации программного обеспечения, зарекомендовал себя высокоэффективным методом, позволяющим снижать суммарные трудозатраты на поиск и локализацию дефектов ПО и, как следствие, уменьшать стоимость разработки за счет значительной степени автоматизации процедуры его проведения при наличии достаточных вычислительных ресурсов [2, 3]. Кроме того, возможность применения СА во время кодирования позволяет сокращать до пяти раз стоимость исправления дефектов ПО в сравнении с этапом тестирования, что дополнительно актуализирует необходимость использования анализаторов [1].

Выигрыш в снижении трудозатрат на верификацию ПО может быть увеличен за счет расширения характерного для СА множества обнаруживаемых дефектов, которое в основном образовано нефункциональными дефектами (НФД) [4]. Примерами НФД для наиболее востребованных языков программирования *C* и *C++*, используемых при создании ПО автоматизированных систем (АС) военного назначения (ВН), являются: использование неинициализированного (освобожденного) указателя или указателя на NULL, утечка ресурсов, выход за границы статических и динамических объектов и др.

В настоящее время основным методом обнаружения функциональных дефектов (ФД), которые являются следствием нарушений спецификации функциональных возможностей программной системы, признано тестирование [5]. Данный подход в общем случае не может быть автоматизирован, а с ростом размера и сложности программ, при выполнении требования о полном тестовом покрытии, становится высоко трудоемким, кроме того, требующим тратить дополнительные силы на локализацию причин невыполнения теста [2]. Следовательно, актуальна задача поиска альтернативных решений для выявления ФД с использованием менее трудоемких способов, например методик СА.

Нарушение требований функциональности ПО АС ВН зачастую связано с искажением алгоритмов СПО, реализующих информационно-расчетные задачи (ИРЗ), что выражается не только в нарушении порядка вычислений (дефекты в уловных и циклических конструкциях), но и в ошибочной записи вычисляющих выражений в операторах программы (подмена истинных значений операций и операндов). Для ряда АС ВН, СПО которых оперирует данными с ясным физическим смыслом, проверку программ на отсутствие дефектов первого и второго видов возможно организовать по принципу размерной однородности физических уравнений ИРЗ.

Обнаруживаемые вышеуказанным способом дефекты предлагается называть дефектами естественной семантики (ДЕС) программных объектов (переменных, функций, полях и методах классов) с целью акцентировать внимание на их различную природу с семантическими дефектами, вызываемыми нарушением требований спецификации языка программирования.

Применение СА в качестве инструмента для поиска ФД возможно реализовать по аналогии с анализом зависимостей между программными объектами при абстрактной интерпретации, где для описания состояния программы используется подмножество логических утверждений, описывающих состояние объектов программы в текущий момент времени. Возможные предикаты разбиваются на утверждения о значениях объектов и утверждения о зависимостях между ними. Домен возможных значений программных объектов выбирается в соответствии с решаемой задачей [6].

Под размерностью физической величины понимается выражение, показывающее ее связь с основными величинами выбранной системы физических величин [7]. В Международной системе величин (СИ) в качестве основных выбраны: длина ($L$), масса ($M$), время ($T$), электрический ток ($I$), термодинамическая температура ($\Theta$), количество вещества ($N$) и сила света ($J$). А в качестве дополнительных – две безразмерные величины: плоский ($\alpha$) и телесный угол ($\Omega$). Таким образом, размерность произвольной физической величины ($A$) возможно записать в виде произведения степеней сомножителей, соответствующих семи основным физическим величинам и двум дополнительным, в которых опущены численные коэффициенты: $dim(A) = L^{e_1} M^{e_2} T^{e_3} I^{e_4} \Theta^{e_5} N^{e_6} J^{e_7} \alpha^{e_8} \Omega^{e_9}$, где $e_j$ – степень $j$-го сомножителя формулы размерности физической величины. Если зафиксировать порядок сомножителей в указанной выше формуле, то для описания размерности будет достаточно вектора, состоящего только из степеней этих сомножителей.

Применительно к рассматриваемой задаче программным объектам ставится в соответствие размерность интерпретируемой в них физической величины, такие утверждения относительно свойств программных объектов имеют точный характер и представляются в виде вектора естественной семантики (ВЕС): $v_{obj} = e_1 \times e_2 \times \ldots \times e_9$, $e \in Q$, где $e$ – степень сомножителя формулы размерности физической величины, интерпретируемой в программном объекте; $Q$ – множество рациональных чисел.

Совокупность возможных ВЕС образует пространство ($V$), которое используется в качестве семантического домена. Формирование домена происходит из внешних источников на основании данных о программных объектах. В данном пространстве естественным образом заданы операции поэлементно сложения и умножения на скаляр, исходя из операций со степенями сомножителей:

$$\vec{s}_1 + \vec{s}_2 = (e_{11} + e_{21}, e_{12} + e_{22}, ..., e_{19} + e_{29},), \vec{s}_1, \vec{s}_2 \in S;$$

$$\lambda\vec{s} = (\lambda e_1, \lambda e_2, ..., \lambda e_9,), \lambda \in Q, \vec{s} \in S.$$

Представленные операции удовлетворяют основным аксиомам линейного пространства, что позволяет говорить о принадлежности к рассматриваемому пространству ВЕС, являющихся результатом выполнения заданных операций.

Связь между программными объектами осуществляется через значения их ВЕС ($v^{obj}$) и имеет различный характер:

арифметические зависимости: $arith\left(v_1^{obj}, v_2^{obj}, \otimes, v_3^{obj}\right) : v_1^{obj} = v_2^{obj} \otimes v_3^{obj}$, где $\otimes \in \{+, -\}$ – арифметическая операция с ВЕС;

зависимости сравнения: $comp\left(v_1^{obj}, \odot, v_2^{obj}\right) : v_1^{obj} \odot v_2^{obj}$, где $\odot \in \{=, \neq\}$ – операция сравнения ВЕС.

### Выбор промежуточного представления ИК программы

СА развился из теории компиляторов и потому тесно связан с принципами их работы, где для генерации кода на целевом языке могут использоваться различные промежуточные представления программы: дерево разбора (ДР), абстрактное синтаксическое дерево (АСД), граф потока управления (ГПУ) и др. [8].

Для анализа достижимости используется ГПУ, который показывает множество всех возможных путей исполнения программы, при этом узлы графа соответствуют базовым блокам – прямолинейным участкам кода без операций передачи управления и точек, принимающих управление из других частей программы. Такое представление программы исключает возможность работы с вычисляющими выражениями, необходимыми для поиска ДЕС.

Результатом грамматического анализа является ДР (иногда конкретное синтаксическое дерево), применительно к решаемой задаче оно имеет избыточную структуру, показывающую последовательность применяемых при разборе кода синтаксических правил, которые не отражают семантику исследуемой программы.

АСД представляет собой конечное помеченное дерево, в котором внутренние вершины обозначены операторами языка программирования, а листья – соответствующими операндами. Эта структура данных сохраняет семантику исходной программы, не перегружена особенностями применения синтаксических правил и может формироваться уже на второй стадии трансляции – синтаксическом анализе. При этом обход дерева позволяет производить вычисления выражений в терминах исходной программы (идентификаторы программных объектов). Вышесказанное в совокупности предопределяет использование АСД в качестве промежуточного представления программы при поиске ДЕС.

Для организации работы с ВЕС требуется в АСД ($N_S$) хранить дополнительную информацию о программных объектах, для чего предлагается модифицированная структура узла ($n$), представляющая из себя шестерку:

$$n = \left\langle m_n, v_n, p_n, a_n, f_n, \left\langle C_n, \prec \right\rangle \right\rangle, \tag{1}$$

где $m_n \in L_S$ – метка узла; $v_n \in V_S$ – ВЕС программного объекта; $p_n \in N_S$ – ссылка на «родительский» узел; $a_n$ – счетчик количества заходов в узел при обходе АСД; $f_n$ – признак наличия дефекта в узле; $\left\langle C_n, \prec \right\rangle$ – упорядоченное множества «потомков» узла, представленного непосредственно самим множеством $C_n \subseteq N_S$, а также отношением порядка $\prec$ на $C_n$. Далее по тексту упорядоченное множество «потомков» $\left\langle C_n, \prec \right\rangle$ для упрощения будем обозначать $C_n$.

## Методика СА для поиска ДЕС

Реализация механизма поиска ДЕС на базе СА представлена в виде набора функций, использующих исходный код (ИК) программы и данные о ВЕС программных объектов, для получения и манипулирования АСД. Структурная схема методики изображена на рис. 1.

Входными данными для методики является ИК программы в виде последовательности символов $s \in S$, по которому происходит формирование АСД на стадиях лексического и синтаксического анализа. Принципы исполнения данных стадий зависят от типа выбранного компилятора (статического анализатора); абстрагируясь от особенностей реализации, представим их работу в виде функций.

Работа лексического анализатора может быть описана многократным применением функции *lexical*: $S \rightarrow L$. В результате для входной последовательности символов ($s \in S$) формируется подмножество лексем ($L_S \subseteq L$), представляющее собой $L_S = I_S \cup C_S \cup O_S \cup K_S \cup D_S$, где $I_S \subseteq I$ – идентификаторы; $C_S \subseteq C$ – константы; $O_S \subseteq O$ – знаки операций; $K_S \subseteq K$ – ключевые слова; $D_S \subseteq D$ – разделители.


Рис. 1. Структурная схема методики СА для поиска ДЕС

Fig. 1. Structural scheme of the static analysis technique for natural semantic defects detection

Синтаксический анализатор, который возможно изобразит функцией *syntactic*: $L \to N$, на основе выделенных лексем $L_S$ строит внутреннее представление программы в виде АСД ($N_S \subseteq N$).

Программные объекты, признаками которых являются ВЕС, в ИК программы представлены идентификаторами $I_S$. Их выявление из полученных лексем $L_S$ производится с помощью многократного применения функции *getid*: $L \to I$. Реализация функции следующая:

$$getid(l) = \begin{cases} l, & \text{если } l \in I; \\ \varnothing, & \text{иначе.} \end{cases} \tag{2}$$

Для хранения ВЕС $v \in V_S$ в модифицированной структуре узла АСД предусмотрено поле, заполнение которого производится путем отображения множества идентификаторов программных объектов $I_S \subseteq I$ на множество возможных ВЕС: *setnsv* : $I \to V$.

Выполнение и контроль корректности операций с ВЕС производится во время обхода модифицированного АСД, который начинается с корневого узла. Для получения корня АСД применяется функция *getroot*: $N \to N$, в результате возвращающая узел, который не имеет «родителя»:

$$getroot(n) = \begin{cases} n, & \text{если } p_n = \varnothing; \\ \varnothing, & \text{иначе.} \end{cases} \tag{3}$$

Функция для обхода *tour*: $N \to N$ принимает значение корневого элемента дерева $n \in N$. Затем для узла $n$, полученного в качестве параметра, проверяется условие существования «потомков», которые не были посещены $a_c = 0$, и функция *tour* вызывает сама себя с непосещенным узлом $c_n$ в качестве параметра, при этом счетчик посещения узла $a_n$ увеличивается на единицу (спуск вниз). В случае обхода всех «потомков» $a_c > 0$ узла $n$ или если рассматриваемый узел является листом $C_n = \varnothing$, функция *tour* вызывает себя с параметром узла «родителя» $p_n$, счет-

чик узла $a_p$ увеличивается на единицу, выполняется расчет значения ВЕС узла «родителя» $p_n$ с помощью функции *calc* ($n$, $p_n$), осуществляется контроль условия корректности операций с ВЕС функцией *test* ($n$, $p_n$) (подъем вверх). При получении функцией обхода корневого узла, у которого все «потомки» уже были хотя бы раз пройдены, рекурсивный вызов прекращается и функцией *tour* возвращается значение полученного узла. Рекурсивная функция *tour* имеет следующую реализацию:

$$tour(n) = \begin{cases} tour\left(c_n = \left\langle m_c, v_c, p_c, (a_c + 1), f_c, C_c \right\rangle\right), & \text{если } \exists c_n \in C_n : a_c = 0; \\ tour\left(p_n = \left\langle m_p, calc(n, p_n), p_p, (a_p + 1), test(n, p_n), C_p \right\rangle\right), & \text{если } C_n = \varnothing \text{ или } \forall c_n \in C_n : a_c > 0; \\ n = \left\langle m_n, v_n, p_n, (a_n + 1), f_n, C_n \right\rangle, & \text{иначе.} \end{cases} \quad (4)$$

Получение производных значений ВЕС в «родительских» узлах АСД вычисляются функцией *calc*: $N \times N \to V$, которая возвращает значение ВЕС в зависимости от метки узла $m_p$. Реализация функции *calc* основана на формулах преобразования ВЕС программных объектов для операторов языков *C* и *C++*:

$$calc(n, p_n) = \begin{cases} v_p = v_p + v_n, & \text{если } \forall m_p \in \{*, *=\} : a_p > 1; \\ v_p = v_p - v_n, & \text{если } \forall m_p \in \{/, / =\} : a_p > 1; \\ v_p = v_n, & \text{если } \forall m_p \in \{=, +, + =, -, - =, ==, != , <, <=, >, >=\} \text{ или } \forall m_p \in \{*, *=, /, / =, \%, \% =\} : a_p \le 1; \\ v_p = v_p, & \text{иначе.} \end{cases} \quad (5)$$

Функция формирования признака наличия ДЕС *test*: $N \times N \to F$ вызывается при возвращении в «родительские» узлы. Множеством возможных значений функции является $F = \{0,1\}$, где ноль соответствует отсутствию ДЕС, а единица – наличию. Функция реализуется исходя из особенностей условий корректности операций с ВЕС программных объектов следующим образом:

$$test(n, p_n) = \begin{cases} 1, & \text{если } \forall m_p \in \{+, -, =, + =, - =, ==, != , <, >, <=, >=\} : a_p > 2, v_n \ne v_p \\ & \text{или } \forall m_p \in \{\%, *=, / =, \% =\} : a_p > 2, v_n \ne 0; \\ 0, & \text{иначе.} \end{cases} \quad (6)$$

Формирование множества ДЕС $D_S \subseteq D$ осуществляется функцией *getdef*: $N \to N$, применяемой к каждому помещаемому узлу АСД. Данная функция реализуется следующим образом:

$$getdef(n) = \begin{cases} n, & \text{если } f_n \ne 0; \\ \varnothing, & \text{иначе.} \end{cases} \quad (7)$$

Правильность программы в целом на предмет отсутствия ДЕС производится функцией *getsol*: $D \rightarrow W$, где получение элемента множества $W = \{1,0\}$, равного единице, говорит о наличии ДЕС в программе, ноль – об их отсутствии:

$$getsol(n) = \begin{cases} 1, & \text{если } D_S \neq \varnothing; \\ 0, & \text{иначе.} \end{cases} \tag{8}$$

В итоге на выходе предлагаемой методики получается признак $w_s \in W$, характеризующий правильность программы, на основании множества выявленных в процесса анализа ДЕС $D_S \subseteq D$.

## Программная реализация методики СА для поиска ДЕС

Программная реализация описанного способа поиска ФД осуществлена на основе инфраструктуры компилятора *LLVM* и фронтенда *Clang* для языков программирования *C*, *C++*, *Objective-C*. Архитектура компилятора *LLVM* сходна с другими современными компиляторами и представлена на рис. 2.

Осуществление поиска ДЕС возможно производить до этапов оптимизации и кодогенерации, что позволяет в работе ограничиться фронтендом фреймворка *LLVM – Clang*. Основным преимуществом *Clang* является его ориентация на максимальное сохранение информации о ходе процесса компиляции и коде программы, что гарантирует его точное воспроизведение в АСД.

Фронтенд дает возможность доступа к АСД через механизм курсоров (узлов), а также удобный инструментарий для его обхода и манипулирования. Стандартной структуры, описывающей курсор в *Clang* (*CXCursor*), недостаточно, поэтому при решении задачи на ее основе был создан ее модифицированный вариант (рис. 3), соответствующий выражению (1).


Рис. 2. Архитектура компилятора LLVM

Fig. 2. LLVM compiler architecture

```
typedef struct {
    CXCursor cursor; //исходный курсор, содержащий метку узла АСД
    int nsv[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0}; //массив, представляющий ВЕС программных объектов
    int parent = 0; //ссылка на родительский элемент - номер элемента в векторе
    int counter = 0; //счетчик количества заходов в узел при обходе
    bool defect = 0; //признак наличия дефекта в узле
} ExtendedCXCursor; //расширенная структура курсора АСД
```

Рис. 3. Структура модифицированного узла АСД

Fig. 3. Structure of a modified abstract syntax tree node

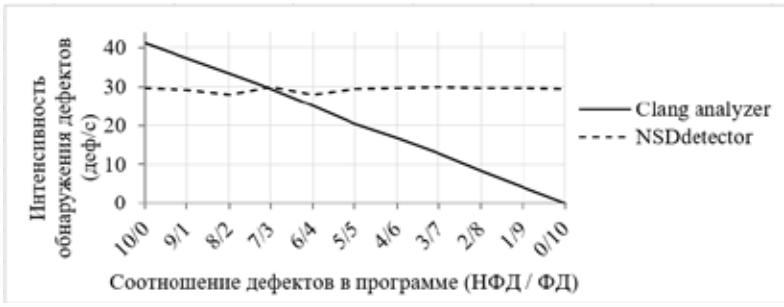


Рис. 4. График зависимости интенсивности обнаружения дефектов от соотношения НФД и ФД в программе для *Clang analyzer* и *NSDdetector*

Fig. 4. Schedule of dependence defects detection rate from the ratio between NFD and FD in software for *Clang analyzer* and *NSDdetector*

Для компилируемого кода *Clang* создает единицу трансляции (*CXTranslationUnit*), содержащую данные о процессе разбора, для которой возможно получить корень АСД (*clang_getTranslationUnitCursor()*) и начать его обход (*clang_visitChildren()*). Проверяя тип курсора (*clang_getCursorKind()*) при каждом шаге обхода, можем организовать вычисление значений ВЕС программных объектов по выражению (5) и проверку корректности операций с ВЕС согласно (6). Такой подход дает возможность по текущему курсору детектировать ДЕС с точностью до строки кода в файле и позиции в операторе.

Для оценки эффективности программной реализации предлагаемой методики воспользуемся показателем в виде интенсивности обнаружения дефектов, то есть количеством выявляемых дефектов в единицу времени: $Э_{А} = \frac{N_{ДЕФ}}{T_{А}}$, где $N_{ДЕФ}$ – количество обнаруживаемых дефектов; $T_{А}$ – время анализа. Логично полагать, что чем выше интенсивность обнаружения, тем эффективнее работает анализатор.

Опыт проводился при одинаковых вычислительных ресурсах на ИК размером 1000 строк исходным *Clang analyzer* и созданным на его основе программным средством (*NSDdetector*). Варьируя соотношением ФД (в данном случае ДЕС) и НФД при неизменном общем их количестве, зафиксировали значения времени анализа и количества обнаруженных дефектов. Результаты опыта представлены на рис. 4.

Анализ полученных результатов показал, что применение *NSDdetector* согласно выбранному критерию является оправданным на начальном этапе создания ПО, когда в ИК широко представлены как НФД, так и ФД. При снижении представительства ФД в коде ниже соотношения 7/3 целесообразно применять исходный *Clang analyzer*.

## Заключение

Полученная методика разработана как альтернатива традиционным способам поиска ФД и позволяет автоматизировать процесс их поиска, что положительно сказывается на времени и объективности проверок. Программная реализация подхода может использоваться как разработчиками ПО АС ВН для раннего обнаружения дефектов, так и орга-

низациями, осуществляющими испытания ПО АС ВН, в качестве диверсной методики испытаний.

**Список литературы**

[1] Лобанова Н.М., Дубровин Д.А. Оценка затрат на качество программного обеспечения. *Вестник университета*, 2016, 6, 87-91 [Lobanova N.M., Dubrovin D.A. Evaluation of software quality costs. *University herald*, 2016, 6, 87-91 (in Russian)]

[2] Кулямин В.В. *Методы верификации программного обеспечения.* М.: Институт системного программирования РАН, 2008. 117 с. [Kuljamin V.V. Methods of software verification. Moscow, Institut sistemnogo programmirovanija RAN, 2008, 117 p. (in Russian)]

[3] Ицыксон В.М., Моисеев М.Ю., Цесько В.А., Захаров А.В., Ахин М.Х. Алгоритм интервального анализа для обнаружения дефектов в исходном коде программ. *Информационно-управляющие системы*, 2009, 2, 34-41 [Itsykson V.M., Moiseev M.Ju., Tsesko V.A., Zakharov A.V., Akhin M.H. Interval analysis algorithm for detecting defects in the software source code. *Information-control systems*, 2009, 2, 34-41 (in Russian)]

[4] Егоров В.В., Томилова Н.И., Амиров А.Ж., Касылкасова К.Н. Методы верификации программного обеспечения. *Молодой ученый*, 2016, 21, 138-141. [Egorov V.V., Tomilova N.I., Amirov A.Zh., Kasylkasova K.N. Software verification methods. *Young scientist*, 2016, 21, 138-141 (in Russian)]

[5] Андреев Г.И., Созинов П.А., Тихомиров В.А. *Управленческие решения при проектировании радиотехнических систем.* М.: Радиотехника, 2018. 560 с. [Andreev G.I., Sozinov P.A., Tikhomirov V.A. *Managerial decisions in the design of radio engineering systems.* Moscow, Radiotehnika, 2018, 560 p. (in Russian)]

[6] Глухих М.И., Ицыксон В.М., Цесько В.А. Использование зависимостей для повышения точности статического анализа программ. *Моделирование и анализ информационных систем*, 2011, 4(1), 68-79 [Glukhikh M.I., Itsykson V.M., Tsesko V.A. The use of dependencies for improving the precision of program static analysis. *Modeling and analysis of information systems*, 2011, 4(1), 68-79 (in Russian)]

[7] Бриджмен П. *Анализ размерностей.* Ижевск.: НИЦ «Регулярная и хаотическая динамика», 2001. 128 с. [Bridzhmen P. *Dimensional analysys.* Izhevs, NITS “Regulyarnaya i khaoticheskaya dinamika”, 2001, 128 p. (in Russian)]

[8] Ахо А., Сети Р., Ульман Дж. *Компиляторы: принципы, технологии и инструменты.* М.: Издательский дом «Вильямс», 2003. 768 с. [Aho A., Seti R., Ul’man Dzh. *Compilers: principles, technologies, tools.* Moscow, Izdatel’skij dom “Vil’yams”, 2003, 768 p. (in Russian)]