Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ *

А.А. Белеванцев <abel@ispras.ru>
Е.А. Велесевич <evel@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В статье рассматривается инструмент статического анализа программ, определяющий сущности программы на языке Си или Си++, их метрики и связи между ними. Сущностями программы являются файлы, функции, классы, методы и т.п., а связями – вызовы, наследование, чтение/запись глобальных переменных, включение, агрегация. Необходимость построения такого инструмента возникает в связи с тем, что для широкого круга задач понимания программ основой для построения решения является автоматическое извлечение необходимой информации о программе из ее исходных кодов. Инструмент должен поддерживать все конструкции языков Си и Си++ и масштабироваться для анализа реальных программных систем в миллионы строк исходного кода.

Статья посвящена методам построения такого инструмента на основе открытой компиляторной инфраструктуры LLVM[1]. Для разбора текстов программ используется промышленный компилятор Clang[2], а для последующего анализа – собственный инструмент на основе LLVM, обеспечивающий консолидацию информации обо всей программе. Так как окончательный анализ выполняется на уровне внутреннего представления (биткода) LLVM, то необходимо обеспечить сохранение в файлах с этим представлением дополнительной информации уровня исходного кода, теряющейся при изначальной трансляции. Для этого, а также для поддержки разнообразных диалектов Си и Си++, было выполнено более 400 доработок компилятора Clang[2]. В анализаторе уровня биткода центральной частью является компоновщик, задачей которого является объединение информации об одних и тех сущностях, участвующих в сборке различных компонент программной системы (например, библиотеки и приложения, ее использующего). Построенные с использованием компоновщика связи между сущностями позволяют точнее отследить отношения между компонентами всей системы. В статье также представляются результаты тестирования инструмента на коде ОС Android.

Ключевые слова: понимание программ; LLVM; статический анализ; метрики исходного кода.

* Работа поддержана грантом РФФИ 14-01-31363 мол_а.

DOI: 10.15514/ISPRAS-2015-27(2)-4

Для цитирования: Белеванцев А.А., Велесевич Е.А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 53-64. DOI: 10.15514/ISPRAS-2015-27(2)-4.

1. Введение

В жизненном цикле больших программных систем, особенно при их длительном использовании, перед разработчиками возникают задачи, которые можно объединить под термином *понимания* программ – получения некоторой информации о программе, тем или иным способом облегчающей работу с ней. Например, требуется доработка системы под новые требования пользователей, однако имеющаяся документация неполна или вообще отсутствует; или же необходимо выполнить рефакторинг кода системы или изменение ее архитектуры для обеспечения возможности дальнейших доработок. Наконец, важной задачей, особенно актуальной из-за развития открытого ПО, является поддержание в актуальном состоянии собственных доработок некоторой крупной открытой программы и перенос этих доработок на новые версии программы (а иногда и на более старые).

Во всех описанных случаях (естественно, в предположении, что доступен исходный код программной системы) решение задачи можно разделить на две части. Во-первых, необходимо собрать информацию о структуре программы – частей, из которых она состоит, их свойств и характеристик, а также об их организации (иерархии и связях). Во-вторых, нужно придумать, как использовать собранную информацию для решения исходной задачи.

Например, в случае использования компонента системы при отсутствующей документации можно собрать информацию об экспортируемых этим компонентом функциях и типах данных, установить способы вызовов этих функций из остальных частей системы, и после ручного анализа сформулировать контракты на использование компонента. Если этого недостаточно, то можно получить информацию об использовании типов данных и функций в исходном коде самого компонента. Для рефакторинга кода, помимо аналогичной информации о функциях и их вызовах другими функциями, использовании функциями глобальных структур данных, может пригодиться статистика сложности отдельных частей исходного кода и их связей: например, если функция вызывает слишком много других, или ее управляющие конструкции имеют большую вложенность, то, возможно, ее нужно разбить на несколько вспомогательных функций. Наконец, для переноса доработок между версиями программы можно взять используемые в доработках функции основной системы, собрать данные об этих функциях в обеих версиях системы и сравнить их, тем самым найти изменившиеся компоненты системы и по их связям с остальным кодом понять, как нужно обновить собственные доработки.

Как видно, все описанные действия ручного анализа структуры программы предполагают возможность автоматической сборки информации о ней. Необходимую информацию можно обобщить следующим образом:

- *сущности* программы: физические – файлы, каталоги – и логические, в зависимости от заданного языка программирования, – функции, классы, поля, переменные, методы;
- *свойства* этих сущностей: имя, прототип (для функций), место определения в программе, родительская сущность, другие различные атрибуты;
- *количественные характеристики* сущностей: размер, количество вложенных подсущностей, количество определенных конструкций в исходном коде;
- *связи* между сущностями: вызовы функций, чтение и запись глобальных переменных и статических полей классов, наследование классов, агрегация (класс содержит поля, файл включает другой файл).

В работе описывается инструмент, предназначенный для сбора этой базовой информации о программе. При наличии такого инструмента, решающего первую часть описанных задач понимания программ, над ним можно строить другие инструменты, автоматизирующие применение этой информации для решения конкретной задачи. В любом случае это применение требует ручного труда программиста. Однако сейчас даже сбор описанной основной информации не производится автоматически. Некоторую ее часть строят современные среды разработки, но не полностью; при этом для программных систем в миллионы строк кода среды разработки не масштабируются – можно исследовать исходный код системы только покомпонентно, а связи между компонентами будут утеряны. Специализированных открытых инструментов анализа подобного рода нет, но на рынке есть ряд коммерческих продуктов – системы Klocwork Architect, Understand[3], Imagix4D[4]. Насколько можно установить по доступной информации об этих системах, их возможности и поддерживаемые языки пересекаются не полностью, т.е. нет одной системы, которая полностью покрывает потребности разработчиков.

Предлагаемый нами инструмент поддерживает языки Си и Си++, а в будущем и язык Java, и анализирует программы для ОС Linux объемом в миллионы строк кода (например, полный исходный код ОС Android). При разработке инструмента были максимально использованы существующие открытые программные продукты – компилятор Clang, инфраструктура LLVM, компоновщик и архиватор из пакета GNU Binutils [7]. Также были использованы компоненты инструмента статического анализа Svace[5-6], разработка которого ведется в ИСП РАН. Это позволило создать прототип инструмента в сжатые сроки.

Далее в разделе 2 статьи описывается архитектура инструмента и его компоненты, в разделе 3 предлагается список вычисляемых метрик и связей и особенности необходимых для этого вычисления алгоритмов анализа, в разделе 4 содержатся некоторые экспериментальные результаты. Раздел 5 завершает статью.

2. Архитектура инструмента анализа

Для разработки описанного инструмента анализа наилучшим образом подходит инфраструктура компилятора. Поиск сущностей программы и связей между ними требует разбора программы, который хорошо выполняет компилятор, и применения анализа потока управления и потока данных, которые, как правило, уже используются в оптимизационных проходах компилятора. В настоящее время единственными промышленными компиляторами с открытым исходным кодом для языков Си и Си++ являются компиляторы GCC и Clang/LLVM. Оба компилятора поддерживают последние версии стандартов языков, содержат удобное внутреннее представление для анализа и необходимые средства. Тем не менее, для нашего анализатора удобнее система LLVM, т.к. требуется сохранять вместе с промежуточным представлением программы собственную информацию о сущностях и метриках, а в LLVM для этого есть стандартный расширяемый формат – т.н. *метаданные*, которые могут создаваться компилятором, сохраняться на диск и впоследствии считываться анализатором.

Инструмент состоит из трех частей, каждая из которых отвечает за один из трех этапов его работы. На первом этапе требуется организовать автоматическое построение необходимого внутреннего представления программы – многократный запуск компилятора вручную для каждого исходного файла неудобен пользователю. Для автоматического создания представления программы нужно знать, как организована её сборка (т.е. как именно и для каких исходных файлов запускаются компиляторы, компоновщики и другие инструменты, чтобы получить искомое приложение). Тогда для каждого исходного запуска компилятора можно параллельно запустить собственный компилятор, который сгенерирует необходимые данные. Чтобы не зависеть от конкретной системы сборки, такая задача решается универсальным *мониторингом сборки*: организуется перехват запуска всех процессов операционной системы; при запуске анализируется командная строка процесса и определяется, имеет ли запущенный процесс отношение к сборке программы (т.е. является ли он компилятором, компоновщиком или подобной утилитой); если процесс успешно распознан, то выполняются действия по запуску собственного компилятора или компоновщика. Мониторинг процесса сборки является единственным существенно зависящим от операционной системы компонентом: реализация перехвата запуска процессов различна в ОС семейства Linux и Windows. В первом случае используется динамическая библиотека, загружаемая до старта

программы (т.н. механизм LD_PRELOAD), во втором случае – запуск процессов под отладкой.

Действия по запуску своего компилятора зависят от типа перехваченного компилятора: требуется фильтрация исходной командной строки для удаления несовместимых опций компилятора, преобразования одинаковых опций с разным написанием из исходного формата в собственный, преобразования пути выходного файла в собственный, передача стандартных путей поиска включаемых файлов исходного компилятора собственному и т.п. Общая цель этих изменений – добиться того, чтобы свой компилятор максимально точно соответствовал настройкам исходного, чтобы собрать именно то приложение, которое подается на вход исходной системе сборки. Из-за гибкости и широких возможностей современных компиляторов полностью повторить одним собственным компилятором несколько других во всех случаях невозможно, и для корректной сборки программ большого объема требуются существенные усилия. Наш анализатор использует мониторинг сборки инструмента анализа Svace, разрабатываемого в ИСП РАН, что минимизирует необходимые доработки.

На втором этапе запускается собственный компилятор на базе Clang, который строит искомое внутреннее представление программы. Наш компилятор существенно модифицирован по сравнению с исходным – выполнено более 300 изменений. Во-первых, это изменения, поддерживающие совместимость с другими популярными компиляторами – либо требуется реализовать расширения других компиляторов в собственном, либо снизить требования к соответствию компилируемой программы стандарту языка. Во-вторых, изменения, требующиеся для вычисления метрик исходного кода, которые можно выполнить только в компиляторе. Такие изменения включают в себя, например, модификацию лексического и синтаксического разбора программы для учета строк с комментариями и без комментариев, пробельных строк и т.п. Наконец, последней частью изменений является код для создания собственных данных (в формате метаданных инфраструктуры LLVM), которые хранят информацию, необходимую для анализатора скомпонованных приложений во внутреннем представлении LLVM (т.н. *биткоде*). Частью этой информации является стандартная отладочная информация, расширенная для нужд анализатора, уже вычисленные метрики и отношения, дополнительная информация о вызовах (например, неявные вызовы) и т.п. Основным принципом являлось максимальное использование имеющейся отладочной информации, и добавления новых метаданных были минимальными. Однако из-за объема собранных данных итоговый размер файлов с биткодами мог превышать первоначальный в 3-4 раза.

Окончанием второго этапа работы инструмента является компоновка сгенерированных файлов с биткодом вместе так же, как объектные файлы компонуются в приложения и библиотеки. Это делается, чтобы анализатору на третьем этапе предъявлялись уже скомпонованные файлы со связями между

символами программы, полностью соответствующими исходным. За компоновку отвечает отдельный модуль анализатора, управляющий ее ходом согласно собранной на первом этапе трассе сборки, где отмечены все запуски компиляторов, ассемблеров, компоновщиков и архиваторов. При этом поддерживаются правильные отображения файлов с исходным кодом, ассемблером, объектным кодом, библиотеками и приложения друг в друга. Компоновка биткода может вестись параллельно, зависимости между необходимыми запусками компоновщика автоматически вычисляются управляющим модулем по трассе сборки.

Например, пусть в исходной программы файлы `a.o` и `b.o` (полученные из исходных файлов `a.c` и `b.c`) компонуются в библиотеку `lib.so`, а потом эта библиотека компонуется с файлом `main.o` (полученного из файла `main.cpp`) в исполняемый файл приложения `main`. Тогда свой компилятор должен обеспечить компиляцию файлов `a.c`, `b.c` и `main.cpp` в файлы `a.o.bc`, `b.o.bc` и `main.o.bc` соответственно, а компоновщик должен скомпоновать файлы `a.o.bc` и `b.o.bc` в файл `lib.so.bc`, а потом этот файл и `main.o.bc` – в файл `main.bc` (файлы с биткодом LLVM традиционно имеют расширение `.bc`). Полученные файлы библиотеки и приложения в формате LLVM будут переданы анализатору.

Наконец, третья компонента инструмента представляет из себя собственно анализатор файлов с биткодом LLVM. На вход анализатору подается список всех скомпонованных в ходе сборки проекта файлов (библиотек или исполняемых файлов). Анализатор по очереди считывает файлы и метаданные в них стандартными средствами LLVM. После этого для всех упомянутых в файле сущностей создается их представление в памяти, сохраняются метрики и проводятся связи необходимых типов.

Для корректного представления информации о сущностях (в особенности о месте определения сущности в файле) анализатор должен уметь объединять информацию об одной и той же сущности, представленную в разных файлах. Например, при анализе приложения известно, что функция приложения `foo` вызывает функцию `bar`, объявленную в файле `lib.h`, определение функции `bar` недоступно. Если код библиотеки `lib` также недоступен, то более точной информации о функции `bar` получить не удастся, иначе при анализе библиотеки `lib` нужно уметь определять, что функция `bar` из файла `bar.c`, объявленная в файле `lib.h`, – это та же самая функция, что уже была встречена при анализе функции `foo` из основного приложения, то есть одна сущность. Ее свойства, полученные при анализе приложения и библиотеки, нужно объединить вместе. Эта функциональность анализатора аналогична компоновщику, но не в рамках одного приложения, а в рамках программной системы в целом, и является центральной частью анализатора, благодаря которой можно получить корректную информацию обо всей системе со связями между ее компонентами.

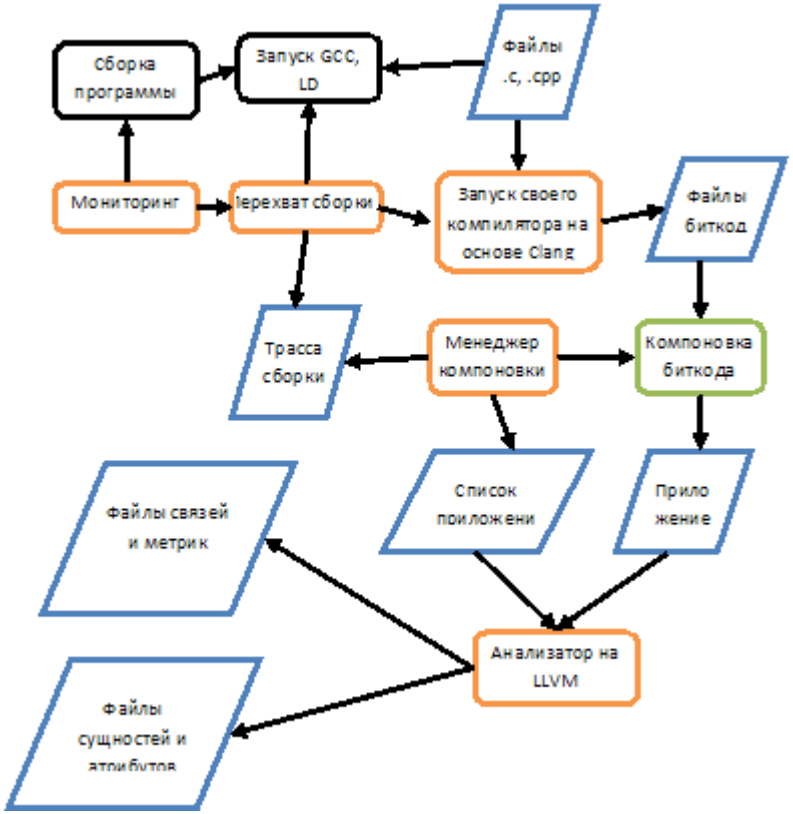

Рис. 1. Архитектура и схема работы предложенного анализатора

Схема работы описываемого инструмента и его архитектура представлены на рисунке 1. Черными прямоугольниками выделены процессы исходной сборки, оранжевым – компоненты анализатора, синим – обрабатываемые данные. Зеленым прямоугольником отмечен используемый компоновщик Gold из пакета GNU Binutils, никак не модифицированный в ходе работы над инструментом (настраивались лишь параметры компоновщика).

3. Сущности программы, их метрики и связи между ними

Поддерживаемыми инструментом сущностями являются: физическими – файл, каталог, строка в файле (при указании места возникновения связи или определения сущности); логическими – глобальные переменные, поля и методы, классы, структуры, функции, макросы, перечислимые и другие сложные типы. Атрибутами сущностей является их место определения, родительская сущность и специфическая информация – прототип для функции, модификаторы доступа для полей, методов и классов и т.п.

Метрики, вычисляемые инструментом, можно классифицировать следующим образом. Во-первых, это метрики физического и логического размеров сущностей (количество строк в функции или файле с учетом/без учета пробелов и комментариев, количество полей/методов в классе, средний/максимальный размер функции в файле). Во-вторых, это метрики сложности сущности (цикломатическая сложность, уровень вложенности управляющих конструкций). Наконец, есть метрики, характеризующие связи между сущностями (количество вызовов данной функции и вызовов других функций из данной). Всего реализовано более 30 различных метрик. Некоторые из них просто агрегируют соответствующие метрики по всем сыновним подсущностям – так, есть метрика среднего и максимального размеров как метода, так и класса; как функции, так и файла и каталога.

Можно заметить, что все метрики так или иначе отражают «сложность» сущности, и для каждой из них можно эмпирически установить некоторый порог, при превышении которого требуется анализ сущности и по возможности ее упрощение или декомпозиция на несколько других сущностей. Наше исследование не ставит себе целью определение таких порогов для каждой метрики, так как эти значения могут отличаться для разных разработчиков и проектов; анализ порогов сложности может выполняться на втором этапе решения задачи понимания программы – при постобработке собранных нашим инструментом данных.

Основными характеристиками сущностей программы являются не метрики, а связи между сущностями. Для функций и методов такой связью является вызов других функций (возможно – неявный вызов, например, конструктора объекта при входе в локальный блок, где объявлена переменная объекта), для глобальных переменных – их чтение или запись. Для построения иерархии сущностей определяется связь «сущность содержит сущность» (класс содержит метод, файл содержит глобальную переменную). Кроме того, для Си++ возникают особые типы связей, характеризующие иерархию классов – связи по наследованию классов и переопределению методов.

Наиболее сложный анализ (потоков управления и данных) требуется как раз при определении связей между сущностями. Для определения вызовов между функциями строится граф вызовов, в том числе с учетом вызовов по указателю. Для Си++ и в будущем для Java важную роль играет девиртуализация вызовов, позволяющая получить более точные данные о вызываемой функции. В будущем планируется использовать в инструменте независимо разработанный в ИСП РАН алгоритм девиртуализации для инфраструктуры LLVM.

Для определения вызовов функций из динамических библиотек (например, с помощью интерфейса `dlopen` ОС Linux) требуется минимальный анализ потока данных и анализ указателей для установления связи между указателем на функцию и переменной, содержащей имя вызываемой функции. Это

позволяет, например, поддержать случаи, когда имеется константный массив имен функций, которому в соответствие ставится массив указателей на функции динамической библиотеки. При этом при вызове функции по некоторому указателю из массива известно ее имя и название библиотеки, полученной через анализ вызова функции `dlopen`.

Наконец, для анализа иерархии сущностей строится граф включения исходных файлов друг в друга и граф наследования между классами программы. Граф наследования используется также для поиска связей переопределения методов и для работы алгоритма девиртуализации.

Анализ остальных связей и метрик, как правило, не доставляет большого труда. Основной задачей анализатора здесь является, как уже было указано, правильное определение эквивалентности сущностей из разных приложений и соответственное объединение их атрибутов, метрик и связей. Кроме этого, анализатор агрегирует составные метрики сущностей из метрик подсущностей (например, среднюю цикломатическую сложность класса из цикломатических сложностей его методов).

4. Экспериментальные результаты

Разработанный инструмент был опробован нами на исходном коде ряда открытых проектов, включая код ОС Android, ядра ОС Linux. Наиболее затратным по времени этапом является компоновка файлов с внутренним представлением, т.к. размер созданных метаданных большой, а инфраструктура LLVM плохо сливает одинаковую информацию из разных файлов (в том числе отладочную). Мы планируем выполнить исследования по разработке алгоритмов лучшего слияния такой информации. Фаза компиляции внутреннего представления и собственно анализа занимает время, сравнимое со временем сборки исходного кода проекта. Точность и полнота получаемых данных составляет свыше 90% по сравнению с доступными коммерческими аналогами. В таблице 1 приведены результаты точности и полноты для некоторых связей для подмножества кода из ОС Android версии 4.4.2.

Табл. 1. Точность построения связей для кода из ОС Android

Тип связи	Точность	Полнота
IMPLICITLY_CALLS	94.70%	85.15%
CALLS	99.48%	88.67%
READS	96.62%	91.71%
WRITES	99.63%	88.91%

INCLUDES	100.00%	99.89%
INHERITS	99.52%	87.38%
OVERRIDES	98.75%	92.96%
Среднее значение	**98.29%**	**90.57%**

Необходимо отметить, что быстрое прототипирование инструмента стало возможно благодаря широкому использованию открытых компонентов проекта LLVM и GNU Binutils[7]. Наши модификации компилятора Clang были сосредоточены только на необходимых нам свойствах, а усилия для полной поддержки языков Си и Си++ тратить не требовалось. Метаданные сохранялась нами также в стандартном формате LLVM, что позволило считывать их имеющимися компонентами инфраструктуры. Наконец, анализ вызовов и анализ указателей могут также использовать имеющиеся компоненты.

5. Заключение

В статье описан разработанный инструмент анализа программ для их понимания, демонстрирующий достаточно точное построение связей между сущностями программы и метрик сущностей на промышленном исходном коде на языках Си и Си++. В дальнейшем предполагается добавить поддержку языка Java и поставить задачу поиска связей между частями программы, написанными на разных языках – например, поиск вызовов функций на языке Си из кода на языке Java через механизм JNI.

Список литературы

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/
[2]. Clang compiler. http://clang.llvm.org
[3]. Инструмент Understand. https://scitools.com/
[4]. Инструмент Imagix4D. http://www.imagix.com/products/source-code-analysis.html
[5]. А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды Института системного программирования РАН, том 21, 2011 г, стр. 23-38.
[6]. Иванников, В. П., Белеванцев, А. А., Бородин, А. Е., Игнатьев, В. Н., Журихин, Д. М., Аветисян, А.И, Леонов, М. И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН, том 26, выпуск 1, 2014 г., стр. 231-250.

[7]. Пакет GNU Binutils. http://www.gnu.org/software/binutils/

Analyzing C/C++ Code Entities and Relations for Program Understanding*

A. Belevantsev <abel@ispras.ru>
E. Velesevich <evel@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Abstract. This paper describes the static analysis tool for finding program entities, their metrics, and relations between entities. Program entities are files/directories (physical structure) and classes/functions/methods/global variables (logical structure). Relations are connections between entities, such as calls, inheritance, aggregation, reading/writing, inclusion. The need for constructing such a tool arises from the program understanding problems. The basis of these problems' solutions should be automatic extraction of necessary data from program source code. The tool implementing this extraction should support all C/C++ constructs and should scale to millions lines of code to be applicable to real life applications.

In the paper we concentrate on the methods for developing such a tool for C/C++ languages based on open source components: LLVM/Clang compiler infrastructure, GNU Binutils linker and archiver. The final whole-program analysis works on the LLVM internal representation (bitcode) level, so it is necessary to save the additional source level data in the bitcode files that is usually lost during compilation. We have made more than 400 patches to the Clang compiler in order to support this additional data storing and also to support many varieties of C/C++ dialects. For the main analyzer, the central component is a kind of linker that unifies the collected data for the same entities used in different program components (like an application and a library it uses). The correct entities unification performed by the linker allows to trace component relations more precisely.

Finally, we briefly present the results of testing our tool on Android OS.

Keywords: program understanding; LLVM compiler; static analysis; code metrics.

DOI: 10.15514/ISPRAS-2015-27(2)-4

For citation: Belevantsev A.A., Velesevich E.A. Analyzing C/C++ Code Entities and Relations for Program Understanding. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 53-64 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-4.

References

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/
[2]. Clang compiler. http://clang.llvm.org

[3]. The Understand tool. https://scitools.com/
[4]. The Imagix4D tool. http://www.imagix.com/products/source-code-analysis.html
[5]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode program [The usage of static analysis for searching vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, 2011. pp. 23-38 (in Russian)
[6]. V. Ivannikov, A. Belevantsev, A. Borodin, V. Ignatiev, D. Zhurikhin, A. Avetisyan, M. Leonov. Staticheskij analizator Svace dlya poiska defektov v iskhodnom kode program [Svace: static analyzer for detecting of defects in program source code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol 26, issue 1, 2014, pp. 231-250 (in Russian)
[7]. The GNU Binutils package. http://www.gnu.org/software/binutils/