УДК 004.05

В. В. Бураков

# МОДЕЛИРОВАНИЕ И ИДЕНТИФИКАЦИЯ ДЕФЕКТОВ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММНОГО КОДА

Представлен подход к моделированию и идентификации дефектов программного кода для улучшения качества программного обеспечения. Подход базируется на графовом моделировании исходного кода приложения и его комплексном анализе.

***Ключевые слова:*** *моделирование программ, графовая модель, дефект кода.*

**Введение.** При создании программного обеспечения бортовых комплексов широко используется объектно-ориентированное визуальное моделирование — бурно развивающаяся в настоящее время область компьютерной инженерии. В начале 1990-х гг. по этой теме появилось много фундаментальных работ. Наибольшее влияние на формирование этой области оказали исследования Г. Буча, И. Джакобсона, Д. Рамбо, П. Коуда, Д. Харела, Б. Селика и др., усилиями которых был создан стандарт в этой отрасли — язык унифицированного моделирования UML (Unified Modeling Language) [1].

На сегодняшний день UML является широко используемым средством для проектирования программных комплексов любой сложности. Несмотря на это, UML имеет ряд существенных недостатков.

— *Неточная семантика*. Так как UML определяется комбинацией собственных атрибутов (абстрактный синтаксис), языка объектных ограничений — OCL (Object Constraint Language) (формальная проверка правильности) и естественного английского языка (подробная семантика), то он лишен согласованности, присущей языкам, точно определенным техниками формального описания. В некоторых случаях абстрактный синтаксис UML, OCL и английский язык противоречат друг другу, в других случаях — не полностью соответствуют друг другу. Неточность описания самого UML одинаково отражается и на пользователях, и на разработчиках программных продуктов, что приводит к несовместимости инструментов из-за уникальной интерпретации спецификаций.

— *Отсутствие полноты по Тьюрингу*.

— *Кумулятивная нагрузка/рассогласование нагрузки*. Как и в любой системе обозначений, UML позволяет описывать одни системы более кратко и эффективно, чем другие. Таким образом, разработчик склоняется к решениям, которые обеспечивают сочетание „сильных“ сторон UML и языков программирования. Проблема становится более очевидной, если язык разработки не соответствует традиционным принципам.

Концепции UML не позволяют смоделировать готовую версию программного продукта, поэтому достаточно сложно выявить ошибки в архитектуре приложения на стадии проектирования. Помимо этого, в течение жизненного цикла программного продукта необходимо учитывать накладные расходы к общей сумме трудозатрат на разработку, связанные с необходимостью поддерживать модель в согласованном с кодом состоянии. Эти факторы могут крайне негативно повлиять на надежность программного обеспечения, что недопустимо в сферах, где от работы аппаратно-программного комплекса может зависеть здоровье и жизнь человека. Учитывая тенденцию всесторонней компьютеризации жизнедеятельности человека, надежность кода можно считать одним из основополагающих критериев качества. Из-за большого объема и высокого уровня сложности программного кода наиболее эффективным

способом выявления дефектов в системах подобного класса является автоматизированный поиск.

В этой связи особую актуальность приобретает возможность строгого и согласованного в математическом смысле моделирования структуры и поведения разрабатываемых программных систем, а также создания моделей и алгоритмов для поиска дефектов архитектуры на любом этапе жизненного цикла программного обеспечения. Предлагаемый в настоящей статье подход основан на использовании теории графов для представления программного кода и включает набор методов и алгоритмов идентификации дефектов.

**Графовая модель кода.** Основой предлагаемой концепции является моделирование программного кода с помощью ориентированного помеченного типизированного графа. Помимо основного графа, представляющего исходный код программы, используется множество подграфов, описывающих дефекты и условия их появления. Для более полного описания условий идентификации дефектов программного кода введены допустимые и недопустимые графы [2].

Определение 1. *Ориентированный граф* $G = (V, E, s, t)$ состоит из двух множеств — конечного множества $V$, элементы которого называются вершинами, и конечного множества $E$, элементы которого называются дугами. Каждая дуга связана с упорядоченной парой вершин. Для обозначения вершин используются символы $v_1, v_2, v_3\ldots$, а для обозначения дуг — символы $e_1, e_2, e_3\ldots$ Если $e_1 = (v_i, v_j)$, то $v_i$ — начальная вершина дуги $e_1$, а $v_j$ — ее конечная вершина. Все дуги, имеющие одну пару начальных и конечных вершин, называются параллельными. Функции $s: E \to V$ и $t: E \to V$ связывают с каждой дугой одну начальную и одну конечную вершины.

Определение 2. *Помеченный граф.* Пусть $L=(VL, EL)$, $A=(VA)$ — пара непересекающихся потенциально бесконечных множеств меток и ролей соответственно. $(L, A)$-помеченный граф $G$ представляет собой тройку $(g, l, a)$, такую что: $g=(V, E, s, t)$ — граф; $l=(vl: V \to VL, el: E \to EL)$ — заданная пара функций соответственно вершин и дуг, при этом функция $vl$ является инъективной; $a=(va: V \to VA)$ — функция отображения вершин на множество ролей.

Определение 3. *Помеченный типизированный граф.* Пусть $T=(VT, ET)$ — пара непересекающихся конечных множеств предопределенных типов вершин и дуг. $(L, A)$-помеченный $T$-типизированный граф $G$ представляет собой двойку $(g, type)$, такую что: $g$ — $(L, A)$-помеченный граф; $type=(vt: V \to VT, et: E \to ET)$ — пара функций, где функция $vt$ связывает с каждой вершиной из множества $V$ ее тип из множества $VT$, а функция $et$ связывает с каждой дугой из множества $E$ ее тип из множества $ET$.

Определение 4. *Подграф.* $H$ является подграфом $G$ (обозначается $H \subseteq G$), если существует инъективный графовый морфизм $m: H \to G$, называемый соответствием $H$ в $G$.

Определение 5. *Допустимый (недопустимый) граф.* Пусть $T=(VT, ET)$ — пара непересекающихся конечных множеств типов вершин и дуг, а $TT$ — $T$-помеченный граф, называемый допустимым (недопустимым) графом. $(L, A)$-помеченный $TT$-типизированный граф представляет собой двойку $(G, tt)$, такую что: $G$ — $(L, A)$-помеченный граф; $tt: G \to TT$ — тотальный LGraph-морфизм. Помеченный $TT$-типизированный графовый морфизм представляет собой графовый морфизм вида $f: G \to H$ между $(L, A)$-помеченными $TT$-типизированными графами. $TT$-сохраняющий помеченный типизированный графовый морфизм представляет собой помеченный $TT$-типизированный графовый морфизм вида $f: G \to H$, такой что $tt_H: f=tt_G$ для $\forall x \in \mathrm{dom}(f)$.

На рис. 1 показан пример представления исходного кода программы на языке C++ в виде графовой модели.

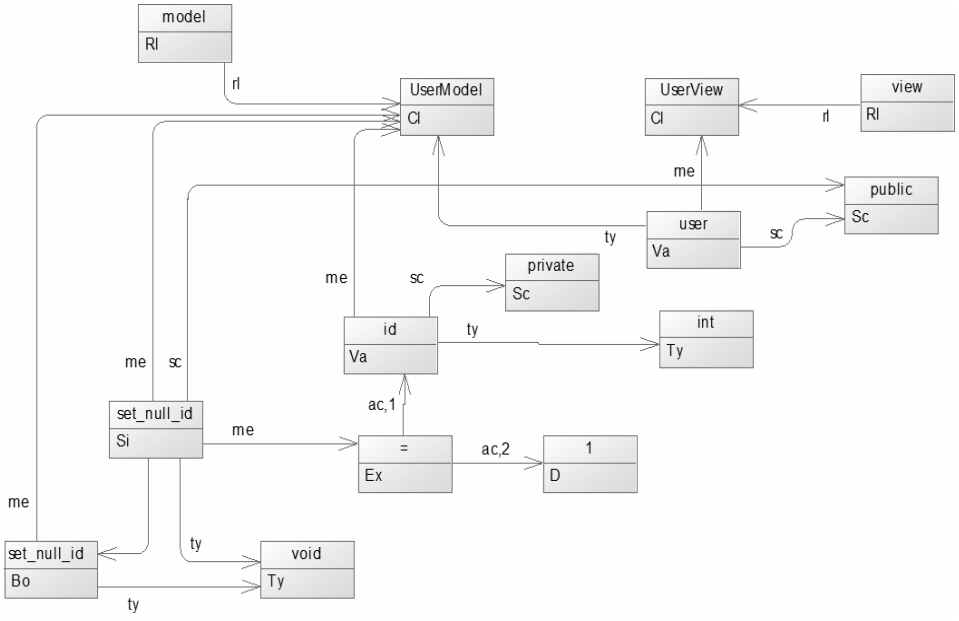


*Рис. 1*

Ниже представлен исходный код на языке C++ для данного графа:

```
class UserView {//role = view
  public: UserModel user;};
class UserModel{ //role = model
                private:
                        int id;
        public:
                        void set_null_id(); };
public: void UserModel::set_null_id(){
                         id = 0;}
```

**Спецификация и поиск дефектов.** В основе предлагаемого подхода лежит представление программного кода и описание дефекта в виде графовой модели. В простейшем случае задача выявления дефекта сводится к поиску подграфа в графе. В терминах графовой модели дефектом может являться как факт нахождения подграфа в графе, так и отсутствие искомого подграфа.

Помимо поиска подграфа в графе, реализация предлагаемого подхода предусматривает:

— распознавание дефектов на основе метрического анализа;

— использование ролей для специализации контекста разрабатываемых программных сущностей и поиска дефектных программных решений;

— спецификацию условий и поиск программных сущностей, нарушающих эти условия.

Конструктивно эти варианты поиска дефектов оформлены в виде плагинов, которые можно комбинировать в зависимости от конкретных задач анализа кода.

***Метрический анализ.*** Суть метрического анализа заключается в подсчете количественных характеристик кода. По умолчанию могут быть определены основные базовые метрики:

— количество входных вершин вершины;

— количество выходных вершин вершины;

— количество входных вершин для вершины с дугами определенного типа;

— количество выходных вершин для вершины с дугами определенного типа;

— длина нисходящего пути вершины из дуг определенного типа.

На основе базовых метрик пользователь может сформировать производные метрики. Каждая производная метрика определяется путем введения функциональной зависимости от

*n* (*n*>0) других как базовых, так и производных метрик. Основными видами функциональных зависимостей, порождающих производные метрики, являются:

— сумма *n* базовых (производных) метрик;

— частное от деления одной базовой (производной) метрики на другую;

— максимальное значение базовой (производной) метрики;

— минимальное значение базовой (производной) метрики;

— среднее (арифметическое, геометрическое и т.п.) значение базовых (производных) метрик.

Для каждой метрики возможно задать граничные значения, что позволяет более гибко настраивать систему под различные проекты.

***Роли и контекст.*** Каждому классу в проекте присваивается роль (или роли), определяющая контекст использования экземпляров этого класса другими программными элементами. Также задаются правила отношений между классами с определенными ролями. В каждом правиле есть возможность указать как разрешенные связи, так и запрещенные. На основе заданных правил для идентификации дефекта производится поиск запрещенных связей между сущностями программного кода. Например, пусть необходимо выявить корректность реализации архитектуры „модель—представление—контроллер“, которая заключается в отделении логики модели от логики представления кода и контроллера, т.е. необходимо исключить прямые обращения из класса модели в классы представления и контроллера. На рис. 2 показаны графы, моделирующие эти дефекты. Суть идентификации дефекта заключается в нахождении подграфов в графе исходного кода.

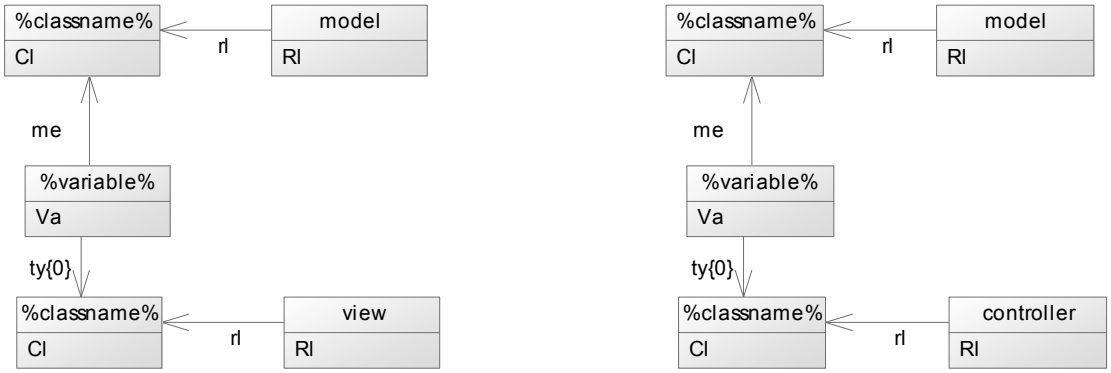


*Рис. 2*

***Условия.*** В практике программирования относительно редко встречаются случаи, когда какая-либо часть программного кода является сама по себе дефектом. Как правило, для этого необходимо выполнение одного или нескольких условий.

Каждый элемент списка условий — есть граф или набор графов, представляющих некое условие. Каждый элемент имеет свой порядковый номер, необходимый для случаев, когда важен порядок выполнения условий. Если порядок выполнения условий важен только для нескольких элементов списка, то порядковые номера задаются только для них. Помимо этого, каждому элементу присваивается признак, содержащий информацию о том, является данное условие обязательным или нет.

Граф, описывающий некое условие, также может быть допустимым или недопустимым. Для выявления некоторого дефекта необходимо, чтобы все обязательные условия были выполнены. Необязательные условия являются уточнениями к выявлению дефекта, так как не всегда могут существовать в исходном коде.

***Практическая апробация.*** Для практической оценки эффективности предлагаемого подхода была разработана программная система. В качестве платформы для ее реализации выбрана среда разработки MS Visual Studio. Компоненты, реализующие вышеуказанные

этапы процессов моделирования и поиска дефектов, представляют собой набор программных модулей, включенных в плагин к этой среде (рис. 3).

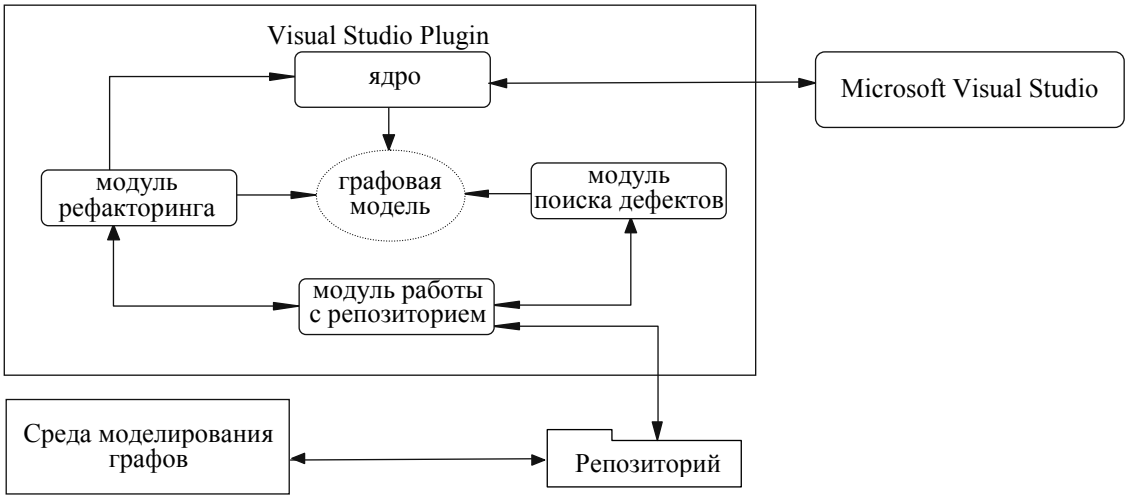


*Рис. 3*

Среда взаимодействует непосредственно с модулем „ядро“, который позволяет работать с объектной моделью открытого проекта. Большая часть модуля „ядро“ представляет собой парсер, преобразующий код проекта в графовую модель. Непосредственно в сам плагин не входят репозиторий дефектов и рефакторингов и среда моделирования графов. Репозиторий представляет собой совокупность каталогов на диске, содержащих XML-файлы с описанием того или иного дефекта или рефакторинга, описанного в виде графа. Модуль поиска дефектов, получив описание дефектов из репозитория, производит поиск в графовой модели, сгенерированной при помощи модуля „ядро“.

Основные возможности разработанной системы:

— математически точное описание структуры и поведения классов;

— математическое моделирование дефектов;

— возможность ведения и пополнения базы дефектов;

— использование разных стратегий для поиска дефектов, выбор наиболее подходящей из них, возможность комбинирования стратегий;

— гибкая настройка системы поиска;

— автоматизированный поиск дефектов.

**Заключение.** В настоящее время осуществляется опытная эксплуатация разработанной системы моделирования и поиска программных дефектов и формирование статистической информации, на основе которой можно будет оценить эффективность подхода.

Предложенный подход позволит автоматизировать процессы выявления дефектов в архитектуре программных приложений и обеспечить повышение их качества и снижение затрат на разработку и сопровождение.

Исследования, выполненные по данной тематике, проводились при финансовой поддержке ведущих университетов Российской Федерации: Санкт-Петербургского государственного политехнического университета (мероприятие 6.1.1), Университета ИТМО (субсидия 074–U01), Программы научно-технического сотрудничества Союзного государства „Мониторинг СГ“ (проект 1.4.1–1), Российского фонда фундаментальных исследований (гранты № 12-07-00302, 13-07-00279, 13-08-00702, 13-08-01250, 13-07-12120, 13-06-0087), Программы фундаментальных исследований ОНИТ РАН (проект № 2.11), проектов ESTLATRUS 2.1/ELRI-184/2011/14, 1.2/ELRI-121/2011/13.

СПИСОК ЛИТЕРАТУРЫ

1. OMG UML Version 2.3 [Электронный ресурс]: <http://www.omg.org/spec/UML/2.3/>, 2010.

2. *Бураков В. В.* Управление качеством программных средств. СПб: СПбГУАП, 2009. 287 с.

***Сведения об авторе***

***Вадим Витальевич Бураков*** — д-р техн. наук, профессор; СПИИРАН, лаборатория информационных технологий в системном анализе и моделировании; E-mail: Burakov@eureca.ru

Рекомендована СПИИРАН

Поступила в редакцию 10.06.14 г.

УДК 681.3.062

Л. Н. ФЕДОРЧЕНКО

# МЕТОД РЕГУЛЯРИЗАЦИИ ГРАММАТИК В СИСТЕМАХ ПОСТРОЕНИЯ ЯЗЫКОВЫХ ПРОЦЕССОРОВ

Описывается алгоритм регуляризации приведенных контекстно-свободных грамматик, основанный на эквивалентных преобразованиях, который совместно с алгоритмом устранения рекурсий редуцирует грамматику к единственному регулярному выражению.

***Ключевые слова:*** *контекстно-свободная грамматика, эквивалентное преобразование грамматики.*

Регулярные множества и контекстно-свободные языки с различными ограничениями и расширениями успешно применяются в технологии построения трансляторов. При создании разного вида трансляторов языков программирования используется множество технологических средств построения анализаторов формальных языков. Как правило, эти технологические средства обеспечивают лишь проверку предъявляемых к грамматике требований и выдачу диагностических сообщений об их нарушениях. Для получения эквивалентной грамматики, удовлетворяющей условиям алгоритма анализа, существуют способы эквивалентных преобразований контекстно-свободных (КС) грамматик. Эти преобразования могут быть выполнены автоматически. Такие эквивалентные преобразования реализованы в программном средстве SynGT (Syntax Graph Transformations), разработанном в СПИИРАН.

В настоящей статье рассматривается алгоритм редукции приведенной КС-грамматики к одному регулярному выражению с помощью эквивалентных преобразований.

Определим отношение $R$ зависимости между нетерминалами КС-грамматики следующим образом.

О п р е д е л е н и е 1. Пусть $G = (V_N, V_T, P, S)$ — КС-грамматика, где $V_N$ — алфавит нетерминалов, $V_T$ — алфавит терминалов, $P$ — множество правил грамматики, $S \in V_N$ — начальный символ грамматики. Будем считать, что нетерминал $A \in V_N$ зависит от нетерминала $B \in V_N$, если существует правило вида $A \to \alpha B \beta \in P$, где $\alpha,\ \beta \in V^*,\ V = V_N \cup V_T$. Этот факт будем записывать как $(A, B) \in \mathbb{D}$, а множество всех таких пар $\mathbb{D}$ будем называть *отношением зависимости между нетерминалами* КС-грамматики. Другими словами, $\mathbb{D} \subseteq V_N \times V_N$. При $A = B$ считаем нетерминал $A$ *самозависимым (рекурсивным).*