УДК: 004.4'22

Представление и анализ архитектуры программных систем на основе графа взаимодействий

И.В. Игнацкая

Аннотация

В статье предложен разработанный автором обобщенный подход к моделированию архитектуры программных систем на основе графа взаимодействий. Данный подход позволяет построить модель в заданной нотации и обеспечивает простую интеграцию различных представлений. Исследованы возможности анализа полученных моделей. Рассматриваются вопросы формализации, классификации, обоснования и применения методик анализа моделей на основе графа взаимодействий.

Ключевые слова

Моделирование программных систем; граф взаимодействий; case.

1. Введение

На сегодняшний день существует множество методов и нотаций представления программных систем. Они предназначены для описания систем вообще, программных систем или каких-либо их аспектов. Отдельные нотации ориентированы на работу в рамках определенных технологий, инструментов и предметных областей [12,10]. Некоторые фирмы, занимающиеся производством программных систем, разрабатывают свои собственные методологии, которые наиболее полно соответствуют их деятельности [1,4]. На этом фоне на первый план выходят вопросы универсального подхода к моделированию, совместного использования различных моделей. Эти вопросы особенно актуальны для разработчиков крупных программных систем, ориентированных на длительное сопровождение. Здесь моделирование используется в борьбе со сложностью программы. При этом из-за больших

объемов информации возрастает цена внедрения новых методик моделирования и поддержания информации в целостном актуальном состоянии.

Универсальные методики моделирования известны уже не одно десятилетие. Однако, во-первых, универсальность всюду понимается по-разному. Как правило, это означает возможность смоделировать любую программную систему, любой ее аспект. Во-вторых, вопрос о том, какие нотации войдут в состав инструмента, решает производитель. Возможности адаптировать инструментарий под конкретный проект или технологию ограничены. Такая ситуация затрудняет анализ программной системы. Кроме того, если появляются новые возможности языков программирования, новые технологии моделирования, инструментарий становится неадекватен.

В работе предлагается разработанный автором принцип моделирования архитектуры, который подходит для любых программных систем вне зависимости от технологий, методов и инструментов, используемых при их создании. Данное представление позволяет построить модель программной системы практически в любой нотации, обеспечить расширяемость представления, интеграцию нескольких методологий и обобщить методики анализа программных систем. Предлагаемый подход ориентирован на сопровождение программной системы.

2. Моделирование архитектуры программных систем на основе графа взаимодействий

Различные методологии моделирования трактуют понятие архитектуры по-разному. Под архитектурой понимают организацию системы и связывают это понятие с выделением в ней основополагающих частей [2]. Помимо этого, считается, что понятие архитектуры включает структуризацию и принципы взаимодействия составляющих системы между собой и с внешней средой, а так же принципы развития программной системы [3]. Любое изменения в системе требует анализа архитектуры. Наконец отмечается, что архитектура должна обеспечивать соответствие приложения своему предназначению [4,5], что расширяет круг составляющих системы, относящихся к архитектуре. Развитие системы может приводить к изменениям в самой архитектуре. Такие модификации наиболее трудоемкие и рискованные. Следовательно, качество сопровождения приложения во многом зависит от эффективности сбора и анализа информации о его архитектуре. Именно на эти задачи ориентируется предлагаемый подход.

Дадим конструктивное определение графа взаимодействий. Рассмотрим размеченный ориентированный граф <V, R>, где V – множество вершин, а R – множество дуг. Множество вершин конечно: $V= \{v_1,v_2,\dots v_n\}$. (1)

Для того, чтобы провести соответствие между моделью и программной системой и согласовать вносимые изменения, разделим множество вершин на две части:

V=E∪F, (2) где E – составляющие архитектуры программы, содержащее как абстрактные элементы так и элементы, близкие к структурным языковым конструкциям, а F – файлы, в которых лежат исходные тексты программы. При этом масштаб элементов множества E не ограничивается. К этому множеству можно отнести как функциональные модули, так и параметры функций. На множестве элементов архитектуры программы вводится разбиение по типам в зависимости используемых технологий, стандартов разработки и языка программирования:

$E = \bigcup_{e=1}^{\varepsilon} E_e$, где ε – количество групп в группировке. (3)

Фактически эта типизация представляет собой терминологию, которой пользуются проектировщики и программисты системы. В качестве примера группировки можно привести деление всех элементов внутренней структуры на переменные, функции, процедуры и так далее. Аналогичное разбиение вводится и для файлов:

$F = \bigcup_{f=1}^{\phi} F_f$, где ϕ - количество типов в этой типизации. (4)

Классификация файлов определяется также инструментами разработки, принятыми стандартами. Часто разделение файлов на типы производят в соответствии с их расширением. Рассмотрим множество дуг R. Дуги графа описывают взаимодействия между вершинами. Это могут быть структурные, управляющие, информационные и другие связи. Множество дуг тоже конечно: $R=\{r_1, r_2 \dots r_p\}$. (5)

Допускается наличие повторяющихся пар, то есть два элемента могут соединяться несколькими дугами. Связи классифицируются по тому, что они связывают:

$R= R_1 \cup R_2 \cup R_3$, где $R_1=\{\rho : a\rho b, a,b \in E\}$, $R_2=\{\rho : a\rho b, a,b \in F\}$,

$R_3=\{\rho : a\rho b, a \in F, b \in E\}$. (6)

Ориентация дуг в графе взаимодействий введена затем, чтобы определить смысловые роли в отношениях из множества $R_1 U R_2$. Например, в отношении «наследование между классами» дуги ориентированы в направлении от вершины-наследника к вершине-потомку. Для отношений между файлами и элементами внутренней структуры ориентация не существенна. Например, тип связи между файлом и функцией «содержание», когда файл содержит исходный текст функции, можно задать как тип дуги либо от вершины-файла к вершине-функции, либо, наоборот, от функции к файлу. Суть отношения от этого не

изменится. В связи с этим, для определенности будем считать, что отношения между файлами и элементами внутренней структуры задаются как дуги от вершин множества F к вершинам множества E, в графе дуги от элементов внутренней структуры к файлам отсутствуют: $\{\rho : a\rho b, a \in E, b \in F\} = \varnothing$. (7)

Далее каждая R_i разбивается еще на типы по смыслу: $R_1 = \bigcup_{k1=1}^{K1} R_{k1}^1$, $R_2 = \bigcup_{k2=1}^{K2} R_{k2}^2$ и $R_3 = \bigcup_{k3=1}^{K3} R_{k3}^3$. (8)

Эта классификация тоже определяется технологиями и средой разработки. В качестве примера видов связей из R_1 можно привести отношения типа «процедура-параметр», «функция-результат», «переменная-тип». К группам второго вида можно отнести отношения между файлами: «предыдущая версия – следующая версия» или «включение» (один файл подставляется в другой при компиляции). Простейший пример типов связей третьего вида – «содержание описания» (файл содержит описание компоненты) и «содержание реализации» (файл содержит реализацию компоненты). Фактически группы R_{k1}^1 представляют способы взаимодействия между элементами внутренней структуры программы, R_{k2}^2 – отношения между файлами, а R_{k3}^3 – особенности файловой организации. Пустые группы вершин и дуг из модели исключаются. При классификации дуг допускается ситуация $a\rho_1 b, a\rho_2 b : \rho_1 \in R_{l1}^k \rho_2 \in R_{l2}^k, l_1 \neq l_2$, то есть дуги, соединяющие одни и те же вершины, необязательно относятся к одной и той же группе. Фактически разметка дуг графа отвечает за их распределение по группам R_l^k. Возможна ситуация, когда взаимодействия, в которых участвуют элементы архитектуры одного типа, бессмысленны для других. Для этого каждому типу элементов внутренней структуры E_e ставится в соответствие множество Σ_e допустимых типов связи

$$\Sigma_e = \{ R_{l1}^1, R_{l2}^1 \ldots R_{lL}^1, \ \overline{R_{lL+1}^1}, \overline{R_{lL+2}^1} \ldots \overline{R_{lL+J}^1}, \ R_{m1}^2, R_{m2}^2 \ldots R_{mM}^2, \ \overline{R_{mM+1}^2}, \overline{R_{mM+2}^2} \ldots \overline{R_{mM+I}^2} \}. \quad (9)$$

Если нет черты сверху, такая дуга может входить в вершину заданного типа, а если есть – то может выходить из нее. Пустые множества исключаются. Множества Σ_e ограничивают взаимодействия элементов архитектуры. Для файлов такие ограничения не вводятся, чтобы упростить модель. В этом и заключается различие между множествами E и F. На первый взгляд может показаться, что разделение модели на файловую и архитектурную часть усложняет модель. Однако, при использовании модели набор операций, который можно производить с файлами и элементами внутренней структуры

будет различаться. Разделение файлов и элементов архитектуры позволит ускорить некоторые из них.

Таким образом, процесс моделирования программной системы состоит из трех частей:

1.Задаем типизацию узлов и связей. На этом этапе вводим разбиение узлов графа: $E_e, e \in N$ и $F_f, f \in N$, классификацию связей: R^1_{k1}, R^2_{k2} и R^3_{k3}. Задаются ограничения. Фактически, на этом этапе определяется, какими свойствами должен обладать элемент внутренней структуры или файл, чтобы попасть в ту или иную категорию. Здесь же определяется, как будут взаимодействовать компоненты программы.

2.Далее идет заполнение определенных групп фактическими составляющими программного комплекса. Этот процесс аналогичен проектированию системы в рамках заданной методологии и ее дальнейшей разработке.

3.В ходе выполнения пункта 2, может оказаться, что часть групп не заполнена. Их следует ликвидировать и убрать соответствующие ограничения.

Деление узлов и связей на группы позволяет показать неоднородность архитектуры и ввести множество ограничений Σ_e, что делает модель более адекватной предметной области. Граф взаимодействий может выразить практически любую современную нотацию представления программной системы, включая нотации семейства IDEF [9,10], DFD-нотации [10] и нотации, входящие в UML [11,12]. Классификация вершин и дуг позволяет интегрировать несколько представлений программной системы либо представления различных систем. Подробнее об это показано в [15]. Модель на основе графа взаимодействий согласуется с подходом Model Driven Architecture (MDA) [13,14], позволяя явно выделить среди множеств E_e и R^1_{k1} категории, которыми оперируют вычислительно-независимая модель, платформенно-независимая и платформенно-зависимая модели, и провести соответствия между уровнями MDA.

3. Граф взаимодействий как инструмент анализа архитектуры

Анализ графа взаимодействий нужно применять в следующих задачах:

- проверка корректности модели. Ошибки при задании типизации вершин и дуг могут привести к тому, что построенное представление будет объемным и громоздким;
- проведение рефакторинга и оптимизации моделируемой системы. Графовое представление значительно нагляднее, чем исходные тексты программы. Оно не зависит от среды и языка программирования. Кроме того, операции поиска на данном представлении более эффективны, чем обычный полнотекстовый поиск. Особенность таких методик состоит в том, что, достаточно проанализировать подграф $\{E, R_1\}$ с учетом множества

ограничений. Файловая часть в этом случае нужна для того, чтобы внести изменения в систему в соответствие с полученными результатами;

- оценка сложности программной системы, а также ее представления;
- оценка хода разработки и сопровождения программной системы.

Методики анализа зависят в первую очередь от выбранного представления. Методики анализа для одной классификации вершин и связей не всегда подойдут для другой классификации. Результаты могут оказаться либо тривиальными, либо некорректными. Однако существуют методики анализа графа взаимодействий, справедливые для любого представления. Будем их называть общими, а методики, ориентированные на конкретное представление, – частными. Кроме того, методики можно разделить по принципу анализа на три группы: количественные без учета топологии графа, топологические и динамические.

3.1 Количественные методики анализа графа взаимодействий

Первый тип анализирует количественные показатели графа без учета топологии его связей. Сюда можно отнести методики оценки сложности. Сложность представления можно оценить по количеству групп в классификации вершин и связей. Кроме того, учитывается количество разрешенных связей для каждой группы. Общая формула оценки сложности представления разработки имеет вид:

$$S = af_1(\varepsilon) + bf_2(\phi) + cf_3(K_1, K_2, K_3) + df_4\left(\frac{\sum_e |\Sigma_e|}{2\varepsilon}, \frac{K_3}{\phi}, \frac{K_2}{2\phi}\right). \quad (10)$$

Коэффициенты a, b, c, d, а также функции f_1, f_2, f_3, f_4 задаются в зависимости от представления и шкалы оценки. Функции f_1, f_2, f_3, f_4 представляют собой эвристики оценки. Параметры функций описаны в формулах-определениях множеств E, F, R_1, R_2, R_3, и множеств ограничений. Первые три слагаемых отражают влияние количества типов в типизации вершин и дуг, а последнее – зависимости сложности от среднего количества возможных связей на группу элементов. Чем меньше типов в классификации вершин и дуг, тем проще представление. При этом при простой нотации для каждого вида вершин количество возможных типов связи тоже должно быть минимальным. Сложность нотации представления может говорить как о подробном представлении, так и сложности технологий и стандартов и инструментов программирования и проектирования системы.

Аналогичным образом можно оценить сложность самой системы. Такая оценка похожа на оценку сложности по методике функциональных показателей. Сложность зависит от количества вершин графа. Эту зависимость можно ранжировать по типам, так как элементы архитектуры программы различного вида различаются по сложности. Например,

разработка интерфейса компонента может быть значительно проще, чем разработка реализации этого интерфейса. Помимо этого, можно ввести зависимость сложности от количества дуг графа. Она будет отражать сложность согласования компонент системы, ведь разрабатывать и, главное, сопровождать независимые компоненты проще, чем связанные. Обычные методики оценки на основе функциональных показателей этот факт не учитывают.

Таким образом, общая формула оценки сложности программной системы по графу взаимодействия будет выглядеть так:

$$\Pi = a\sum_{e}^{\varepsilon} f_e^1(|E_e|) + b\sum_{f}^{\phi} f_f^2(|F_f|) + c_1\sum_{k}^{K_1} g_k^1 |R_k^1| + c_2\sum_{k}^{K_2} g_k^2 |R_k^2| + c_3\sum_{k}^{K_3} g_k^3 |R_k^3|, \quad (11)$$ где

коэффициенты a, b, c_1, c_2, c_3 определяются представлением и принятой шкалой, а функции $f_e^1, f_f^2, g_k^1, g_k^2, g_k^3, \quad e, f, k \in N$ – эвристики оценки. Это монотонные неубывающие неотрицательные функции натурального аргумента. В простейшем случае это могут быть постоянные, ступенчатые функции или функции, возвращающие свой аргумент. Требование монотонного неубывания и неотрицательности обусловлено усложнением системы при наращивании архитектуры. Увеличить оценку сложности может также и усложнение представления в том случае, если добавляются новые группы и новые элементы графа. Упростить систему при помощи изменения представления можно только в том случае, если при новом представлении сократится количество элементов в некоторых категориях

Другая проблема, которая может быть выявлена данным способом – очень детальное моделирование. Не эффективно описывать систему графом взаимодействий с точностью до операторов языка программирования: теряется наглядность. Разрастание графа усложняет операции анализа вне зависимости от масштаба.

Преимущество предложенных методик оценки заключается в их простоте и наглядности. Недостаток проявляется в трудности подбора коэффициентов и эвристик, необходимости сбора статистики, а так же в том, что оценку невозможно провести до проектирования системы.

Аналогичные методики оценки можно применять при анализе корректности как модели, так и архитектуры программной системы. При этом анализируется соотношение мощностей типов вершин и дуг. Большие мощности одних групп и сравнительно-небольшие мощности других могут свидетельствовать о следующих ситуациях:

- неправильно подобрано представление, система неправильно спроектирована. Возможно, от отдельных групп в классификации можно совсем отказаться;

- неверно выполнена декомпозиция крупных компонент системы. То есть, в системе много крупных перегруженных компонент, их следует разбить на подкомпоненты, ввести дополнительные уровни декомпозиции;
- неверно выполнена декомпозиция, но проблема обратная: в системе накопилось много однотипных мелких элементов, возможно, их стоит объединить. Например, при объектно-ориентированном подходе в системе появилось много функций с большим количеством простых параметров. Ситуация может усугубляться тем, что набор параметров у большинства из них совпадает. Кроме того, может оказаться, что часть этих функций передают между собой вызовы. В этом случае целесообразно ввести сложный тип и сократить число параметров.

Для анализа, как и при оценке сложности, вводится шкала и вычисляется эвристическая функция:

$$\mathrm{K} = \mathrm{K}(|E_1|,|E_2|,...,|E_{\mathrm{E}}|,|F_1|,|F_2|,...|F_{\phi}|,|R_1^1|...|R_{K1}^1|,|R_1^2|...|R_{K2}^2|,|R_1^3|...|R_{K3}^3|). \quad (12)$$

Для этой функции определяется по шкале область допустимых значений и проверяется, чтобы полученный результат попал в указанную область. Достоинство данного метода – простота, недостаток же опять в необходимости собирать статистику и подбирать эвристики в зависимости от введенной классификации. Более того, чтобы выявить проблему, может потребоваться несколько эвристик, при этом метод не покажет, какие конкретно компоненты надо перестраивать. Этот метод целесообразно применять в начале моделирования либо на начальных этапах рефакторинга. Для дальнейшего анализа применяются топологические методики анализа.

3.2 Анализ топологии графа взаимодействий

Эти методики исследуют, как располагаются связи в графе, и могут применяться для локализации изменяемых компонент системы, при ее рефакторинге, оптимизации, а также при проверке корректности построения модели. Их особенность заключается, в том, что при анализе необходимо учитывать не только взаимодействие компонент программной системы, но и характер представления архитектуры. В частности, необходимо учитывать, что одна и та же система может быть описана в нескольких представлениях, и особое внимание надо уделять согласованию этих представлений. Вторая особенность заключается в неоднородности группировки. Это приводит к тому, что для каждой методики анализа ограничиваются типы рассматриваемых вершин и дуг, то есть анализируется не весь граф, а некоторое его подмножество. Рассмотрим некоторые наиболее общие методики анализа. Как

правило, в графе взаимодействий исследуют: смежность, связность, возможности поиска, блочность, циклы.

3.2.1 Анализ смежности в графе взаимодействий.

Две вершины $a,b \in V$ смежные, если существует дуга $\rho \in R$: $a\rho b$ или $b\rho a$. Дуги, у которых начало и конец совпадают, называются петлями. В графе взаимодействий петли запрещены. На множестве F петли бессмысленны. На множестве E петли могут обозначать взаимодействие элемента самого с собой. Для структурных и информационных связей такое взаимодействие тривиально. Оно может иметь смысл разве что для управляющих дуг, например, чтобы показать рекурсию. В этом случае целесообразно выделить отдельную группу вершин, взаимодействующих с собой указанным образом, а петли удалить. Например, пусть имеется группа «функции», среди них встречаются рекурсивные, и это надо показать. Необходимо завести дополнительно группу «рекурсивные функции», переместить туда все вершины типа «функция», у которых есть петли, и эти петли удалить.

Две дуги $\rho, \rho_1 \in R$ называются кратными, если $a\rho b$ и одновременно $a\rho_1 b$, то есть они соединяют одинаковые вершины. Для некоторых типов связей кратные дуги одного и того же типа необходимо запретить. Для структурных связей кратные дуги бессмысленны. Для информационных связей кратные дуги необходимы только в том случае, если между двумя компонентами идет несколько информационных потоков, которые невозможно объединить в один. Такое может возникать, например, если данные существенно различаются по типу или по источнику. При описании передачи управления количество взаимодействий – переменная величина. В этом случае дуга обозначает возможность взаимодействия при некоторых условиях, а кратные дуги объединяются в одну дугу.

Наличие большого количества кратных дуг, относящихся к разным типам, может свидетельствовать о том, что следует пересмотреть типизацию связей. Для каждого типа связи R_j^i целесообразно рассчитать отношение:

$$\frac{D_j^i}{|R_j^i|}, \text{ где } D_j^i = |\{\rho : \rho \in R_j^i, a\rho b, \exists \rho_1 \notin R_j^i, a\rho_1 b\}|. \quad (13)$$

Таким образом, D_j^i – количество дуг во множестве R_j^i, для которых существует кратная дуга другого типа. Если отношение $\frac{D_j^i}{|R_j^i|}$ близко к единице, тип взаимодействия R_j^i не самостоятельный. Связи из R_j^i представляют собой дополнение к взаимодействиям других типов. В таком случае надо выявить эти типы: $R_l^k : \exists \rho_1 \in R_l^k, a\rho_1 b, \exists \rho \in R_j^i, a\rho b$ и для

каждого из них создать подгруппу R_{l1}^{k1}. Объединяем кратные дуги: $\forall \rho_1, \rho: \quad \rho_1 \in R_l^k, a\rho_1 b, \rho \in R_j^i, a\rho b$ создаем связь $\rho_2 \in R_{l1}^{k1}, a\rho_2 b$ а дуги ρ_1 *и* ρ удаляем. Кроме того, в этом случае необходимо доработать множество ограничений для новых групп. Очевидно, что дуга типа R_{l1}^{k1} может быть заведена в том и только в том случае, если может быть заведена дуга из R_j^i и дуга из R_l^k. После того, как обработаны все типы дуг, необходимо группу R_j^i удалить из типизации. Кроме того, если дуги из R_l^k не применялись без дуг из R_j^i, то часть групп в классификации будут пустыми. Их надо удалить.

Рассчитаем отношение: $\dfrac{\overline{D_j^i}}{|R_j^i|}$, $\overline{D_j^i} = |\{\rho : \rho \in R_j^i, a\rho b, \exists \rho_1, b\rho_1 a\}|$ (14)

Количество дуг во множестве R_j^i, для которых существует обратная дуга. Если эта величина близка к единице, то, как и в предыдущем случае, необходимо перестроить группировку связей. Для каждого типа $\forall R_l^k : \exists \rho_1 \in R_l^k, b\rho_1 a, \exists \rho \in R_j^i, a\rho b$ создаем группу связей R_{l1}^{k1}. Объединяем обратные дуги: $\forall \rho_1, \rho : \rho_1 \in R_l^k, b\rho_1 a, \rho \in R_j^i, a\rho b$ создаем связь $\rho_2 \in R_{l1}^{k1}, b\rho_2 a$ а дуги ρ_1 *и* ρ удаляем. Так же, как и в предыдущем случае, дорабатываем множества ограничений. Дуга типа R_{l1}^{k1} может быть заведена в том и только в том случае, если может быть заведена дуга из R_j^i и обратная дуга из R_l^k.

Описанные действия направлены на сокращение количества дуг графа. При этом для информационных взаимодействий этот алгоритм не подходит, так как приводит к разрастанию типизации дуг. Получается много видов связей, но дуг в каждой такой группе будет мало. Следовательно, среднее количество дуг на группу будет низким. Для всех типов взаимодействий, кроме информационных, описанные преобразования не затрагивают программную систему. Они касаются только классификации вершин и дуг графа, то есть представления программной системы. Для анализа самой программной системы необходимо обратить внимание на вершины. Как правило, анализируется множество вершин, связанных с данной. Это может быть множество вершин, связанных с заданной исходящими дугами:

$\Gamma(x) = \{a : \exists \rho \in R_1, x\rho a\}$, либо входящих: $\Gamma^{-1}(x) = \{a : \exists \rho \in R_1, a\rho x\}$, где x – указанная вершина. (15)

Множество $\Gamma(x)$ называется множеством потомков, а множество $\Gamma^{-1}(x)$ – множество предшественников. Иногда целесообразно ограничить типы вершин или типы дуг:

$$\Gamma_{\bigcup_e E_e,\bigcup_k R_k^1}(x) = \{a \in \bigcup_e E_e, e < \varepsilon : \exists \rho \in \bigcup_k R_k^1, k < K_1, x \rho a\} \quad (16)$$

$$\Gamma^{-1}{}_{\bigcup_e E_e,\bigcup_k R_k^1}(x) = \{a \in \bigcup_e E_e, e < \varepsilon : \exists \rho \in \bigcup_k R_k^1, k < K_1, a \rho x\} \quad (17).$$

Так, например, решается задача отыскания всех вызовов процедуры или функции при переименованиях. Вызовы оформляются как отдельный тип связи, например R_k^1. Дуги этого типа идут непосредственно от вызывающей вершины к вызываемой. При переименовании вызываемой вершины необходимо найти все места, где используется старое наименование, чтобы его изменить. По графу взаимодействий для вершины x, которую мы будем переименовывать, можно найти множество всех предшественников, связанных с данной вершиной дугой типа «вызов» $\Gamma^{-1}{}_{E,R_k^1}(x) = \{a : \exists \rho \in R_k^1, a \rho x\}$ и посмотреть, в каких файлах находится реализация этих вершин. Непосредственный полнотекстовый поиск по исходным текстам менее эффективен.

Количество дуг, входящих в вершину называется полустепенью захода, а исходящих из вершины – полустепенью исхода. Полустепень захода обозначается $\deg^+(x)$:

$$\deg^+(x) = |\{\rho, a \rho x\}| \quad (18)$$

Полустепень исхода – $\deg^-(x)$: $\deg^-(x) = |\{\rho, x \rho a\}|$ (19).

Их сумма называется степенью вершины и обозначается

$$\deg(x) = \deg^-(x) + \deg^+(x) .(20)$$

При отсутствии кратных дуг и петель $\deg^-(x) = |\Gamma(x)|$, а $\deg^+(x) = |\Gamma^{-1}(x)|$. (21)

В общем случае $\deg^+(x) \geq |\Gamma^{-1}(x)|$ и $\deg^-(x) \geq |\Gamma(x)|$. (22)

Для изолированных вершин степень нулевая. Как правило, такие компоненты из программной системы можно удалить, за исключением тех случаев, когда это отдельные программные утилиты. Так же как и в предыдущем примере, можно ограничить типы вершин и типы связей: $\deg^-{}_{\bigcup_e E_e,\bigcup_k R_k^1}(x) = |\{\rho, \rho \in \bigcup_k R_k^1, k < K_1, x \rho a, a \in \bigcup_e E_e\}|$ (23)

$$\deg^+{}_{\bigcup_e E_e,\bigcup_k R_k^1}(x) = |\{\rho, \rho \in \bigcup_k R_k^1, k < K_1, a \rho x, a \in \bigcup_e E_e\}| \quad (24).$$

Анализ этих величин поможет выявить перегруженные компоненты системы – те, у которых эти величины большие, и недогруженные – те, у которых эти величины невелики или даже нулевые. Это используется при рефакторинге. Из системы удаляются не используемые компоненты. Для элементов системы, у которых слишком много непосредственных составляющих, вводят новые уровни абстракции. Кроме того, о сложности компоненты могут свидетельствовать конструкции с множественным

наследованием. Аналогично, если обнаруживается большое число компонент простых типов, например, среди параметров функций, бывает целесообразно их сократить, введя сложный тип. Задачи, решаемые таким способом, очень разноплановые, однако, принцип один. Достоинство этого метода, заключается в его простоте.

3.2.2 Анализ связности графа взаимодействий

Путь из вершины a в вершину b – последовательность вершин и дуг графа вида $a(a,x_1)\otimes x_1(x_1,x_2)...x_{n-1}(x_{n-1},b)b$. Путь простой, если ни одна вершина не повторяется в нем дважды. Длина пути измеряется по количеству дуг, входящих в него. Путь, начало и конец которого совпадают, называется циклом.

Вершина w достижима из вершины v, если либо они совпадают v=w, либо существует путь из v в w. Граф (ориентированный граф) связным (сильно связным), если для любых двух его вершин v и w существует путь, соединяющий v и w (из v в w). Ориентированный граф односторонне связный, если для любых двух его вершин по крайней мере одна достижима из другой. Для ориентированного графа вводят понятие ассоциированного с ним псевдографа. Ассоциированный псевдограф получается из заданного орграфа путем снятия ориентации с его дуг. Ориентированный граф слабо связный, если его ассоциированный псевдограф связный.

Если граф не является связным, то в нем можно выделить компоненты связности. Компонента связности (сильной связности) графа (орграфа) – его связный (сильно связный) подграф, не входящий ни в какой другой связный (сильно связный) подграф. Граф взаимодействий может быть несвязным тогда и только тогда, когда выполнено хотя бы одно из трех условий:

- В системе можно выделить независимые программные компоненты. Это означает, что некоторые части системы между собой не взаимодействуют. На практике подобная ситуация может свидетельствовать о неполноте информации или об ошибке.
- Система описана в нескольких представлениях, и среди них есть как минимум два, никак не согласованных между собой. Возможно, граф взаимодействий был получен путем объединения двух или более графов согласно второму способу объединения моделей, но связь, отражающая соответствие, была проставлена некорректно.
- Выбранное представление не позволяет показать некоторые виды взаимодействия между компонентами программной системы.

Если получается сильная связность, типизация связей избыточна, и часть типов необходимо удалить. Причиной того, что граф стал сильно связным, может оказаться

слишком сложная организация программы или неправильно подобранное представление. В этом случае необходимо сократить количество связей в графе, чтобы он стал хотя бы односторонне связным. Чтобы избежать ситуации, когда в графе становится слишком много дуг, мощность множества R можно ограничить.

Граф полносвязный, если любые две его вершины связаны непосредственно. Если граф взаимодействий полносвязный с учетом петель, то $| R |\geq (| E | + | F |)^2$.

При отсутствии кратных дуг и петель будет справедливо:

$| R |\leq (| E | + | F |)^2 - (| E | + | F |)$. (25)

Соответственно, равенство будет достигнуто при наличии дуг между любыми двумя различными вершинами. В общем случае вводят ограничение :

$| R |\leq \varphi(| E | + | F |) \leq (| E | + | F |)^2 - (| E | + | F |)$. (26)

Функция $\varphi(x)$ такова, что $\varphi(x) \to kx, x \to \infty$. То есть, при возрастании количества вершин ограничение числа дуг начинает зависеть от него линейным образом: в крупных системах, вероятность того, что компонента будет непосредственно взаимодействовать со всеми компонентами без исключения, крайне мала. Абстракция сужает круг непосредственного взаимодействия каждой компоненты.

3.2.3 Возможности поиска

Как правило, графовые модели используют для поиска путей. Граф взаимодействий позволяет сконцентрировать внимание на вершинах. Критерием поиска может быть тип вершины, наличие связи определенного вида, наличие связи с вершиной определенного типа, а также комбинация этих критериев. Совокупность критериев поиска задается шаблоном. Шаблон – слабо связный граф, вершины и дуги которого классифицируются и подчиняются тем же ограничениям, что и вершины и дуги графа взаимодействий. Кроме того, к классификации дуг и вершин шаблона добавляются неопределенный тип для дуги и для вершины. Ограничение на использование этих типов нет. Шаблон представляет собой искомую комбинацию вершин и дуг графа взаимодействий.

Пусть задан шаблон $T = \{TV, TR\}$. Обозначим неопределенный тип для дуг и вершин R_n и E_n соответственно. Подграф $g = \{gV, gR\} \subseteq G$ – основного графа взаимодействий будет соответствовать шаблону тогда и только тогда, когда найдутся два отображения $\eta V : TV \to gV$ для вершин и $\eta R : TR \to gR$ для дуг. Отображения должны быть определены для всех составляющих шаблона. Возможно, не для всех элементов подграфа существует обратное отображение. Отображения ηV и ηR должны быть инъективны, то есть разным

дугам и вершинам шаблона должны соответствовать разные дуги и вершины подграфа. Кроме того, требуется выполнения следующего:

$\forall v \in TV, \eta V(v) = w \in gV \Leftrightarrow v, w \in E_e$ или $v, w \in F_f$ – принадлежат одной группе или $v \in E_n$. (27)

Это означает, что вершине неопределенного типа может соответствовать любая вершина, а вершинам, которые распределены по классификации графа взаимодействий, должны соответствовать вершины тех же типов. Аналогичное правило действует и для дуг, но для них дополнительно требуется соответствие концов:

$\forall \rho \in TR, \quad \eta R(\rho) = \rho_1 \in gR \Leftrightarrow (\rho, \rho_1 \in R_j^i \text{ или } \rho \in R_n)$ и $(v\rho w, v_1\rho_1 w_1, \quad v, w \in TV, \quad v_1, w_1 \in gV)$ и $(\eta V(v) = v_1, \eta V(w) = w_1)$.(28)

Обозначим соответствие подграфа g шаблону T как $g \sim T$. Из требований к отображениям видно, что из $g \sim T$ следует $\forall g_1 \supseteq g \; g_1 \sim T$. Очевидно, что для $g \sim T$ не существует $g_2 \subset g$, такого что $g_2 \sim T$ тогда и только тогда, когда отображения ηV и ηR будут биекциями, то есть взаимно однозначными. Это означает, что дополнительно для любой дуги или вершины подграфа будет определен свой собственный объект шаблона.

Отображения ηV и ηR изначально определяются как инъективные, чтобы отсечь необходимость описывать в шаблоне полную структуру искомой компоненты. Это связано с тем, что чем меньше мощности множеств TV и TR, тем проще искать подходящие подграфы. Алгоритм поиска по шаблону похож на поиск подстроки в строке, с той лишь разницей, что строка – линейная структура, а граф – нет:

1. В шаблоне выделяем произвольную начальную вершину и делаем её текущей. Желательно, чтобы ее тип не был неопределенным. Это поможет ускорить поиск.

2. Выбираем из рассматриваемого графа все вершины соответствующего типа или любого типа, если тип первой вершины не определен. Это будет список вершин-претендентов. Далее выполняем алгоритм для каждой вершины из этого списка.

3. Находим все дуги в шаблоне, входящие или выходящие из текущей вершины. Проверяем, есть ли у рассматриваемой вершины-претендента все дуги таких же типов. Если какой-то дуги не хватает, то она не подходит под шаблон и надо перейти на новую вершину-претендента и повторить п.3 с начала .

4. Если все дуги подошли, то сверяем множество предшественников и последователей у рассматриваемой вершины и текущей вершины шаблона с учетом типов вершин и типов выбранных дуг. То есть дуги в шаблоне и соответствующие им дуги в графе

должны вести к вершинам, подходящим по типам. Если чего-то не хватает, то переходим на следующую вершину в списке претендентов и начинаем все с п.3.

5. В противном случае помечаем рассмотренные вершины и дуги. Это делается, чтобы исключить повторную обработку. В дальнейших операциях эти элементы графа участвовать не будут.

6. Далее аналогично рассматриваются все вершины-предшественники и последователи текущей вершины шаблона. Если для каждой из них нашлись входящие и исходящие дуги, нужного типа, а так же подходящие предшественники и последователи, то алгоритм продолжается.

7. В противном случае снимаем все метки и переходим на следующую вершину-претендента графа. Переходим на п.3.

8. Если, наконец, мы дошли до вершины шаблона, у которой нет необработанных входящих и исходящих дуг, то мы нашли одно из соответствий шаблона.

9. После этого все помеченные вершины переписываются, как результат, а временные пометки с них снимаются. Если требуется найти другие соответствия, алгоритм переходит на новую вершину-претендента, и все начинается с п.3.

Важно отметить, что соответствие шаблону является лишь архитектурным. Соответствие нескольких подграфов одному шаблону не означает их полного совпадения. Ведь если мы нашли в программной системе, например, две процедуры, у которых все параметры соответствуют по типу, это не означает, что они выполняют одни и те же действия. Тем не менее, использование шаблонов эффективно при поиске дублирующегося кода, при структуризации системы, при поиске компонент системы для повторного использования, типовых решений и типичных ошибок. Похожие по структуре архитектурные составляющие содержат дублирующийся код. При поиске компонент системы или типового решения шаблон описывает ситуацию, в которой применяется искомый компонент или решение. При структуризации указывается внутренняя структура искомого элемента. Например, если ищем функцию, можно указать типизацию параметров. Неизвестные компоненты отмечаются неопределенными типами. Для поиска типичных ошибок в архитектуре шаблон описывает, в чем конкретно заключается ошибка. Шаблоны для графа взаимодействий можно применять как шаблоны рефакторинга [6,7]. Если использование шаблонов в этих задачах не позволяет решить их полностью автоматически, то, по крайней мере, помогает сузить область ревизий.

3.2.4 Анализ блоков

Для многих нотаций представления характерна декомпозиция программной системы на части. Эти части называются по-разному, здесь и далее будем называть их блоками. С точки зрения графа блок представляет собой слабо связный подграф, состоящий из вершины, обозначающей блок, вершин, обозначающих содержимое блока, и связей между ними. Для нахождения блоков выбираем один или несколько типов связей. Назовем дуги этих типов межблочными связями: $EBR = \{\rho : \rho \in R^1_{k1} \cup R^1_{k2} ... \cup R^1_{kn} \subset R^1\}$. (29)

Тогда вершины-блоки находятся как вершины, соединенные межблочными дугами:

$BV = \{v : \exists \rho \in EBR, v_1, v\rho v_1\} \cup \{v : \exists \rho \in EBR, v_1, v_1 \rho v\}$. (30)

Часть типов связей, не являющихся межблочными, обозначают как внутриблочные:

$IBR = \{\rho : \rho \in R^1_{k1} \cup R^1_{k2} ... \cup R^1_{kn} \subset R^1 / EBR\}$. (31)

Таким образом, блок состоит из вершины-блока, вершин, с нею связанных и связей между ними: $B = \{IBV, BR\}$, где

$IBV = \{v : v \in BV, \forall w \in BV \cap IBV \Rightarrow w = v\} \cup \Gamma_{IBR}(v \in BV \cap IBV) \cup \Gamma^{-1}{}_{IBR}(v \in BV \cap IBV)$

$BR = \{\rho \in IBR : \exists v, w \in IBV, v\rho w\}$ (32).

Если множество межблочных связей определено верно и информация о системе полная, то в графе не будет вершин, не попавших ни в один блок, то есть $E = \bigcup_i B_i$. Однако, не всегда можно гарантировать при данном построении, что блоки не пересекутся. В общем случае $\forall i, j: \quad B_i \cap B_j \neq$ Ш. (33)

У блока могут появиться связи, относящиеся к внутриблочным и выходящие за его пределы. Важно рассмотреть вершины блока, участвующие в таких связях. Это внутренние компоненты блока, обращающиеся к другим блокам. Рассмотрим блок B. Множество его внешних связей, являющихся внутриблочными:

$SR = \{\rho \in IBV / BR : \exists v \in IBV, w \notin IBV, v\rho w\} \cup \{\rho \in IBV / BR : \exists v \in IBV, w \notin IBV, w\rho v\}$ (34).

Внутренние вершины блока, участвующие в этих связях:

$SV = \{v \in IBV : \exists \rho \in SR, v\rho w\} \cup \{v \in IBV : \exists \rho \in SR, w\rho v\}$ (35).

Если у некоторой вершины из этого множества количество связей, относящихся к блоку, меньше, чем количество связей, к нему не относящихся, то вершину надо пересмотреть. Скорее всего, ее необходимо вынести из этого блока. Так находятся компоненты, которые необходимо поднять вверх или опустить вниз по иерархии декомпозиции, либо те, которые надо переместить между ее горизонтальными звеньями [7]. При этом природа этих компонент в общем случае не важна.

Данный принцип не работает для некоторых структурных методологий. Особенность состоит в том, что явно выделить межблочные связи нельзя. Вершины-блоки взаимодействуют между собой так же, как и обычные вершины. Кроме того, эти методологии отличаются тем, что почти любую компоненту можно декомпозировать, и взаимодействие между содержимым блока и другими блоками идет только через компоненту-блок. В этом случае заводят специальный тип дуги «декомпозиция», соединяющий декомпозируемый элемент, со всеми непосредственными составляющими. Обозначим этот тип связей R_d. К внутриблочным связям отнесем все типы связей, кроме декомпозиции $R_D \subset IBR$. Множество вершин-блоков находится как множество декомпозированных компонент: $BV = \{v : \exists \rho \in R_d, v_1, v \rho v_1\}$ (36).

Для каждой вершины-блока $v \in BV$ строим блок:

$B(v) = \{IBV(v), BR\}$, где $IBV(v) = \Gamma_{R_D}(v) = \{w : \exists \rho \in R_D, v \rho w\}$ (37) – все вершины, декомпозирующие данную, а $BR = \{\rho \notin R_d : \exists v, w \in IBV, v \rho w\}$ – множество связей между ними. Обратим внимание, что сама вершина блок в блок не входит $v \notin IBV(v)$. Тогда во множество SR необходимо отобрать дуги между вершинами блока и соответствующей вершиной-блоком. Именно эти дуги будут обозначать взаимодействия внутренних вершин блока с внешними. Как и в предыдущем случае, строим множество SV и считаем для его вершин соотношение связей с вершинам внутри блока и вне его. Такой способ построения блоков поможет выявить неправильную декомпозицию, если не удается явно выделить межблочные связи.

3.2.5 Анализ циклов

Рассмотрим, дуги каких типов образуют циклы. Структурные связи циклы образуют редко. Они обычно описывают иерархию, и, следовательно, их топология древовидна. Циклы в таких связях образуются, когда надо ввести ассоциацию между элементами одной ветви, но разных уровней. Следовательно, часть циклов можно обнаружить по дугам этого типа.

Информационные и управляющие связи образуют циклы значительно чаще. Циклы между управляющими связями необходимо удалять, потому что они свидетельствуют либо о неправильно построенном взаимодействии, либо о том, что описывается не архитектура системы, а ее поведение. Пусть имеется цикл из *m* вершин. Перенумеруем их по порядку для определенности. Управляющие связи описывают передачу управления, значит, связь от *m*-той вершины к первой означает передачу управления. Это может быть возврат управления, и

тогда, эту связь можно удалить из графа, так как возврат управления произойдет по умолчанию, либо это новый вызов первой вершины, то есть эта вершина рекурсивна, и связь описывает поведение системы, а не ее архитектуру. При удалении циклов на управляющих дугах сокращается количество дуг в графе, модель упрощается и повышается эффективность анализа архитектуры.

Поиск таких циклов представляет более сложную задачу, чем просто отыскание дуг определенного типа. Для этого используются общие алгоритмы нахождения циклов в графе, которые основывающиеся на исследовании достижимости, для чего, как правило, применяется поиск в глубину. Примером может служить алгоритм Джонсона [8]. Так как граф взаимодействий редко бывает сильно связным, для корректной работы в графе необходимо выделить компоненты сильной связности, что тоже достаточно трудоемкая задача. Для упрощения анализа исключаются дуги структурных и информационных типов.

Для информационных связей возникновение циклов вполне законно. Если удалить такие циклы, то будет непонятно, как происходит обмен информацией между компонентами архитектуры системы. Анализ циклов на информационных связях применяется при рефакторинге и оптимизации системы и сводится к сокращению циклов, превышающих по количеству различных вершин определенное ограничение. Циклы обработки информации большой длины сложны для понимания архитектуры, выявления ошибочных ситуаций. Длинные циклы на информационных связях могут встречаться при излишнем разделении архитектуры на слои [2], при большом количестве посредников в обработке информации. Также возможны ситуации, когда информация хранится в заведомо неадекватной форме, и при ее использовании постоянно требуется ее преобразование. Ограничение на длину цикла позволяют упростить алгоритм их поиска.

3.3 Анализ динамики графа взаимодействий

При анализе динамики графа взаимодействий исследуются его модификация на протяжении жизненного цикла программной системы. Исследуемые изменения не должны быть связаны с изменениями классификации дуг и вершин графа, а также введенных ограничений. Такие методики можно использовать при анализе хода разработки и сопровождения системы. Они помогают оценить изменения в архитектуре программной системы.

Граф взаимодействий позволяет строго разделить все изменения на три группы: добавление в систему новых вершин и связывание их между собой и со старыми вершинами, добавление в систему связей между уже существующими вершинами, а также удаление из графа вершин и связей. Добавление новых функций в систему подразумевает создание

новых вершин в графе и связывание их со старыми. Удаление вершин и связей можно отнести к рефакторингу системы, а добавление новых связей между старыми вершинами – к переработке старой функциональности. Такое разделение может помочь точно классифицировать вносимые в систему изменения. Кроме того, это дает возможность по количеству вершин и дуг, попавших в каждую категорию, рассчитать соотношение объемов проведенных работ.

Производимые доработки могут быть мелкими и незначительными, а могут быть достаточно глобальными. Выяснить масштаб вносимых изменений поможет классификация дуг и вершин, затрагиваемых изменениями. Это можно сделать по соотношениям количеств новых и удаленных дуг и вершин в разных классах введенной типизации. При анализе можно дополнительно учитывать классификацию старых вершин, для которых изменились входящие или исходящие дуги.

Анализ графа взаимодействий поможет выяснить скорость вносимых изменений. На этапе разработки смысл этой скорости – средняя производительность труда программистов, а на этапе сопровождения – средняя частота вносимых изменений. Для этого вводят шкалу времени и делят количество новых и удаленных дуг или вершин графа на период времени. При расчетах можно также учитывать типизацию вершин и классификацию изменений. Очевидно, что для сопровождения и разработки масштаб шкалы будет различен.

Для обобщения этого подхода ход разработки программной системы опишем графиком зависимости количества вершин и дуг в графе взаимодействий от времени. Аналогично, ход сопровождения описывается графиком зависимости количества новых и удаленных вершин и дуг от времени. При этом учитывается типизация элементов графа и категорию изменений.

Приведенная классификация методик условна. Например, для некоторых методик количественного анализа обоснование строится, исходя из топологии графа, а некоторые топологические и динамические методики базируются на количественных показателях.

4. Заключение

Предложенный подход позволяет описать систему в рамках заданной нотации, определяемой инструментами и технологиями разработки. Представление задает типизацию дуг и вершин графа. Допускается уточнение, расширение представления и совместное использование нескольких нотаций. Для этого в классификацию вершин и дуг вносятся новые группы. Организация модели позволяет согласовать внесение изменений в модель и ее прототип при сопровождении системы.

Для решения задач сопровождения программной системы в рамках описанного представления разработан аналитический аппарат, который можно адаптировать под заданное представление. Графовая организация обеспечивает сравнительно простую реализацию методик анализа. Наличие классификации дуг и вершин позволяет обобщить некоторые операции анализа графа на различные представления. Использование при анализе теории графов создает математическую базу для операций анализа и обеспечивает строгость вычислений.

Библиографический список

[1] Глас Р. Факты и заблуждения профессионального программирования.: Пер с англ. –СПб.: Символ-Плюс,2007

[2] Фаулер М. Архитектура корпоративных программных приложений.: Пер с англ. – М.:Издательский дом «Вильямс», 2008

[3] Recommended Practice for Architectural Description of Software-Intensive Systems, ANSI/IEEE Std 1471-2000

[4] McGregor J. D. Software Architecture: Journal of Object Technology (JOT), vol.3, No. 5, pp. 65-77.

[5] Vinoski S. Do You Know Where Your Architecture Is?: IEEE Internet Computing, September-October 2003.

[6] Кериевски Д. Рефакторинг с использованием шаблонов.: Пер с англ. – М: ООО «И.Д. Вильямс», 2006

[7] Фаулер М. Рефакторинг: улучшение существующего кода.: Пер с англ. – СПб: Символ-Плюс, 2008

[8] Евстигнеев В.А. Применение теории графов в программировании. Под ред. А.П. Ершова. – М.: Наука, Главная редакция физико-математической литературы, 1985.

[9] Черемных С.В., Семенов И.О. Ручкин В.С. Структурный анализ систем: IDEF-технологии. – М.: Финансы и статистика, 2001

[10] Калянов Г.Н. CASE: структурный системный анализ (автоматизация и применение). – М.: ЛОРИ. 1996.

[11] Фаулер М. UML. Основы, 3-е издание.-СПб:Символ-Плюс,2006

[12] Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. 2-е изд.:–СПб: ДМК Пресс, 2007

[13] Brown, A. An Introduction to Model Driven Architecture //IBM Rational Developer Works, 2004.
[14] Кузнецов. М. MDA — новая концепция интеграции приложений // Открытые системы.2003. № 09.
[15] И.В Игнацкая, В.Н. Лукин. Моделирование программных систем на основе графа взаимодействий // Вестник Московского Авиационного Института 2009, т.16, №7, с 70-76

Сведения об авторе

Игнацкая Ирина Владимировна, аспирант Московского авиационного института (государственного технического университета), e-mail: Yrisky@yandex.ru.