Добыча формальных спецификаций программ для дальнейшего применения в автоматической генерации сообщений к коммитам

И.А. Косьяненко, Р.Г. Болбаков

***Аннотация* — В современной разработке программного обеспечения, имеющей командный характер, крайне важны хорошие сообщения к коммитам — комментарии к внесенным изменениям на естественном языке. Метрикой, позволяющей оценить сообщение к коммиту, является его релевантность. Хорошее сообщение к коммиту должно описывать не только внесенные изменения, но и их контекст.**

С ростом популярности систем контроля версий возросло число исследований, направленных на создание инструментов автоматической генерации сообщений к коммитам. Наиболее перспективными для решения данной задачи являются методы, основанные на глубоком обучении и нейронном машинном переводе. На вход таким методам подается программный код или что-то, к чему он может быть сведен. Единственный возможный способ автоматически сгенерировать сообщение к коммиту — «заглянуть» в историю изменений (diff). Все современные системы контроля версий предоставляют пользователю такую историю, а для сравнения изменений между двумя версиями программного кода были спроектированы специальные алгоритмы.

Но парадигм программирования великое множество, еще больше существует языков программирования со своими особенностями и синтаксисом. Чтобы сделать инструмент генерации сообщений к коммитам универсальным, необходимо иметь возможность свести программный код на любом языке программирования к единой «нотации».

Одним из возможных примеров такой нотации являются формальные спецификации. Спецификация показывает, что программа должна делать. Существует множество способов записи формальных спецификаций: от алгебраических выражений и языков спецификаций до детерминированных конечных автоматов. Так как сообщение к коммиту должно давать представление о том, что и как изменилось в новой версии ПО, очень богатой кажется идея применения спецификаций программ в инструментах генерации сообщений к коммитам.

В настоящее время большинство программ не имеют формальной спецификации. Считается, что затраченное на проработку спецификаций время не стоит получаемого результата. Один из эффективных подходов к решению этой проблемы — добыча спецификаций.

Статья получена 14 октября 2022.
Косьяненко Иван Александрович, РТУ МИРЭА (kosyanenko.edu@gmail.com)
Болбаков Роман Геннадьевич, РТУ МИРЭА (bolbakov@mirea.ru)

***Ключевые слова* —автоматическая генерация сообщений к коммитам, добыча спецификаций, конечный автомат, спецификация ПО, формальная спецификация.**

I. Введение

Для любого инструмента или алгоритма, претендующего на автоматическую генерацию сообщений к коммитам, необходима “обучающая «выборка данных. Такая выборка должна содержать наборы размеченных частей программ (или программного кода) и их описания на естественном языке. Действительно, задача получения описания работы программы на естественном языке (ее также называют задачей обобщения кода) является достаточно сложной [1], но без ее решения задача генерации сообщений к коммитам усложняется еще больше.

Более того, задача генерации сообщений к коммитам усложняется большим количеством языков программирования. Подразумевается, что инструмент генерации сообщений к коммитам должен быть способен обрабатывать любой существующий высокоуровневый язык, что, в свою очередь, требует гигантского размера обучающей выборки данных.

Из этого следует вывод о необходимости наличия возможности сведения программы на любом языке программирования к единой нотации. Такой подход позволит применять для задачи генерации сообщений к коммитам sequence-to-sequence модели [2] с обучающим набором данных гораздо меньшего размера.

Теперь ответим на вопрос — какие существуют нотации для описания действия программного кода? Обработка абстрактного синтаксического дерева (АСД) в любом случае подразумевает обучение модели на конкретном языке (иначе бы могли взять компилятор от одного языка программирования и применить его к другому языку) [3].

В качестве такой единой нотации, позволяющей облегчить задачу верификации и тестирования кода, могут быть рассмотрены **формальные спецификации** [4]. В настоящее время далеко не все программы содержат формальную спецификацию. Более того, программисты неохотно пишут их, поскольку считают, что временные затраты на описание формальных спецификаций программ больше, чем получаемая польза. Одним из эффективных подходов к решению этой проблемы является добыча формальных спецификаций — то есть их вывод из артефактов,

которые создают программисты. В артефактах заложена богатая информация, потенциально полезная для инструментов анализа кода.

II. Методы добычи спецификаций

A. Обзор

В информатике **формальная спецификация** — это математическое описание программной или аппаратной системы, которая может быть реализована в соответствии с этим описанием. Формальная спецификация отвечает на вопрос «что делает программный компонент?» [5]. Существует множество нотаций формальных спецификаций: Z [6], Larch [7], ДРАКОН [8] и др. Говоря про задачу выбора набора данных для sequence-to-sequence задач, выбор нотации формальной спецификации не имеет значения, главное, чтобы она была единственной. Такой монолингвистический набор данных позволит существенно повысить эффективность инструмента генерации сообщений к коммитам [9].

Хотя формальные спецификации и являются важной частью программной инженерии, в своей работе программисты формируют их очень редко. Такой вывод можно сделать на основании анализа популярных репозиториев программного обеспечения с открытым исходным кодом. Кроме того, даже если программное обеспечение имеет формальную спецификацию, она имеет свойство устаревать с эволюцией программного продукта.

Тем не менее, с динамичным развитием области нейронных сетей и машинного обучения активно решается задача добычи спецификаций. Особенно хорошо с указанной задачей справляются методы, основанные на **глубоком обучении.**

B. Метод добычи спецификаций, основанный на глубоком обучении

Так, например, профессорами Сингапурского Университета менеджмента был предложен подход, названный DSM (deep specification miner) [10]. Подход заключается в применении алгоритмов глубокого обучения на следах выполнения программ (execution traces). В подходе авторов в качестве исходного материала берется класс целевой библиотеки A, а для создания тысяч тестовых примеров используется автоматизированный инструмент генерации. Целью процесса генерации тестовых примеров является захват более богатого набора допустимых последовательностей вызываемых методов A. Далее алгоритм проводит глубокое обучение на следах выполнения сгенерированных тестовых примеров с целью подбора весов рекуррентной нейросетевой модели (RNNLM) [11]. После этого строится префиксное дерево (PTA) [12] и используется обученная языковая модель для извлечения ряда особенностей (функционала) из узлов PTA. Эти особенности затем используются в кластеризационных алгоритмах для объединения похожих состояний (т.е. узлов PTA). Результатом применения алгоритма кластеризации является простой и обобщенный конечный автомат, который отражает следы выполнения обучения. Наконец, подход предсказывает точность построенных ФСА (сгенерированных различными алгоритмами кластеризации с учетом различных настроек) и выводит тот, который имеет наибольшее предсказанное значение F-меры [13].

На Рисунке 1 представлены все шаги рассматриваемого алгоритма. В нем три главных процесса — генерация тестов, применение языковой модели и построение конечного автомата.

Процесс генерации тестов играет важную роль, так как на основе результатов его работы будет исполняться языковая модель. На данном этапе авторы обсуждаемой работы предлагают использовать инструменты для автоматической генерации тестов, такие как Randoop [14].

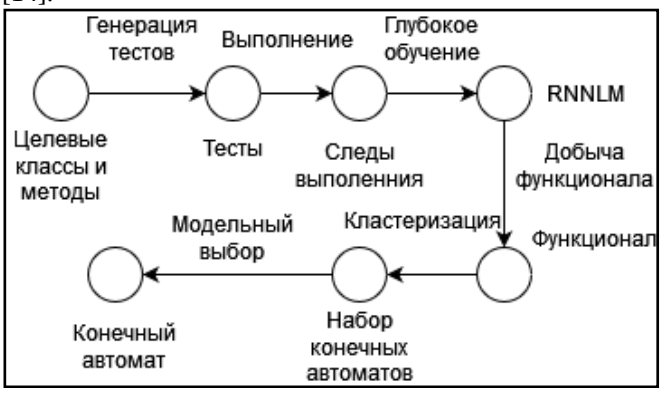

Рисунок 1 — Шаги алгоритма добычи спецификаций

Исполнение тестов позволяет собрать “следы” (traces) выполнения. Как правило, набор следов выполнения имеет большой размер, и часто эти следы пересекаются, то есть покрывают один и тот же набор команд. Поэтому в подходе предусмотрен процесс отбора следов исполнения. Алгоритм такого отбора представлен на **Рисунке 2.**

Ввод: $S = \{S_i \mid 1 \le i \le N\}$: набор следов исполнения, где N — число следов исполнения.
Вывод: O: Выбранные следы исполнения

1. Сортировка S в порядке возрастания длинны следа исполнения
2. $D \leftarrow$ инициализация словаря
3. $P \leftarrow$ пустое множество
4. $for\ S_i \in S$ ***do***
5. $\quad for\ (a,b)\ where\ a \in S_i \wedge b \in S_i \wedge a < b$ ***do***
6. $\quad\quad D[(a,b)] += [S_i]$
7. $\quad\quad$ Добавить (a,b) в P
8. $\quad$ ***end***
9. ***end***
10. $O \leftarrow$ пустое множество
11. $while\ \exists (a,b) \in P\ do$
12. $\quad$ Выбрать $(a,b) \in P$ такие, что $D[(a,b)]$имеют наименьшее
13. $\quad$ количество элементов
14. $\quad$ Найти $S_i \in D[(a,b)]$ при условии что S_i самая короткая
15. $\quad$ последовательность и $S_i \notin O$
16. $\quad$ Добавить S_i в O и удалить все пары (a,b) из P где
17. $\quad a \in S_i \wedge b \in S_i \wedge a < b$
18. **end**
19. **return** O

Рисунок 2 — Алгоритм отбора следов исполенния программы.

Сначала определяются такие пары *(a,b)*, где *a* и *b* являются методами, которые встречаются вместе хотя бы в одном следе *S* и добавляются в множество *P*. Затем создается пустое множество *O*. После итеративно выбираются пары *(a, b)*, которые не встречаются в множестве O и меньше всего встречаются в S. Обрабатывая пару *(a,b)*, алгоритм ищет самый короткий след $Si \notin O$ где $a,b \in Si$. Как только след найден, он

добавляется в множество *O*. Поиск продолжается до тех пор, пока O не охватит все пары совпадений *(a,b)* во входных данных.

Далее следует процесс работы рекуррентной нейронной сети. Он состоит из двух этапов:

1. **Построение обучающего набора данных**. На прошлом этапе алгоритмом были отобраны последовательности следов исполнения программы. Для их подачи в рекуррентную нейронную сеть их необходимо подготовить: каждая последовательность "обрамляется" специальными символами <START> и <END>, означающими начало и конец последовательности соответственно.
2. **Обучение модели**. Собранный набор данных используется для обучения языковой модели. Авторы обсуждаемой работы используют нейронную сеть, основанную на архитектуре "длинная цепь элементов краткосрочной памяти" (LSTM) [15].

В результате данного процесса получается нейронная сеть, обученная для добычи спецификаций.

На этапе **построения конечного автомата** подход, предлагаемый авторами, принимает на вход набор следов исполнения программ и обученную языковую модель на основе рекуррентной нейронной сети. Этап предполагает несколько шагов:

- **Извлечение особенностей (функционала)**. На основе последовательности следов исполнения строится акцептор дерева исполнения (PTA). Акцептор префиксного дерева (PTA) — это древовидный детерминированный конечный автомат, построенный на основе обучающей выборки путем принятия всех префиксов в выборке в качестве состояний. Конечными состояниями построенных PTA являются состояния, имеющие входящие ребра с метками <END>. На рисунке 2 изображен пример акцептора префиксного дерева [10].

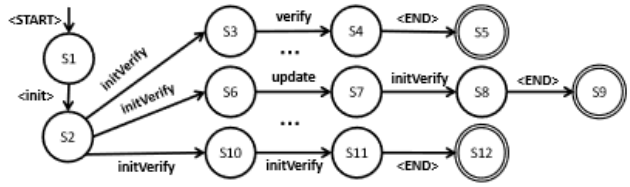

Рисунок 2 — Пример префиксного дерева.

- **Кластеризация**. Отобранные на прошлом шаге особенности обрабатываются с помощью метода k-средних (k-means) [16] и иерархической кластеризации. Целью данного шага является создание простого и обобщенного автомата, способного обрабатывать спецификации целевого класса библиотек.
- **Выбор модели**. На данном шаге выбирается лучший конечный автомат из множества всех автоматов, полученных на этапе кластеризации.

Результатом подхода, предложенным авторами обсуждаемой работы по добыче спецификаций, является **конечный автомат**, отображающую работу программного средства. Но для подходов к генерации сообщений к коммитам, основанным на нейронном машинном переводе, передавать конечный автомат на вход модели не представляется возможным. Поэтому предлагаемый авторами алгоритм можно доработать для решаемой задачи. Например, перевести конечный автомат в озвученную ранее нотацию формальных спецификаций Z. В работе [18] автор предлагает инструмент для формирования из нотации Z конечного автомата, но и обратный процесс представляется возможным.

C. Метод добычи спецификаций, основанный на сопоставлении шаблонов

Параллельно подходам к добыче спецификаций, основанным на нейронных сетях и глубоком обучении, развиваются также и другие подходы. Например, исследователи из университета Беркли предложили свой алгоритм по добыче спецификаций, основанный на сопоставлении шаблонов [19]. Высокоуровневая архитектура разработанного авторами упомянутой работы программного средства представлена на Рисунке 3.

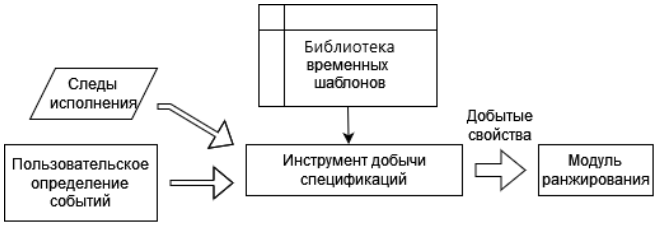

Рисунок 3 — Архитектура инстурмента добычи спецификаций.

Программное средство принимает на вход набор следов исполнения программы и, опционально, описанные пользователем события, и генерирует набор поведенческих шаблонов, удовлетворяющих следам исполнения, в качестве выходных данных.

Шаблон спецификации — это синтаксически ограниченная формула или автомат. Шаблоны спецификаций опираются на два важных определения:

- **Событие** (event) — кортеж вида $(v, \dot{v})$, где v — массив булевых переменных, а $\dot{v}$ — присвоенное значение каждой булевой переменной в массиве.
- **Дельта-событие** (delta-event) — обозначает некоторое изменение значения v, описывается формулой линейной временной логики (LTL) (1) [20]

$$(v) = ((v = \dot{v}_1) \quad \wedge (X(v = \dot{v}_2) \quad \wedge (\dot{v}_1 \neq \dot{v}_2)) \quad (1)$$

Авторы выделяют три вида поведенческих шаблонов спецификаций:

1. Чередование (alternating) — шаблон регулярного выражения $(\delta(a)\ \delta(b))*$ определяет, что изменение значение сигнала a всегда должно чередоваться с изменением значения сигнала b.
2. Пока не (until) — выявленный LTL-шаблон $\boldsymbol{G}\ (\delta(a) \wedge \neg\delta(a) \rightarrow X\dot{a}_{\delta}^{+} \cup\ \delta(b)))$ означает, что всякий раз, когда сигнал a изменяет значение, он должен сохранять это значение до тех пор, пока другой сигнал b не изменит значение.
3. В итоге (eventually) — выявленный LTL-шаблон $G\ (\delta(a)\ \rightarrow\ X\ (F\ \delta(b)))$ описывает, что за

изменением значения a в итоге должно последовать изменение значения b.

Таким образом, имея шаблон спецификации ξ^p, определенный на алфавите шаблонов Σ^p и некоторое доказательство ev над разделенным алфавитом Σ, проблема поиска спецификации состоит в том, чтобы найти все возможные полные и единственные отображения $\kappa : \Sigma^p \to \Sigma$ такие, что доказательство удовлетворяет Σ^p с символами в Σ^p, замененными символами в Σ при отображении к.

Авторы рассматриваемой работы предлагают свой алгоритм добычи спецификаций, основанный на построении монитора для каждого экземпляра шаблона спецификаций. В качестве монитора может выступать детерминированный конечный автомат. Переходами в таком автомате являются LTL-формулы. Основной структурой данных, используемой алгоритмом, является таблица шаблонов T.

На первом шаге каждый след исполнения программы преобразуется в дельта-след. Дельта-след τ^{δ} — это след исполнения, в котором каждое состояние содержит набор дельта-событий E^{δ}, которые происходят (истинны) в этом состоянии. Таким образом резко снижается количество данных, подаваемых алгоритму на вход, что важно в контексте решаемой задачи, так как она является NP-полной и выводится из задачи поиска гамильтонова пути [21].

Монитор (детерминированный конечный автомат), обрабатывая последовательность дельта-следов, производит некую последовательность нулей и единиц, в зависимости от соответствия следов исполнения выявленным ранее шаблонам. Пример такого монитора представлен на Рисунке 4 [19]

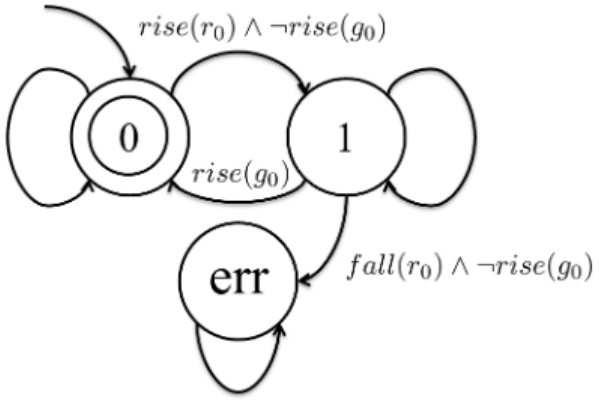

Рисунок 4 — Пример детерминированного конечного автомата, рассматриваемого в алгоритме

Полученная информация используется для записи количества вхождений шаблона, связанного с конкретным экземпляром шаблона. Дельта-событие, которое вызывает переход из состояния "0" в состояние "1" или из состояния "1" в состояние "0", является составляющим событием шаблона. В примере, изображенном на Рисунке 4, дельта-события $rise(g_1)$ и $rise(r_0)$ являются составными событиями паттерна “Пока не”, описываемого LTL-формулой ранее. Циклы, в которых происходит составляющее событие, записываются как временные метки для соответствующего события, если соответствующая спецификация не нарушена следом исполнения программы.

Помимо этого, необходимо создать таблицу T с размерами $|E^{\delta}| \times |E^{\delta}|$ такую, что каждая запись (i, j) в таблице описывает текущее состояние соответствующего монитора для экземпляра ξ^p с отображением к, которое отображает $a \in \Sigma^p$ на $e_i^{\delta} \in \Sigma^{\delta} = \Sigma$ и $b \in \Sigma^p$ на $e_j^{\delta} \in \Sigma^{\delta} = \Sigma$. Описываемая идея продемонстрирована на Рисунке 5.

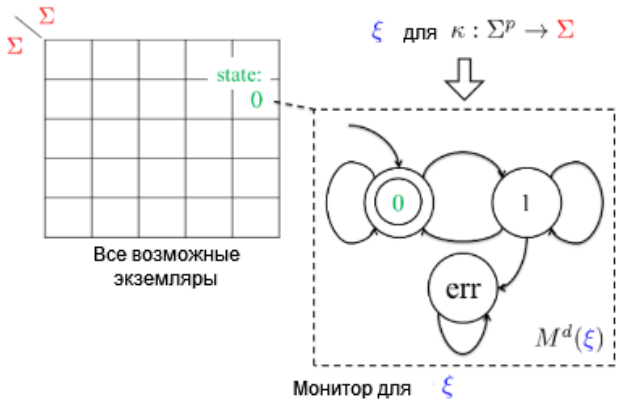

Рисунок 5 — Пример таблицы шаблонов.

Дельта-след τ^{δ} обрабатывается последовательно, начиная с τ_0^{δ}. Для каждого состояния τ_i^{δ} обновляются соответствующие записи в таблице, т.е. мониторы осуществляют переходы состояний в соответствии с τ_i^{δ}. В частности, если дано дельта-событие $e^{\delta} \in \tau_i^{\delta}$, то сначала обновляется каждая запись с отображением ρ, которое отображает a на e^{δ}. Это соответствует строке в таблице. Затем обновляется каждая запись с отображением к, которое отображает b в e^{δ}. Это соответствует столбцу в таблице. Алгоритм избегает многократных обновлений одной и той же записи в одном цикле выполнении поочередного обновления строк и столбцов. Каждая запись в таблице, не заканчивающаяся на состоянии "err", соответствует спецификации-кандидату, которая и подается на выход алгоритма. Алгоритм на псевдоязыке представлен на Рисунке 6.

Ввод: конечный след исполнения τ
Ввод: множество событий E в следе τ
Ввод: шаблон спецификации ξ^p
Вывод: Набор вероятных свойств Ξ как инстанций ξ^p.
1. $(\tau^{\delta}, E^{\delta})$ = ДельтаСобытие(τ, E)
2. Таблица = создатьТаблицу(E^{δ}, ξ^p)
3. Для каждого состояния τ_i^{δ} делать:
4. обновитьТаблицу(Таблица, τ_i^{δ})
5. закончить
6. Ξ = выходнойШаблон(Таблица)

Рисунок 6 — Алгоритм добычи спецификаций на псевдоязыке.

Таким образом, описанный авторами алгоритм выдает

на выходе набор LTL формул-паттернов спецификаций. Такие паттерны также можно применить в задаче генерации сообщений к коммитам. Так, например, новое исследование [22] показывает, как при помощи алгоритмов нейронного машинного перевода перейти от LTL-формул к конечным последовательностям символов. Именно такие последовательности необходимы для подходов к автоматической генерации сообщений к коммитам, основанным на глубоком обучении.

III. Заключение

Гипотеза применения формальных спецификаций для генерации сообщений к коммитам является актуальной, но на данный момент малоизученной. В подходах к генерации сообщений к коммитам, основанным на нейронном машинном переводе и глубоком обучении [23], сведение программного кода, к которому генерируется коммит, к формальной спецификации может значительно снизить объем обучающего набора данных для модели. Более того, наличие так называемого “синтаксического сахара” в языках программирования может значительно снизить качество генерируемых сообщений к коммитам, а сведение программного кода к формальным спецификациям, наоборот, позволит избежать этой проблемы. Задача обработки полученных описанными в статье алгоритмами спецификаций методами нейронного машинного обучения подлежит дальнейшему изучению.

Библиография

[1] Chen F., Kim M., Choo J. Novel Natural Language Summarization of Program Code via Leveraging Multiple Input Representations //Findings of the Association for Computational Linguistics: EMNLP 2021. – 2021. – С. 2510-2520

[2] Sutskever I., Vinyals O., Le Q. V. Sequence to sequence learning with neural networks //Advances in neural information processing systems. – 2014. – Т. 27

[3] Choi Y. S. et al. Learning Sequential and Structural Information for Source Code Summarization //Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021. – 2021. – С. 2842-2851.

[4] Бурдонов И. Б., Демаков А. В., Косачев А. С., Максимов А. В., Петренко А. К. Формальные спецификации в технологиях обратной инженерии и верификации программ // Труды ИСП РАН. 2000. №. URL: https://cyberleninka.ru/article/n/formalnye-spetsifikatsii-v-tehnologiyah-obratnoy-inzhenerii-i-verifikatsii-programm (дата обращения: 25.09.2022).

[5] Lamsweerde A. Formal specification: a roadmap //Proceedings of the Conference on the Future of Software Engineering. – 2000. – С. 147-159.

[6] Bowen J. P. Formal specification and documentation using Z: A case study approach. – London : International Thomson Computer Press, 1996. – Т. 66.

[7] Guttag J. V., Horning J. J., Wing J. M. The Larch Family of Specification Languages //IEEE Softw. – 1985. – Т. 2. – №. 5. – С. 24-36.

[8] Монастырная В. С., Фролов В. В. Визуальный язык Дракон и его применение //Актуальные проблемы авиации и космонавтики. – 2016. – Т. 2. – №. 12. – С. 78-79.

[9] Sennrich R., Haddow B., Birch A. Improving neural machine translation models with monolingual data //arXiv preprint arXiv:1511.06709. – 2015.

[10] Le T. D. B., Lo D. Deep specification mining //Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. – 2018. – С. 106-117.

[11] Mikolov T. et al. Recurrent neural network based language model //Interspeech. – 2010. – Т. 2. – №. 3. – С. 1045-1048.

[12] Тищенко В. А. Сжатое по путям префиксное дерево как основа классификатора по лексикографическому признаку //Материалы XXXIII Международной научно-практической конференции" Advances in Science and Technology". – 2020. – С. 129.

[13] Chen T. Y., Kuo F. C., Merkel R. On the statistical properties of the f-measure //Fourth International Conference onQuality Software, 2004. QSIC 2004. Proceedings. – IEEE, 2004. – С. 146-153.

[14] Robinson B. et al. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs //2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). – IEEE, 2011. – С. 23-32.

[15] Hochreiter S., Schmidhuber J. Long short-term memory //Neural computation. – 1997. – Т. 9. – №. 8. – С. 1735-1780.

[16] McQueen J. Some methods for classification and analysis of multivariate observations //Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1967. – 1967. – С. 281-297.

[17] Rokach L., Maimon O. Clustering methods //Data mining and knowledge discovery handbook. – Springer, Boston, MA, 2005. – С. 321-352.

[18] Khalid S., Nadeem A. Automated generation of finite state machine from object-oriented formal specifications //2010 6th International Conference on Emerging Technologies (ICET). – IEEE, 2010. – С. 304-309.

[19] Li W. Specification mining: New formalisms, algorithms and applications. – University of California, Berkeley, 2013. — С.25-42.

[20] Bauer A., Leucker M., Schallhart C. Comparing LTL semantics for runtime verification //Journal of Logic and Computation. – 2010. – Т. 20. – №. 3. – С. 651-674.

[21] Gabel M., Su Z. Symbolic mining of temporal specifications //Proceedings of the 30th international conference on Software engineering. – 2008. – С. 51-60.

[22] Cherukuri H., Ferrari A., Spoletini P. Towards Explainable Formal Methods: From LTL to Natural Language with Neural Machine Translation //International Working Conference on Requirements Engineering: Foundation for Software Quality. – Springer, Cham, 2022. – С. 79-86.

[23] Косьяненко И. А., Болбаков Р. Г. Об автоматической генерации сообщений к коммитам в системах контроля версий //International Journal of Open Information Technologies. – 2022. – Т. 10. – №. 4. – С. 55-60.

Иван Александрович КОСЬЯНЕНКО, аспирант кафедры инструментального и прикладного программного обеспечения РТУ МИРЭА
email: kosyanenko.edu@gmail.com

Роман Геннадьевич БОЛБАКОВ, заведующий кафедрой инструментального и прикладного программного обеспечения РТУ МИРЭА
email: bolbakov@mirea.ru

Mining of formal software specifications for further use in automatic commit message generation

I.A. Kosyanenko, R.G. Bolbakov

***Abstract* — In today's team-based software development, good commit messages - comments on changes made in natural language - are essential. The metric for evaluating a commit message is its relevance. A good commit message should not only describe the changes made, but also their context.**

With the growing popularity of version control systems, the number of studies aimed at creating tools for automatic generation of commit messages has increased. The most promising for this task are methods based on deep learning and neural machine translation. Such methods have program code or something to which it can be reduced as input. The only possible way to automatically generate a message for a commit is to "look" at the history of changes (diff). All modern version control systems provide the user with such a history, and special algorithms have been designed to compare changes between two versions of program code.

But there are a great many programming paradigms, even more there are programming languages with their own features and syntax. To make a tool for generating commit messages universal, it is necessary to be able to reduce the program code in any programming language to a single "notation".

One possible example of such notation is formal specifications. A specification shows what a program should do. There are many ways to write formal specifications, from algebraic expressions and specification languages to deterministic finite state machines. Since a commit message should give an indication of what and how the new software version has changed, the idea of using program specifications in commit message generation tools seems very rich.

Currently, most programs do not have a formal specification. It is believed that the time spent on elaborating specifications is not worth the result. One of the effective approaches to solve this problem is specification mining.

***Keywords*—automatic commit message generation, finite state machine, formal specification, specification mining, software specification.**

REFERENCES

[1] Chen F., Kim M., Cho J. Novel Natural Language Summary of Program Code via Leveraging Multiple Input Representations //Findings of the Association for Computational Linguistics: EMNLP 2021. – 2021. – pp. 2510-2520

[2] Sutzkever I., Vinyals O., Le Q. V. Sequence to sequence learning with neural networks //Advances in neural information processing systems. – 2014. – Vol. 27

[3] Choi Y. S. et al. Learning Sequential and Structural Information for Source Code Summarization //Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021. – 2021. – pp. 2842-2851.

[4] Burdonov I. B., Demakov A.V., Kosachev A. S., Maksimov A.V., Petrenko A. K. Formal specifications in reverse engineering and program verification technologies // Proceedings of the ISP RAS. 2000. no. URL: https://cyberleninka.ru/article/n/formalnye-spetsifikatsii-v-tehnologiyah-obratnoy-inzhenerii-i-verifikatsii-programm (accessed: 25.09.2022) (in russian).

[5] Lamsweerde A. Formal specification: a roadmap //Proceedings of the Conference on the Future of Software Engineering. - 2000. – pp. 147-159.

[6] Bowen J. P. Formal specification and documentation using Z: A case study approach. – London : International Thomson Computer Press, 1996. – Vol. 66.

[7] Guttag J. V., Horning J. J., Wing J. M. The Larch Family of Specification Languages //IEEE Softw. – 1985. – Vol. 2. – No. 5. – pp. 24-36.

[8] Monastyrnaya V. S., Frolov V. V. Visual language Dragon and its application //Actual problems of aviation and cosmonautics. – 2016. – Vol. 2. – no. 12. – pp. 78-79 (in russian).

[9] Sennrich R., Haddow B., Birch A. Improving neural machine translation models with monolingual data //arXiv preprint arXiv:1511.06709. – 2015.

[10] Le T. D. B., Lo D. Deep specification mining //Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. – 2018. – pp. 106-117.

[11] Makarov T. et al. Recurrent neural network based language model //Interspeech. – 2010. – Vol. 2. – No. 3. – pp. 1045-1048.

[12] Tishchenko V. A. Path-compressed prefix tree as the basis of a classifier by lexicographic feature //Materials of the XXXIII International Scientific and Practical Conference "Advances in Science and Technology". – 2020. – p. 129. (in russian)

[13] Chen T. Y., Cuo F. C., Marcel R. On the statistical properties of the f-measure //Fourth International Conference on Quality Software, 2004. QSIC 2004. Proceedings. – IEEE, 2004. – pp. 146-153.

[14] Robinson B. et al. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs //2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). – IEEE, 2011. – pp. 23-32.

[15] Hochreiter S., Schmidhuber J. Long short-term memory //Neural computing. – 1997. – Vol. 9. – No. 8. – pp. 1735-1780.

[16] McQueen J. Some methods for classification and analysis of multivariate observations //Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1967. – 1967. – pp. 281-297.

[17] Rokach L., Maison O. Clusteringmethods //Data mining and knowledge discovery handbook. – Springer, Boston, MA, 2005. – pp. 321-352.

[18] Khalid S., Nadeem A. Automatic generation of finite state machine from object-oriented format specifications //2010 6th International Conference on Emerging Technologies (ICET). – IEEE, 2010. – pp. 304-309.

[19] Li W. Specification mining: New formalisms, algorithms and applications. – University of California, Berkeley, 2013. — pp.25-42.

[20] Bayer A., Lucker M., Schallhart C. Comparing LTL semantics for runtime verification //Journal of Logic and Computation. – 2010. – Vol. 20. – No. 3. – Pp. 651-674.

[21] Label M., Su Z. Symbolic meaning of temporal specifications //Proceedings of the 30th international conference on Software engineering. - 2008. – pp. 51-60.

[22] Cherukuri H., Ferrari A., Spoletini P. Towards Explainable Formal Methods: From LTL to Natural Language with Neutral Machine Translation //International Working Conference on Requirements

Engineering: Foundation for Software Quality. – Springer, Cham, 2022. – pp. 79-86.

[23] Kosyanenko I. A., Bolbakov R. G. On automatic generation of commit messages in version control systems //International Journal of Open Information Technologies. – 2022. – Vol. 10. – No. 4. – pp. 55-60. (in russian)

Ivan KOSYANENKO, Postgraduate student of the Department of Instrumental and Applied Software of RTU MIREA
email: kosyanenko.edu@gmail.com
Roman BOLBAKOV, Head of the Department of Instrumental and Applied Software of RTU MIREA email: bolbakov@mirea.ru