Journal of Soft Computing and Information Technology (JSCIT)
Babol Noshirvani University of Technology, Babol, Iran
Journal Homepage: jscit.nit.ac.ir
Volume 12, Number 3, Autumn 2023, pp. 74-84
Received: 06/13/2020, Revised: 03/24/2023, Accepted: 01/07/2024

Automatic test data generation to improve fault-localization based on causal-statistical analysis

Morteza Zakeri-Nasrabadi[1], Saeed Parsa[2*], Zahra Hayati[3]
1- School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.
2*- School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.
3- School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.

[1]morteza_zakeri@comp.iust.ac.ir, [2*]parsa@iust.ac.ir, [3]z_hayati@comp.iust.ac.ir

Corresponding author's address: S. Parsa, School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran, Iran.

Abstract— The statistical-based software fault localization approaches highly depend on the program inputs and become unstable as input data changes. Therefore, generating appropriate test data plays an essential role in the quality of the software fault-localization process. This paper presents an approach to improving program fault localization by generating new test data. The minimized test suite determines the faulty execution path and the fault suspiciousness of each statement in the path is generated. First, the suspicious statements in the faulty path are determined. To this aim, the conditions of the faulty execution path are contradicted from the end to the beginning, and test data is created for the desired path using the Z3 solver. Afterward, the program under test is executed with the generated test data using the Concolic testing technique. The fault-suspicious branch is determined depending on the passing or failing of the program execution. As a result, the region of statements for applying the causal-statistical approach is minimized. The proposed approach is evaluated on the four projects in the Defects4J benchmark. The results show that 75% of faults are localized by examining a maximum of 1% of the program's source code. Compared to the related work, the results have improved by 17.98%. Moreover, the mean number of sentences examined for fault localization decreases by 16.78% in the worst case.

Keywords— Debugging, Software Fault Localization, Statistical Causal Analysis, Test Data Generation, Concolic Execution.

مجله علمی پژوهشی رایانش نرم و فناوری اطلاعات
دانشگاه صنعتی نوشیروانی بابل
صفحه مجله: jscit.nit.ac.ir
جلد ۱۲، شماره ۳، پائیز ۱۴۰۲ ، صفحه ۷۴-۸۴
دریافت: ۱۳۹۹/۰۳/۲۴، بازنگری: ۱۴۰۲/۰۱/۰۴، پذیرش: ۱۴۰۲/۱۰/۱۷

تولید خودکار مجموعه داده آزمون با هدف بهبود مکان‌یابی خطا مبتنی بر تحلیل علّی-آماری

مرتضی ذاکری نصرآبادی[۱]، سعید پارسا*[۲]، زهرا حیاتی
۱- دانشکده مهندسی کامپیوتر، دانشگاه علم و صنعت ایران، تهران، ایران.
*۲- دانشکده مهندسی کامپیوتر، دانشگاه علم و صنعت ایران، تهران، ایران.
۳- دانشکده مهندسی کامپیوتر، دانشگاه علم و صنعت ایران، تهران، ایران.

[1]morteza_zakeri@comp.iust.ac.ir, [2*]parsa@iust.ac.ir, [3]z_hayati@comp.iust.ac.ir

* نشانی نویسنده مسئول: سعید پارسا، دانشکده مهندسی کامپیوتر، دانشگاه علم و صنعت ایران، خیابان هنگام، میدان رسالت، تهران، ایران.

چکیده— روش‌های آماری مکان‌یابی خطا در نرم‌افزار وابستگی زیادی به داده‌های ورودی برنامه داشته و با تغییرات داده‌ها دچار ناپایداری می‌شوند. از این رو، تولید داده آزمون مناسب نقش کلیدی در کیفیت فرآیند مکان‌یابی خطای نرم‌افزار ایفاء می‌کند. در این مقاله، روشی برای بهبود مکان‌یابی خطا با تولید داده‌های آزمون جدید ارائه می‌شود. مجموعه آزمون به صورت کمینه و هدفمند، جهت تعیین شاخه خطادار و پس از آن جملات مظنون به خطای درون شاخه، تولید می‌گردد. ابتدا، محدوده جملات مظنون به خطا در یک مسیر اجرایی مشخص می‌شود. برای این کار، در مسیر اجرایی خطادار، شرط‌ها از انتها به ابتدا نقیض شده و با استفاده از حل‌کننده *Z3* داده آزمون برای مسیر مورد نظر ایجاد می‌گردد. سپس، برنامه مجدداً با داده آزمون‌های به دست آمده توسط فن اجرای نمادین پویا اجرا می‌شود. با توجه به موفق و یا ناموفق بودن اجرا، مشخص می‌کنیم که کدام شاخه مظنون به خطا است. بدین ترتیب، محدوده جملات برای اعمال روش علّی-آماری به حداقل ممکن می‌رسد. ارزیابی روش پیشنهادی روی چهار پروژه از مجموعه محک Defect4J، انجام شده است. نتایج نشان دهنده کشف ۷۵% از خطاها با بررسی حداکثر یک درصد از کد این برنامه‌ها است که در مقایسه با کارهای موجود ۱۷/۹۸% بهبود دارد. همچنین، متوسط جملات مورد بررسی جهت کشف خطا، در بدترین حالت به میزان ۱۶/۷۸% کاهش داشته است.

واژه‌های کلیدی: اشکال‌زدایی، مکان‌یابی خطای نرم‌افزار، تحلیل علّی-آماری، تولید داده آزمون، اجرای نمادین پویا.

۱- مقدمه

مکان‌یابی خودکار خطای برنامه‌های نرم‌افزاری، یک ایده‌آل در مهندسی نرم‌افزار است. لازمه آن تولید خودکار داده‌های آزمون مؤثر برای یافتن مکان خطا است [۱]. به‌منظور تشخیص مکان خطاهای پنهان در کد، باید داده‌های آزمون را به قسمی تعیین نمود که کلیه مسیرهای اجرایی برنامه را پوشش دهند. با مشخص شدن اجراهای خطا دار، بر اساس آمار حضور جملات در آنها، میزان مظنون به خطایی هر جمله تعیین می‌گردد. پژوهش‌های اخیر نشان از برتری روش‌های آماری نسبت به سایر روش‌ها دارد [۲]–[۵]. مشکل عمده این روش‌ها، وابستگی آنها به داده‌های آزمون است [۲], [۶], [۷]؛ زیرا، این روش‌ها براساس آمار حضور جملات در اجراهای خطا دار اقدام به محاسبه میزان مظنون به خطایی جملات می‌کنند [۵], [۸]. بنابراین، تولید داده آزمون[۱] مناسب نقش کلیدی در فرآیند مکان‌یابی خطا ایفا می‌کند [۹], [۱۰]. روش‌های خودکار آزمون نرم‌افزار، داده‌های آزمون را مستقل از مکان‌یابی خطا تولید و صرفاً با هدف کشف خطاهای جدید تولید می‌کنند [۱۱]. در نتیجه، تمرکز آنها روی افزایش معیارهای کفایت آزمون مانند پوشش دستور، شاخه و امتیاز جهش است [۱۲]. در حالی که مکان خطا در صورتی

آشکار می‌شود که برنامه تحت آزمون با مقدار خاص و یا دامنه خاصی از ورودی اجرا شود [۱۳]، [۱۴]. تولید داده آزمونی که کمک به کشف مکان خطا نکند باعث اتلاف منابع می‌گردد. این مقاله نشان می‌دهد که چگونه می‌توان با استفاده از آزمون نمادین پویا [۱۱]، [۱۵] و یک حل‌کننده محدودیت، مثل Z3 [۱۶]، داده‌های آزمون را به‌صورت کمینه و هدفمند برای تشخیص محدوده خطا، تولید نمود.

پیچیدگی فرایند مکان‌یابی خطا در روش پیشنهادی، به سه پیچیدگی *تعیین مسیر اجرایی خطادار*، *تعیین شاخه مظنون به خطا* و *تعیین محدوده مظنون به خطا* تقسیم شده است. روش پیشنهادی در این مقاله تحت عنوان TD-CAFL[۲] با تمرکز بر تولید داده‌های آزمون جدید، در چهار گام اقدام به غلبه بر این پیچیدگی‌ها و مکان‌یابی خطا می‌کند. در گام اول، داده‌های آزمون با بهره‌گیری از ابزار آزمون نمادین پویای JDART [۱۷] تولید شده و مسیر اجرایی خطادار مشخص می‌شود. در گام دوم، شاخه اجرایی مظنون به خطا با تولید داده های آزمون هدفمند و شناسایی انواع اجراهای موفق، ناموفق و تصادفاً موفق معین می‌گردد. در این گام، جملات تأثیرگذار در اجرای خطادار، با استفاده از برش‌بندی پویا[۳]ی [۱۸] برنامه توسط ابزار JavaSlicer [۱۹] تعیین می‌شود. در برش پویای برنامه، تنها دستورات اجرا شده برای یک ورودی خاص وجود دارند [۱۸]. چالش اصلی در اینجا، وجود اجراهای تصادفاً موفق است که آنها را با توجه به میزان حضور جملات در برش اجراهای مختلف، مشخص می‌کنیم. در گام سوم، شاخه‌ی خطادار با استفاده از روند متوالی تولید داده آزمون برای هر محدودیت مسیر برنامه، اجرای برنامه و بررسی موفق بودن یا نبودن اجرا، تعیین می‌شود. در گام چهارم، با استفاده از تحلیل علّی-آماری، آمار حضور هر جمله در اجراهای موفق و ناموفق تحلیل شده و میزان مظنون به خطایی هر جمله مشخص می‌گردد. به‌طور خلاصه، نوآوری‌های روش پیشنهادی عبارتند از:

۱- تولید داده آزمون جدید به‌صورت متوالی با هدف تعیین شاخه خطادار؛

۲- تولید داده‌های آزمون جدید برای افزایش دقت تشخیص اجراهای تصادفاً موفق در روش‌های آماری مکان‌یابی خطا.

در ادامه این مقاله، در بخش ۲، مفاهیم اولیه و کارهای مرتبط خلاصه شده است. روش پیشنهادی در بخش ۳ بحث می‌شود. ارزیابی روش و نتیجه‌گیری به ترتیب در بخش‌های ۵ و ۶ بیان می‌شوند.

۲- کارهای مرتبط

هر قدر میزان حساسیت نرم‌افزار تحت آزمون بیشتر باشد، اهمیت مکان‌یابی خطا نیز افزایش می‌یابد. پژوهش‌های عمده‌ای برای بهبود و خودکارسازی فرآیند مکان‌یابی خطا صورت گرفته است، که می‌تواند توسعه‌دهندگان را در راستای یافتن محل خطا با کمترین مداخله انسانی یاری دهد [۵]. در ادامه این بخش، با توجه به خط مشی مقاله، کارهای مرتبط در زمینه مکان‌یابی خطا مبتنی بر تحلیل علّی-آماری و روش‌های مکان‌یابی خطا مبتنی بر تغییر وضعیت برنامه بحث خواهند شد.

۲-۱- روش‌های مکان‌یابی خطا مبتنی بر تحلیل علّی-آماری

مکان‌یابی خطا یک مسئله علّی است؛ زیرا، هدف مکان‌یابی خطا پیدا کردن علّت خرابی برنامه است. اکثر روش‌های آماری مکان‌یابی خطا در برآورد اثر علّی جملات بر خروجی ناموفق برنامه دارای سوگیری[۴] هستند [۵]. وجود سوگیری به خاطر همبستگی میان متغیرهای برنامه است. به متغیرهایی که باعث ایجاد همبستگی بین برخی متغیرهای دیگر می‌شوند، *متغیرهای اختلاطی* یا *متغیرهای پنهان* گفته می‌شود. در واقع، روش‌های آماری در محاسبه اثر دستور بر خروجی ناموفق، اثراتی که دستورات بر یکدیگر می‌گذارند را در نظر نمی‌گیرند. Baah و همکاران [۲۰] نشان دادند که می‌توان از ضابطه دربِ پشتی[۵] به منظور شناسایی متغیرهای اختلاطی استفاده کرد. این ضابطه بر روی گراف علّی حاصل از وابستگی‌های برنامه اعمال شده و مسیرهای جعلی که بین جملات برنامه با خروجی آن ایجاد شده است را شناسایی می‌کند. برای مسدود کردن این ارتباطات جعلی، با استفاده از قضیه جداپذیری مستقیم [۲۱] مجموعه جملاتی که سبب ایجاد ارتباطات جعلی شده‌اند ، شناسایی می‌شوند. با داشتن این مجموعه از جملات، اطمینان حاصل می‌شود که ارتباط اندازه‌گیری شده میان هر جمله با خروجی برنامه یک ارتباط علّی بوده و بیان‌گر اثر علّی یک جمله بر خروجی برنامه است [۲۱].

برخی ساختارها دستورات برنامه را تحت تأثیر قرار داده و سبب می‌شوند که راهکارهای SBFL آنها را به‌اشتباه طبقه‌بندی کنند. به‌عنوان نمونه، امتیاز یک جمله خطادار در کلاس اصلی می‌تواند کم برآورد شود؛ زیرا، همیشه توسط موردهای آزمون موفق و ناموفق اجرا می‌شود. در مقابل، امتیاز یک جمله بدون خطا در بلاک catch می‌تواند بیش برآورد شود؛ زیرا تنها در موردهای آزمون ناموفق اجرا می‌شود [۲۲].

برای غلبه بر این چالش، مکان‌یابی خطا با پیشبینی خطا ترکیب شده است. در این روش اطلاعاتی خطاخیزی هر دستور با تحلیل ایستا محاسبه شده و با اطلاعات حاصل از آمار اجرای دستور ترکیب می‌شود [۲]. اما، این راهکارها نیز همچنان وابسته به نمایه‌های آزمون هستند. تولید داده‌های آزمون جدید و بیشتر و تمرکز بر رتبه‌بندی دستورات داخل شاخه خطا دار می‌توان دقت مکان‌یابی را افزایش دهند.

۲-۲- روش‌های مکان‌یابی خطا مبتنی بر تغییر وضعیت

مناسب‌ترین فن برای یافتن علت‌های عدم موفقیت برنامه، انجام آزمایش‌هایی از طریق تغییر موجودیت‌های برنامه است. این تغییرات می‌تواند باعث شود تا علّت عدم موفقیت برنامه آشکار شود. روش‌های مکان‌یابی خطا مبتنی بر تغییر وضعیت برنامه از این رویکرد استفاده می‌کنند. Zhang و همکاران [۲۳] روشی به نام *جایگزینی گزاره*[۶] به منظور مکان‌یابی خطاهای مرتبط با گراف جریان کنترلی ارائه داده‌اند. در این روش، ابتدا برنامه توسط یک مورد آزمون که مشخص کننده خطا است، اجرا می‌شود. سپس، در زمان اجرا گزاره‌ها را نقیض می‌کنند. هدف یافتن گزاره‌ای است که اگر مقدار آن نقیض شود (از نادرست به درست یا بالعکس)، وضعیت خروجی برنامه از ناموفق به موفق تغییر پیدا می‌کند. این گزاره به صورت بالقوه علت خطا بوده و *گزاره بحرانی* نامیده می‌شود. روش ارائه شده از گزاره‌های بحرانی شروع کرده و یک برش دوجهته محاسبه می‌کند. این روش مشابه با روش مکان‌یابی خطا مبتنی بر جهش [۲۴] است؛ زیرا، هر دوی این روش‌ها در ابتدا جهش را به برنامه اعمال کرده و سپس نتیجه اجرا را بررسی می‌کنند. با این وجود، جایگزینی گزاره به عنوان یک روش مکان‌یابی خطای جدا درنظر گرفته می‌شود؛ چرا که در این روش، جهش تنها بر روی گراف جریان کنترلی اعمال می‌گردد. اشکال اساسی این روش بزرگ بودن اندازه برش محاسبه شده و رتبه‌بندی نشدن جملات موجود در آن است. لذا، روش جدیدی به نام *جایگزینی مقادیر*[۷] ارائه شده است [۲۵]. در این روش مقادیر تمام متغیرهای موجود در یک جمله در اجراهای موفق و ناموفق، در جدولی بنام جدول نگاشت ذخیره می‌گردند. سپس مقادیر این متغیرها در اجراهای ناموفق با مقادیر متناظرشان در اجراهای موفق جایگزین می‌شوند. اگر بعد از این جایگزینی اجرای ناموفق برنامه به اجرای موفق تبدیل شود، این زوج مقادیر به‌عنوان زوج مقادیر نگاشت مناسب (IVMP[8]) درنظر گرفته می‌شوند. سپس جملات متناظر با هر IVMP با فرمول روش تارانتولا [۲۶] امتیازدهی می‌گردد.

محدودیتی که در روش‌های تغییر وضعیت برنامه وجود دارد این است که با تغییر یک وضعیت از برنامه، این روش تضمین نمی‌کند که اجرای جدید به‌دست‌آمده یک اجرای واقعی باشد. منظور از اجرای واقعی، یعنی یک ورودی وجود دارد که بتوان برنامه را با آن اجرا کرد.

پژوهش‌های اخیر بر ترکیب روش‌های مختلف متمرکز شده‌اند. Dutta و همکاران [۸] سه روش آماری، یادگیری ماشینی و مبتنی بر جهش را ترکیب کرده و نشان داده‌اند که دقت مکان‌یابی ٪۲۸.۴۲ افزایش داشته است. هرچند در این روش‌ها نیز از یک مجموعه داده آزمون از پیش تهیه شده استفاده شده است و داده‌های آزمون جدید با هدف مکان‌یابی تولید نشده‌اند.

۳- روش پیشنهادی

روش پیشنهادی این مقاله، TD-CAFL، با رویکرد تولید داده‌های آزمون جدید جهت کاهش تعداد جملات مورد بررسی در مکان‌یابی خطا و همچنین افزایش دقت در محاسبه محدوده مظنون به خطا ارائه شده است. ورودی روش پیشنهادی، برنامه خطادار و مجموعه آزمون اولیه دارای یک مورد آزمون منجر به باز تولید خطا است. خروجی روش امتیاز مظنون به خطایی هر دستور از برنامه است. مهم‌ترین عملیاتی که در راستای کاهش تعداد جملات مورد بررسی و در حین حال افزایش دقت روش‌های مکان‌یابی انجام می‌شود، محاسبه محدوده مظنون به خطا با شناسایی اجراهای تصادفاً موفق است. روش پیشنهادی متشکل از چهار گام زیر است:

- **گام اول:** به دست آوردن مسیرهای اجرایی مختلف برنامه تحت آزمون به همراه داده آزمون متناظر با هر مسیر
- **گام دوم:** شناسایی اجراهای تصادفاً موفق
- **گام سوم:** محاسبه شاخه خطادار
- **گام چهارم:** برآورد اثر علّی جملات شاخه خطادار و تعیین امتیاز مظنون به خطایی برای هر دستور در شاخه خطا دار.

در ادامه این بخش، جزئیات هر گام توضیح داده می‌شود.

۳-۱- تعیین مسیرهای اجرایی مختلف برنامه تحت آزمون

برای بررسی رفتار زمان اجرای برنامه، برخی از فنون مکان‌یابی خطا، تمامی کد برنامه را خط به خط تحلیل نمی‌کنند. دلیل این امر سربار زمانی و فضایی است که جمع‌آوری و تحلیل کل جملات برنامه ممکن است ایجاد کند. بنابراین، برخی از نقاط برنامه که اغلب تعیین‌کننده مسیرهای برنامه هستند، مورد بررسی و تحلیل قرار می‌گیرند. برای هرکدام از این نقاط یک عبارت منطقی در نظر گرفته می‌شود. این عبارت منطقی یک *گزاره* نامیده می‌شود. در حقیقت گزاره‌ی برنامه یک عبارت منطقی با دو مقدار درست یا نادرست است. مقادیر گزاره‌ها رفتار زمان اجرای برنامه را انعکاس می‌دهند [۲۷].

گزاره‌ها در برنامه موجب تغییر روند اجرا در برنامه خواهند شد، پس می‌توان از این دستورات در مکان‌یابی خطا بهره برد. وجود گزاره‌های مختلف در برنامه باعث ایجاد مسیرهای اجرایی مختلف در برنامه خواهد شد. در اینجا ما برای به دست آوردن مسیرهای اجرایی مختلف براساس مسیر خطادار از مفهوم اجرای نمادین پویا و درخت محدودیت استفاده می‌کنیم [۱۷]. درخت محدودیت با برای یک محدودیت مسیر داده شده با استفاده از نقیض کردن یک به یک

محدودیت‌های آن مسیر ایجاد می‌گردد به نحوی که تمامی شاخه‌های حاصل از محدودیت مسیر پوشش داده شوند [۱۷].

برای تولید داده آزمون با هدف دست‌یابی به پوشش بالای کد ابتدا برنامه تحت آزمون را با یک ورودی تصادفی اجرا کرده سپس محدودیت مسیر برای این ورودی محاسبه می‌شود. بعد از محاسبه محدودیت مسیر برای ورودی مورد نظر، حال به منظور تولید داده آزمون مختلف برای برنامه باید محدودیت‌های مسیر دیگر نیز درنظر گرفته شود. برای انجام این کار براساس جست‌وجوی اول عمق، از روی محدودیت مسیر به دست آمده از مرحله قبل، یک محدودیت مسیر جدید تولید کرده و سپس با استفاده از حل‌کننده Z3 [۱۶]، برای این محدودیت مسیر، داده آزمون مورد نظر تولید می‌گردد. حل کننده محدودیت Z3، محدودیت مسیر حاصل از اجرای نمادین پویا را دریافت کرده و مقادیر متغیرهای آن را به نحوی پیدا می‌کند که این مسیر اجرا شود [۲۸]. این کار تا زمانی که تمام مسیرهای ممکن بررسی شده و داده برای آن‌ها تولید شده است ادامه می‌یابد.

ترکیب محدودیت‌های مسیر یک برنامه سبب ایجاد درخت محدودیت می‌شود که برای به دست آوردن محدودیت مسیر جدید یک برنامه، از آن استفاده می‌کنیم. در درخت محدودیت هر گزاره‌ای که تحت تأثیر داده ورودی است، یک گره از درخت محدودیت را تشکیل می‌دهد. هر گره با توجه به اینکه یک دستور شرطی است دو حالت دارد، که با توجه به خروجی شرط یا درست است یا نادرست.

الگوریتم شکل (۱)، نحوه ایجاد محدودیت‌های جدید و تولید متوالی موارد آزمون برای مسیرهای اجرایی را نشان می‌دهد. برای تشخیص مسیر خطا دار، ابتدا برنامه را با یک داده آزمون که می‌دانیم نتیجه آن ناموفق است، اجرا کرده و برای این داده آزمون محدودیت مسیر را به دست می‌آوریم (خطوط ۲ و ۳). سپس محدودیت‌های این مسیر به درخت محدودیت اضافه می‌کنیم (خطوط ۶ تا۱۱). در روش پیشنهادی فرض شده است، که مجموعه آزمون اولیه برای مکان‌یابی حاوی یک مورد آزمون ایجاد کننده خطا موجود است. به‌عبارت دیگر، این مجموعه آزمون حاصل فرایند آزمون نرم‌افزار است که وجود خطا در کد را با پوشش حداکثری برنامه تشخیص داده است. در ادامه الگوریتم از آخرین گزاره در درخت محدودیت شروع کرده (خط ۱۴)، هر بار آخرین گزاره از مجموعه گزاره‌هایی که بررسی نشده‌اند را منفی کرده، محدودیت مسیر جدید را تولید و آن را به درخت محدودیت اضافه می‌نماید (خطوط ۱۵ تا ۲۶). سپس محدودیت مسیر جدید به حل‌کننده داده می‌شود تا برای این محدودیت مسیر، داده آزمون تولید کند (خط ۵). در اینجا از حل‌کننده شناخته شده Z3 [۱۶] استفاده می‌کنیم. در ادامه، برنامه را با مورد آزمون جدید مجدداً به صورت نمادین پویا اجرا می‌کنیم و نتیجه موفق یا ناموفق بودن داده آزمون که توسط اوراکل مشخص می‌شود را در ثبت می‌کنیم. با توجه به مشخص بودن مورد آزمون خطادار، اوراکل، یعنی خروجی مورد انتظار کاربر از برنامه [۱۲]، برای سایر داده‌های آزمون توسط عامل انسانی تعیین می‌شود.

در هر تکرار الگوریتم (خط ۴)، درخت محدودیت بررسی شده و شاخه‌هایی که هنوز جستجو نشده‌اند، مشخص می‌گردند. مطابق با جست‌وجوی اول عمق، زمانی که همه شاخه‌های مربوط به یک گره بررسی شدند، به سراغ گره بالاتر رفته، شاخه‌های آن را بررسی می‌کنیم و محدودیت مسیر جدید را به دست می‌آوریم. پس از بررسی همه مسیرهای اجرایی برنامه، نتایج به دست آمده را به گام ۲ یعنی شناسایی موارد آزمون تصادفاً موفق ارسال می‌کنیم.

Algorithm TD-CAFL-Step1

```
Input: ProgramUnderTest, FailedTestData
Output: PathConstraint, TestSuite
// Create constraint tree with Concolic Execution
1.  stack = new Stack();
2.  InputData ← FailingTestData;
3.  PathConstraint ← ExecuteInJDART(
        InputData, ProgramUnderTest);
4.  while PathConstraint ! = Null do
5.      InputData ← Z3Solver(PathConstraint);
6.      ExePath ← ExecuteInJDART(
            InputData, ProgramUnderTest);
7.      if ExePath ∈ ExePathsPool then
8.          ExePathsPool.add(
              ExePath.getPredicates(),
              ExePath.failingOrPassingResult);
9.          foreach condition in ExePath do
10.             stack.push(condition, ExePath);
11.         end for
12.     end if
13.     PathConstraint ← Null;
14.     condtionsInExePath ← stack.pop();
15.     if condtionsInExePath ! = Null do
16.         ExePath ← condtionsInExePath.ExePath;
17.         predicate ← condtionsInExePath.condition;
18.         foreach condition in ExePath do
19.             if predicate == condition then
20.                 PathConstraint.add(~ condition);
21.                 break;
22.             else
23.                 PathConstraint.add(condition);
24.             end if
25.         end for
26.     end if
27. end while
```

شکل(۱): الگوریتم گام اول محاسبه مسیرهای اجرایی به همراه داده آزمون متناسب با هر یک

۳-۲- شناسایی اجراهای تصادفاً موفق

یک چالش مهم در مکان‌یابی آماری خطا، اجراهای تصادفاً موفق است. به موردی تصادفاً موفق گفته می‌شود که در اجرای برنامه با آن مورد آزمون، جمله خطادار اجرا شده است؛ ولی خروجی ناموفق مشاهده نشده است [۲۹], [۳۰]. اجراهای تصادفاً موفق عامل مهمی در کاهش عملکرد روش‌های آماری مکان‌یابی خطا هستند [۳۰]–[۳۲].

در این گام، به منظور شناسایی موارد آزمون تصادفاً موفق از روش مبتنی بر برش‌بندی استفاده می‌کنیم [۲۹]. این روش احتمال وجود پدیده موفقیت تصادفی در مورد اجراهای موفق را محاسبه می‌کند. به جای استفاده از برش ایستا [۳۳] از برش پویای پسرو استفاده می‌شود. برش پویای پسرو [۱۸] برای یک متغیر، تنها دستورات موجود در اجرای قبلی را با تحلیل دستورات برنامه از انتها به ابتدا، استخراج می‌کند که حاوی متغیر داده شده هستند. برش ایستای یک متغیر، کلیه دستورات حاوی نام متغیر داده شده را بدون توجه دستورات اجرا شده، در نظر می‌گیرد [۳۴] که در نتیجه دقت روش مکان‌یابی خطا کاهش می‌یابد. همچنین، داده‌های آزمون تولید شده در مرحله قبل نیز افزون بر مجموعه آزمون موجود در فرایند محاسبه برش‌ها در نظر گرفته می‌شوند که سبب افزایش تعداد اجراها و بیشتر شدن اطلاعات مربوط به آمار اجرای جملات می‌گردد.

ممکن است در برخی حالات، جمله خطادار در برش پویای محاسبه شده حضور نداشته باشد، در نتیجه، برش پویای محاسبه شده، باید به‌گونه‌ای بسط داده شود که شامل این جملات مظنون به خطا نیز باشد. در روش پیشنهادی، برای محاسبه مجموعه جملاتی که در تولید خروجی ناموفق نقش داشته‌اند، از برش‌بندی پویای پسرو و بسط برش پویا استفاده می‌شود. این مجموعه جملات، عوامل احتمالی خطا (FCC)[9] نامیده می‌شوند. برای شناسایی FCC ابتدا با استفاده از برش‌بندی، برش‌های پویای به دست آمده بسط داده می‌شوند. برای محاسبه برش پویای بسط یافته هر اجرای برنامه به یک بردار تبدیل می‌شود. هر خانه از این بردار متناظر با یک جمله از برنامه است. حال برای مشخص کردن مقادیر متناظر هر جمله در این بردار بررسی می‌شود که آیا این جمله در برش پویای پسرو وجود دارد یا خیر.

فرض شود که برنامه P دارای یک جمله خروجی به نام O_P شامل k نمونه اجرایی باشد، به‌طوری که $O_P = \{o_{p_1}, o_{p_2}, \dots, o_{p_k}\}$، به ازای برخی موردهای آزمون، نتایج نادرست تولید کند. در اینجا برای سادگی، یک نمونه اجرایی از O_P را در نظر می‌گیریم که دارای یک مقدار نادرست در حداقل یک اجرای P باشد. برای این نمونه اجرایی، o_{p_j}، برش پویای پسرو $BDS_P^m\left(o_{p_j}\right)$ در اجرای ناموفق tc_m^{fail} و در اجرای موفق tc_n^{pass} محاسبه می‌شود که آن را با $BDS_P^n\left(o_{p_j}\right)$ نشان می‌دهیم. برای برنامه‌های دارای بیش از یک خروجی نادرست، کافی است اجتماع برش‌های پسروی همه جملات خروجی نادرست محاسبه شود. در این مرحله اگر موردهای آزمون موفقی باشد که BDS آن‌ها مشابه برخی موردهای آزمون ناموفق باشد، از تحلیل‌های بعدی حذف می‌شوند.

برای محاسبه برش‌های پویای پسرو برنامه‌های جاوا از ابزار JavaSlicer [۱۹] استفاده شده است. با در اختیار داشتن مجموعه‌ای از جملات که از برش پویای پسروی مربوط به اجراهای موفق و ناموفق برنامه استخراج شده است یک فضای اقلیدسی n بعدی $Prog - Space(P)$ به نام $\{x_1, x_2, \dots, x_n\}$ تعریف می‌شود که در آن x_j نمایانگر یک جمله مشخص در برش پویای پسرو است و می‌تواند مطابق با تعریف (۱-۴)، سه مقدار $\{-1, 0, 1\}$ داشته باشد.

تعریف ۴-۱. هر اجرای برنامه توسط موردآزمون tc_k، بدون درنظرگرفتن وضعیت پایانی آن، می‌تواند به صورت بردار اجرایی $V_P^k = \{X_1, X_2, \dots, X_n\}$ در فضای $Prog - Space(P)$ نمایش داده شود به طوری که X_i مقدار بعد x_i است و می‌تواند یکی از سه مقدار زیر را داشته باشد:

$$X_i = \begin{cases} 1, & if\left(x_i \in BDS_P^k(op_j)\right) \\ 0, & if\left(x_i \in \{G_P^k(N,E) - BDS_P^k(op_j)\}\right) \\ -1, & if(x_i \in \{All.St_P - G_P^k(N,E)\}) \end{cases} \quad (۱)$$

پس از تولید بردارها، به منظور محاسبه برش پویای پسروی بسط یافته از الگوریتم K-Means [۳۵] به‌صورت زیر استفاده می‌شود. برای خوشه‌بندی m بردار اجرایی (بدون در نظر گرفتن وضعیت خروجی برنامه) به k خوشه که $(k \leq m)$ مطابق با تابع هدف بیان شده در رابطه (۲) عمل می‌کنیم:

$$argmin \sum_{i=1}^{k} \sum_{V_K^P \in cl_i} V_K^P - C_i^2 \quad (۲)$$

که در آن cl_i خوشه iام و C_i مرکز آن است.

بعد از خوشه‌بندی، در هر خوشه جملاتی که در حداقل یک بردار اجرایی دارای ارزش صفر هستند، شناسایی می‌شوند. چنین جمله‌ای با نماد x نشان داده شده است. برای آنکه مشخص شود جملات مستعد خطا هستند یا خیر از احتمال شرطی x به ازای اجراهای ناموفق موجود در آن خوشه دارای ارزش ۱ باشد، استفاده می‌شود. یعنی مقدار $P(x = 1|\, fail)$ محاسبه می‌شود. به همین ترتیب احتمال ۱ بودن جمله به ازای اجراهای موفق، $P(x = 1|\, pass)$ را محاسبه می‌کند. اگر مقدار احتمال محاسبه شده کمتر از ۰/۵ باشد، جمله مورد نظر مستعد خطا نبوده و ارزش صفر جمله به ۱- تبدیل می‌شود. در غیراینصورت، اگر مقدار احتمال بیشتر از ۰/۵ باشد، جمله مورد نظر می‌تواند مظنون به خطا باشد و در نتیجه ارزش صفر آن به ۱+ تبدیل می‌شود. چنان‌چه احتمال محاسبه شده دقیقاً برابر

۰/۵ باشد، در مورد بردارهای اجرایی ناموفق، مقدار صفر به ۱+ و در مورد بردارهای اجرایی موفق به ۱- تبدیل می‌گردد. چنانچه ارزش یک جمله در همه بردارهای اجرایی موفق یا ناموفق برابر صفر باشد، محافظه‌کارانه عمل شده و در مورد بردارهای اجرایی موفق، ارزش صفر به ۱- و در مورد بردارهای اجرایی ناموفق به ۱+ تبدیل می‌گردد. پس از محاسبه برش پویای بسط یافته، نوبت به محاسبه عوامل احتمالی خطا، یعنی مجموعه FCC می‌رسد. برای این منظور، فرض شود که $TR = \{tr_1, tr_2, \ldots, tr_n\}$، مجموعه اطلاعات اجرایی مربوط به موردهای آزمون برای برنامه P و مجموعه برش‌های بسط‌یافته آنها، $ES = \{es_1, es_2, \ldots, es_n\}$ باشد. مجموعه‌ی TR را به دو زیر مجموعه TR_p و TR_f و مجموعه ES را به دو زیر مجموعه ES_P و ES_F تقسیم می‌کنیم؛ که متناظر با موردهای آزمون موفق و ناموفق هستند. سپس، اطلاعات اجرایی، که به بردارهای اجرایی تبدیل می‌شوند، را خوشه‌بندی کرده و اشتراک ES_Fها را در هر خوشه بدست می‌آوریم. با فرض اینکه R_i بیانگر جملاتی باشد که در همه برش‌های بسط‌یافته اجراهای ناموفق موجود در خوشه i حضور دارند و r_i عضوی از اشتراک ES_Fها برای خوشه iام باشد، خواهیم داشت:

$$FCC = \cup_{i=1}^{k} R_i, \quad R_i = \{r_i | \forall r_i \in \cap_{j=1}^{m} tr_j \wedge tr_j \in TR_f\} \quad (۳)$$

برای محاسبه احتمال تصادفاً موفّق هر یک از موارد آزمون موفق، $CC(p)$، از رابطه (۴) استفاده می‌شود که دو ابتکار را ترکیب می‌کند: در ابتکار اول، میانگین امتیاز مظنون به خطایی جملاتی که توسط مورد آزمون موفق اجرا می‌شوند؛ توسط روش Ochiai [۳۶] که یک روش سبک‌وزن و کارآمد مکان‌یابی خطا است، محاسبه می‌شود. انتخاب این ابتکار بر این ایده استوار است که هر چقدر یک مورد آزمون موفق جملات مظنون به خطای بیشتری را اجرا کرده باشد، احتمال اجرا کردن جمله خطا هم در آن بالا خواهد رفت و در نتیجه از احتمال بیشتری برای تصادفاً موفق بودن برخوردار است.

ابتکار دوم، میزان پوشش کد حاصل از اجرای یک مورد آزمون موفق، است. اجرای برنامه با یک مورد آزمون موفق، هر چقدر منجربه پوشش میزان بیشتری از جملات برنامه شود، احتمال اینکه مورد آزمون تصادفاً موفق باشد، بالا می‌رود. رابطه (۴) به صورت میانگین دو ابتکار تشریح شده تعریف و محاسبه می‌گردد:

$$CC(p) = \frac{1}{2}\left(\frac{\sum_{p \in S_{p-FCC-Covered}} Ochiai(p)}{|S_{p-FCC-Covered}|} + \frac{|S_{p-FCC-Covered}|}{|S_{FCC}|}\right) \quad (۴)$$

در رابطه فوق، $S_{p-FCC-Covered}$ دستورات اجرا شده داخل مجموعه FCC برای مورد آزمون p و S_{FCC} کل دستورات مجموعه FCC است.

۳-۳- محاسبه محدوده مظنون به خطا

بعد از شناسایی موارد آزمون تصادفاً موفق حال نوبت به محاسبه محدوده مظنون به خطا رسیده است. این محدوده در واقع یک بلاک اولیه برنامه است. همان‌طور که در گام اول بیان شد با تغییر یک گزاره یک محدودیت مسیر جدید تولید شده و برای این محدودیت مسیر داده آزمون متناظر نیز تولید می‌شود. لازم به ذکر است که در هر مرحله تنها یک گزاره تغییر می‌کند.

با توجه به محدودیت‌های مسیر به دست آمده از گام اول و نتایج آزمون حاصل از گام دوم که هر کدام متناظر با یک محدودیت مسیر هستند می‌توان ناحیه مظنون به خطا را بدین صورت محاسبه کرد. ابتدا مسیرخطادار را انتخاب کرده سپس نتیجه این مسیر را با نتیجه محدودیتِ مسیر بعدی به دست آمده از خود، مقایسه می‌کنیم. اگر نتیجه آزمون از ناموفق به موفق تغییر پیدا کرده بود آنگاه گزاره‌ای که تغییر کرده است به عنوان شاخه خطادار معرفی می‌گردد. این گزاره را *گزاره بحرانی* می‌نامیم. در غیر اینصورت نتیجه محدودیت مسیر دوم با نتیجه محدودیت مسیر سوم مورد مقایسه قرار می‌گیرد. این کار تا زمانی که اولین شاخه خطادار پیدا شود، ادامه می‌یابد.

۳-۴- برآورد اثر علّی جملات شاخه خطادار

بعد از محاسبه شاخه خطادار به منظور محاسبه اثر علّی برای هر یک از جملات درون این شاخه از راهکاری مشابه روش فیضی و پارسا [۳۷] استفاده می‌کنیم، با این تفاوت که تنها جملاتی از بلاک اولیه تحت شاخه خطادار در نظر گرفته می‌شوند، که در برش پویای پسرو [۱۸] اجرای خطادار حضور دارند. بدین ترتیب، امتیاز مظنون به‌خطایی تنها برای این جملات محاسبه خواهد شد که محدوده مکان‌یابی شده را کوچک‌تر و دقیق‌تر می‌کند.

در این روش، برای محاسبه امتیاز مظنون به خطایی جملات، ابتدا موردهای آزمون براساس نمرات گرایش همتاسازی می‌شوند [۲۳]. برای همتاسازی از رده‌بند رگرسیون لجستیک [۳۸] با رابطه (۵) استفاده می‌شود. در این رابطه، عناصر بردار X_s که شامل فاکتورهای اختلاطی است، به عنوان متغیرهای پیشگو بوده، $\Pr[T_s = 1|X_s]$، احتمال خطادار بودن جمله s و $\Pr[T_s = 0|X_s]$ احتمال فاقط خطا بودن جمله s است. در فرایند برازش ضرایب β به نحوی تعیین می‌گردند که میزان خطای دو طرف تساوی رابطه (۵) کمینه گردد.

$$log\frac{\Pr[T_s=1|X_s]}{\Pr[T_s=0|X_s]} = X'_s\beta \quad (۵)$$

سپس، با توجه به روابط (۶) و (۷) اقدام به محاسبه برآورد اثر علّی می‌کنیم و جملات را براساس امتیازشان به صورت نزولی مرتب می‌کنیم. در رابطه (۶)، برای هر مورد آزمون i، یکی از دو نتیجه بالقوه مشاهده می‌گردد، به‌طوری‌که، اگر $T_i = 1$ آنگاه نتیجه

مشاهده شده Y_{i1} و چنانچه $T_i = 0$ آنگاه نتیجه مشاهده شده Y_{i0} خواهد بود. نتیجه‌ی بالقوه‌ی دیگری که ناموجود است را می‌توان بر اساس میانگین نتیجه‌ی همتاهای آن یعنی M مورد آزمون دیگر، بدست آورد. بعد از مشاهده و محاسبه نتایج بالقوه به ازای هر اجرا، می‌توان با استفاده از برآوردگر ارائه شده در رابطه (۵) به برآورد اثر علّی N مورد آزمون پرداخت.

در پایان، لیست مرتب شده جملات، برحسب امتیاز مظنون به خطایی آنها، به کاربر گزارش می‌شود.

$$\hat{Y}_{i0} = \begin{cases} \frac{1}{M}\sum_{j\in J_M(i)} Y_j & if\ T_i = 1 \\ Y_i & if\ T_i = 0 \end{cases}$$

$$\hat{Y}_{i1} = \begin{cases} Y_i & if\ T_i = 1 \\ \frac{1}{M}\sum_{j\in J_M(i)} Y_j & if\ T_i = 0 \end{cases} \quad (۶)$$

$$\hat{\tau} = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_{i1} - \hat{Y}_{i0}) \quad (۷)$$

۴- ارزیابی روش پیشنهادی

در این بخش، عملکرد روش پیشنهادی به صورت تجربی مورد ارزیابی قرار می‌گیرد. در این راستا، روش پیشنهادی با تعدادی از روش‌های برجسته مکان‌یابی خطای مبتنی بر طیف مورد مقایسه قرار گرفته است. به منظور بررسی عملکرد روش پیشنهادی، دو پرسش پژوهش زیر مطرح شده‌اند تا بتوان روش پیشنهادی را از جنبه‌های مختلف ارزیابی نمود:

۱- روش پیشنهادی در تعداد جملات بررسی شده برای مکان‌یابی خطا، چه میزان بهبود داشته است؟

۲- با تولید داده آزمون جدید برای کشف شاخه خطادار و محدود کردن تعداد جملات مورد بررسی به این شاخه آیا دقت مکان‌یابی خطا افزایش یافته است؟

تمامی آزمایش‌ها بر روی ماشینی با سیستم عامل ویندوز ۱۰، پردازنده مرکزی Intel®Core™i5 و 8GB حافظه اصلی اجرا شده است، در نتیجه روش پیشنهادی قابلیت اجرا بر روی یک سیستم متوسط را دارد.

۴-۱- برنامه‌های آزمون

برای ارزیابی از چهار پروژه متن باز JFreeChart، CommonsLang، CommonsMath و Joda-Time در مجموعه Defects4J [۳۹] که در مجموع شامل۲۲۴ خطای واقعی است، استفاده شده است. جدول (۱) برنامه‌های آزمون را نشان می‌دهد.

جدول(۱): برنامه‌های مورد استفاده در ارزیابی روش پیشنهادی

Projects	Faults	Line of code
Appache Commons Math	106	85 K
Appache Commons Lang	65	22 K
Joda-Time	27	28 K
JfreeChart	26	96 K
Total	224	231 K

۴-۲- معیارهای ارزیابی

برای ارزیابی و مقایسه روش پیشنهادی از دو معیار EXAM و متوسط تعداد جملات بررسی شده استفاده می‌گردد. این معیارها، مهمترین معیارهای ارزیابی در پژوهش‌های مکان‌یابی خطا هستند [۱۰], [۴۰] و به‌صورت زیر تعریف می‌شوند.

- **معیار EXAM:** این معیار بیانگر درصدی از جملات برنامه است که برای یافتن اولین جمله خطا باید بررسی شوند. بر اساس [۴۱] هدف اصلی در یک روش مکان‌یابی خطا، تعیین یک نقطه‌ی شروع برای برنامه‌نویسان است تا بتوانند از آن مکان شروع کرده و علّت خروجی ناموفق را شناسایی کرده و اصلاحات مورد نیاز را انجام دهند.
- **معیار متوسط تعداد جملات بررسی شده:** این معیار بیانگر متوسط تعداد جملات بررسی شده برای مکان‌یابی خطا در برنامه است.

لازم به ذکر است که در روش پیشنهادی و سایر روش‌های مکان‌یابی خطا، ممکن است چند دستور از یک برنامه امتیاز مظنون به خطایی یکسانی دریافت کرده باشند (که از این جملات، برخی عامل خطا نیستند). در این وضعیت، با توجه به اینکه این جملات با چه ترتیبی مورد بررسی قرار گیرند، میزان کاوش دستی کد مورد نیاز می‌تواند متفاوت باشد. برای رفع این مسئله، ما نتایج را در دو سطح مختلف، بهترین و بدترین حالت، ارائه می‌دهیم. در آزمایش‌ها، فرض کرده‌ایم که از جملاتی که امتیاز یکسان دریافت کرده‌اند، در بهترین حالت، جمله خطا به عنوان اولین جمله و در بدترین حالت، جمله خطا به عنوان آخرین جمله مورد بررسی قرار گرفته باشد.

۴-۳- نتایج

برای بررسی کارایی روش ارائه شده آن را از منظر معیارهای ارزیابی بیان شده در بخش ۴-۲، بررسی کرده‌ایم. جهت سنجش میزان بهبود حاصل از روش ارائه شده آن را با روش‌های مکان‌یابی خطا مبتنی بر طیف مقایسه می‌کنیم. ابتدا متوسط تعداد جملات مورد بررسی را برای هر کدام از این روش‌ها محاسبه و ثبت می‌کنیم. سپس برای روش پیشنهادی ابتدا شاخه خطادار را تعیین و متوسط

تعداد جملات مورد بررسی را به دست می‌آوریم.

جدول‌های (۲) و (۳)، به ترتیب متوسط تعداد جملات مورد بررسی در برنامه‌های آزمون، برای هر یک از روش‌های مکان‌یابی خطا را در بهترین و بدترین حالت، نشان می‌دهند. همان‌طور که مشخص است متوسط تعداد جملات مورد بررسی برای روش پیشنهادی با عنوان TD-CAFL، در مقایسه با سایر روش‌ها کمتر شده است. بهترین مقادیر در هر ستون پر رنگ شده‌اند. در روش پیشنهادی به دلیل محاسبه ناحیه مظنون به خطا، کاهش محدوده جملات مورد بررسی به بلاک اولیه و در نظر گرفتن برش پویای پس‌رو دستورات برنامه، شاهد کاهش متوسط تعداد جملات مورد بررسی نسبت به سایر روش‌های مکان‌یابی خطا هستیم. به عنوان مثال، در جدول (۲) متوسط تعداد جملات مورد بررسی توسط روش Ochiai در مورد نسخه‌های خطادار برنامه C-Math برابر با ۳۰۴۶/۱۹ است. این در حالی است که با محاسبه ناحیه مظنون به خطا توسط روش پیشنهادی، متوسط تعداد جملات مورد بررسی برای این برنامه ۱۷۴۶/۰۲ خواهد بود.

همچنین، در جدول (۳) متوسط تعداد جملات مورد بررسی در بدترین حالت برای روش پیشنهادی در مورد نسخه‌های خطادار برنامه C-Math برابر با ۳۱۵۱/۸۱ جمله است. در مجموع روش پیشنهادی برای چهار برنامه انتخابی از Defects4J [۳۹] نسبت به سایر روش‌های مکان‌یابی خطا بهتر عمل می‌کند و متوسط تعداد جملات مورد بررسی آن کمتر است. متوسط جملات مورد بررسی جهت کشف خطا، در بدترین حالت به میزان ٪۱۶/۷۸ کاهش داشته است.

جدول(۲): متوسط تعداد جملات مورد بررسی در برنامه‌های آزمون (بهترین حالت)

Technique	J-Chart	J-Time	C-Lang	C-Mat
TD-CAFL	**3719.16**	**162.47**	**1171.45**	**1746.02**
D-Star	4821.72	239.52	1408.65	3021.25
H3b	4872.15	235.42	1421.85	2956.11
H3c	4872.15	229.67	1411.56	2916.69
Ochiai	4770.90	241.49	1449.78	3046.19
O	5359.24	342.51	1479.94	4148.72
O^P	5448.62	362.14	1499.48	4182.42
Baah, 2010	4764.72	254.38	1412.26	2958.72
Baah, 2011	4625.38	224.50	1389.92	2914.65

جدول(۳): متوسط تعداد جملات مورد بررسی در برنامه‌های آزمون (بدترین حالت)

Technique	J-Chart	J-Time	C-Lang	C-Math
TD-CAFL	**4659.60**	**210.04**	**1340.41**	**3151.81**
D-Star	5658.26	294.52	1682.64	3945.27
H3b	5708.82	299.26	1698.11	3879.36
H3c	5108.82	281.75	1684.56	3824.12
Ochiai	5682.46	351.49	1699.78	4026.19
O	5848.75	392.69	1714.85	4841.56
O^P	5969.26	446.74	1736.62	4926.48
Baah, 2010	5548.83	344.28	1605.42	3916.92
Baah, 2011	5359.27	329.65	1582.39	3892.64

برای بررسی دقت روش پیشنهادی از معیار EXAM استفاده می‌کنیم. مقایسه دقت روش پیشنهادی با سایر روش‌های مکان‌یابی خطا، برای هریک از پروژه‌های Defects4J [۳۹]، به صورت جداگانه با نمودار مقادیر EXAM را در شکل‌های (۲) تا ۵، نشان داده شده است. روش پیشنهادی (TD-CAFL) با چهار روش مکان‌یابی خطای Ochiai [۳۶]، O، D-Star [۴۱] و FPA-FL [۲] مورد مقایسه قرار گرفته است و درصد خطاهایی که در هر بخش از کد پیدا شده، محاسبه شده است.

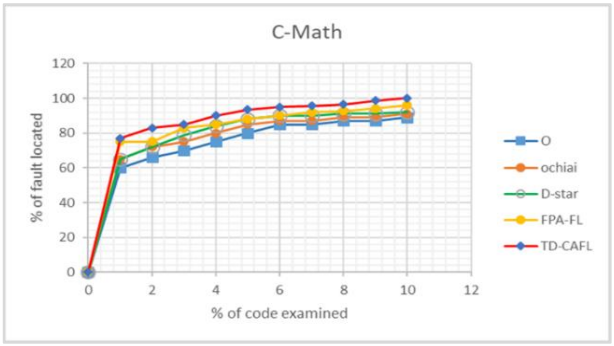
در شکل (۲) نتایج ارزیابی روش پیشنهادی در مورد نسخه‌های خطادار برنامه J-Chart در مقایسه با سایر روش‌های مکان‌یابی خطا با توجه به معیار EXAM ارائه شده است. محور افقی نمایانگر درصد کاوش دستی مورد نیاز کد و محور عمودی نمایان‌گر تعداد نسخه‌های خطاداری است که با توجه به مقدار مشخصی از کاوش دستی مورد نیاز قابل مکان‌یابی هستند. برای مثال، زمانی که EXAM برابر یا کمتر از یک درصد است؛ یعنی یک درصد از کد برنامه بررسی می‌شود، روش D-star توانسته ٪۴۴، روش Ochiai، ٪۵۵ و روش O، ٪۴۱، و روش FPA-FL، ٪۵۷/۰۲ از کل خطاهای مربوط به J-Chart را گزارش کند. این در حالی است که روش پیشنهادی ٪۷۵ از کل خطاها را گزارش می‌کند. مشاهده می‌شود که روش پیشنهادی، با بررسی ٪۸ از کد، حدود ٪۹۸/۵۶ از کل خطاهای برنامه J-Chart را گزارش می‌کند. مقدار بیشتر خطاهای مکان‌یابی شده در نمودار برای هر روش، بیان‌گر دقت بیشتر آن روش است. بنابراین، روش پیشنهادی عملکرد بهتری را در مقایسه با سایر روش‌ها، در مورد خطاهای واقعی موجود در برنامه J-Chart نشان می‌دهد.

شکل (۳) نتایج ارزیابی روش پیشنهادی در مورد نسخه‌های خطادار برنامه J-Time را در مقایسه با سایر روش‌های مکان‌یابی خطا با توجه به معیار EXAM نشان می‌دهد. زمانی که EXAM برابر یا کمتر از سه درصد است؛ یعنی سه درصد از کد برنامه بررسی می‌شود، روش‌های D-star و Ochiai و O توانسته‌اند ٪۸۴ از کل خطاهای مربوط به J-Time را گزارش کنند. این در حالی است که روش ارائه شده ٪۹۵ از کل خطاها را گزارش می‌کند. روش ارائه شده با بررسی ٪۵ از کد، ٪۱۰۰ خطاهای J-Time را گزارش می‌کند و عملکرد بهتری نسبت به سایر روش‌ها دارد.

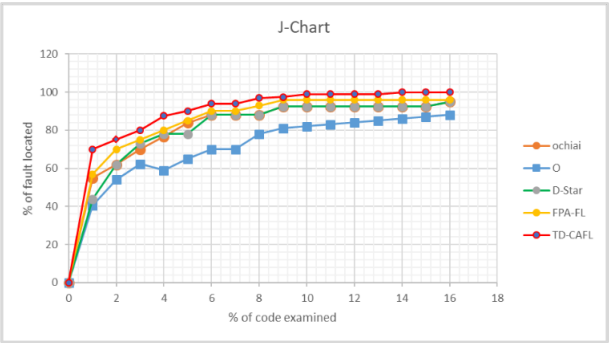
نتایج ارزیابی روش پیشنهادی از منظر دقت در گزارش خطاها در مورد نسخه‌های خطادار برنامه C-Lang با توجه به معیار EXAM در شکل (۴) نشان داده شده است. مطابق با شکل (۴) زمانی که معیار EXAM برابر یا کمتر از شش درصد است روش ارائه شده می‌تواند ٪۹۲ از کل خطاهای مربوط به C-Lang را گزارش کند. این

درحالی است که FPA-FL ۸۷/۰۴٪، Ochiai ۸۲٪، O ۸۰٪ از کل خطاها را گزارش می‌کنند.

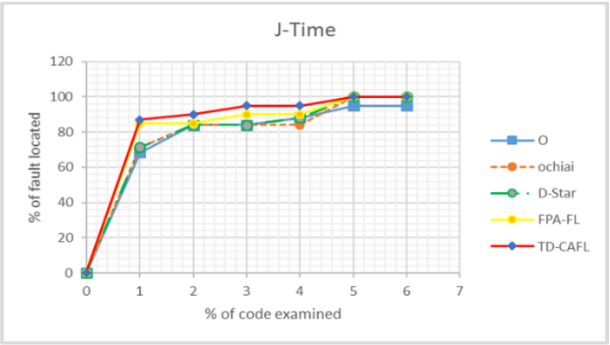
نتایج ارزیابی روش ارائه شده از منظر دقت در گزارش خطاها در مورد نسخه‌های خطادار برنامه C-Math با توجه به معیار EXAM در شکل (۵) نشان داده شده است. مطابق با شکل (۵) روش ارائه شده قادر است با ۱۰٪ کاوش دستی، ۱۰۰٪ خطاهای برنامه C-Math را مکان‌یابی کند. روش FPA-FL تنها ۹۵٪ از این خطاها را گزارش می‌کند.



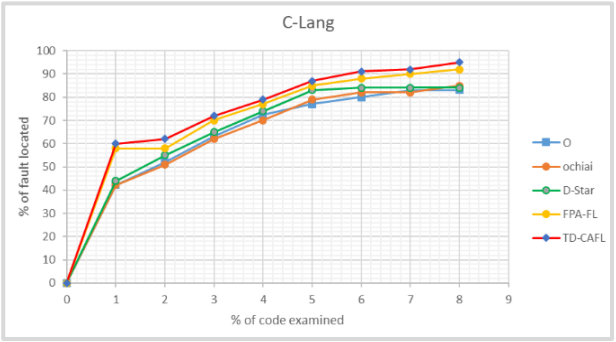
شکل(۵): مقایسه روش‌ها بر مبنای معیار EXAM برای برنامهC-Math



شکل(۲): مقایسه روش‌ها بر مبنای معیار EXAM برای برنامه J-Chart



شکل(۳): مقایسه روش‌ها بر مبنای معیار EXAM برای برنامهJ-Time



شکل(۴): مقایسه روش‌ها بر مبنای معیار EXAM برای برنامهC-Lang

با توجه به نمودارهای حاصله برای هرکدام از پروژه‌های Defects4J [۳۹] و مقایسه روش پیشنهادی با سایر روش‌های مکان‌یابی خطا، مشاهده می‌شود که روش ارائه شده در همه پروژه‌ها، دقت بالاتری نسبت به سایر روش‌ها کسب کرده است و نه تنها سبب مقیاس‌پذیر شدن روش‌های تحلیل علّی-آماری شده، بلکه باعث افزایش دقت در یافتن محل خطا نیز می‌گردد. در مجموع، نتایج نشان دهنده کشف ۷۵٪ از خطاها با بررسی حداکثر ۱ درصد از کد این برنامه‌ها است که در مقایسه با کارهای موجود ۱۷/۹۸٪ بهبود دارد.

براساس پژوهش‌های اخیر، روش‌های مکان‌یابی خطا مبتنی بر طیف کمترین مقدار EXAM را دریافت کرده‌اند [۴۰]. در جدول (۴) مشخص شده است که روش پیشنهادی نسبت به روش‌های مبتنی بر طیف EXAM پایین‌تری دارد. بنابراین، نسبت به روش‌های مکان‌یابی خطا در خانواده‌های دیگر، مانند روش‌های مبتنی بر جهش و مبتنی بر یادگیری ماشینی، نیز عملکرد بهتری خواهد داشت.

روش TD-CAFL با کاهش متوسط تعداد جملات مورد بررسی و در کنار آن، افزایش دقت مکان‌یابی خطا سبب مقیاس‌پذیر شدن و کاهش سربار محاسباتی روش‌های تحلیل علّی-آماری شده است. زمانی که یک روش برای کاهش تعداد جملات مورد بررسی ارائه می‌شود، اولین مسئله‌ای که تحت تأثیر قرار می‌گیرد، دقت مکان‌یابی خطا است. در بسیاری از روش‌ها تنها با هدف بهبود دقت مکان‌یابی خطا میزان هزینه اجرا تحمیل شده به سیستم را نادیده می‌گیرند و یا تنها با هدف کاهش میزان هزینه اجرا به دقتی کمتر یا مشابه بسنده می‌نمایند. با این حال نه تنها روش ارائه شده سبب کاهش دقت مکان‌یابی خطا نسبت به سایر روش‌های ارائه شده نشده است بلکه آن را بهبود داده است.

جدول(۴): میانگین EXAM روش‌های مکان‌یابی برای مجموعه Defects4j

Family	EXAM
TD-CAFL	**0.028**
SBFL	0.033
Predicate Switching	0.331

۵- نتیجه‌گیری

مکان‌یابی خطا قابل تفکیک به سه مرحله است. در مرحله اول می‌بایست مسیر خطادار، در مرحله دوم شاخه مظنون به خطا و در مرحله سوم محدوده مظنون به خطا در شاخه مشخص شده، تعیین گردند. روش‌های آماری یا مبتنی بر طیف، به دلیل عدم تفکیک سه مرحله، در عمل نسبت به روش پیشنهادی این مقاله، موفق نبوده‌اند و در ارزیابی‌ها، دقت کمتری را برای مکان‌یابی خطا نشان می‌دهند. علاوه بر این، روش‌های مبتنی بر طیف، وابسته به چگونگی و کیفیت داده‌های آزمون ممکن است نتایج متفاوتی را ایجاد کنند و در اصطلاح غیر پایدار هستند. روش پیشنهادی در واقع با یک جست‌وجوی دودویی در جهت معکوس مسیر اجرایی خطادار، سعی به تعیین شاخه اجرایی خطادار می‌نماید و با یافتن شاخه اجرایی خطادار با استفاده از شناسایی موارد آزمون تصادفاً موفق و اعمال روش علّی-آماری اقدام به رتبه‌بندی جملات در شاخه مظنون به خطا می‌نماید. در این روش، همگام با مکان‌یابی خطا، داده‌های آزمون با استفاده از اجرای نمادین پویا به مرور و به طور مؤثر ایجاد می‌گردد. در نتیجه، سرعت مکان‌یابی خطا افزایش می‌یابد. تعداد داده‌های آزمون تولید شده برای یافتن شاخه خطادار در روش پیشنهادی، حداکثر به تعداد شاخه‌های منشعب از مسیر اجرایی خطادار است.

برای برنامه‌های بزرگ کماکان آزمون همه شاخه‌ها ممکن است از نظر هزینه و منابع محاسباتی میسر نباشد، به‌همین جهت روش پیشنهادی بر روی برنامه‌هایی با تعداد شاخه‌های بالا، بودجه مکان‌یابی بالایی می‌طلبد. برای غلبه بر این چالش، می‌توان ابتدا خطاخیزی شاخه‌های برنامه را به‌صورت ایستا محاسبه نموده و سپس در شاخه‌های با خطاخیزی بیشتر عملیات جستجو و تولید داده آزمون را ادامه داد، که به‌عنوان کارهای آتی در این زمینه مد نظر است.

مراجع

[1] Kshirasagar Naik and Priyadarshi Tripathy, *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[2] F. Feyzi and S. Parsa, "FPA-FL: Incorporating static fault-proneness analysis into statistical fault localization," *Journal of Systems and Software*, vol. 136, pp. 39–58, Feb. 2018, doi: 10.1016/j.jss.2017.11.002.

[3] Y. Yang, F. Deng, Y. Yan, and F. Gao, "A fault localization method based on conditional probability," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2019, pp. 213–218. doi: 10.1109/QRS-C.2019.00050.

[4] T. Shu, T. Ye, Z. Ding, and J. Xia, "Fault localization based on statement frequency," *Inf Sci (N Y)*, vol. 360, pp. 43–56, Sep. 2016, doi: 10.1016/j.ins.2016.04.023.

[5] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016, doi: 10.1109/TSE.2016.2521368.

[6] A. Aghamohammadi, S.-H. Mirian-Hosseinabadi, and S. Jalali, "Statement frequency coverage: a code coverage criterion for assessing test suite effectiveness," *Inf Softw Technol*, vol. 129, p. 106426, Jan. 2021, doi: 10.1016/j.infsof.2020.106426.

[7] N. Neelofar, L. Naish, J. Lee, and K. Ramamohanarao, "Improving spectral-based fault localization using static analysis," *Softw Pract Exp*, vol. 47, no. 11, pp. 1633–1655, Nov. 2017, doi: 10.1002/spe.2490.

[8] A. Dutta, S. S. Srivastava, S. Godboley, and D. P. Mohapatra, "Combi-FL: Neural network and SBFL based fault localization using mutation analysis," *J Comput Lang*, vol. 66, p. 101064, Oct. 2021, doi: 10.1016/J.COLA.2021.101064.

[9] H. L. Ribeiro, P. A. R. de Araujo, M. L. Chaim, H. A. de Souza, and F. Kon, "Evaluating data-flow coverage in spectrum-based fault localization," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–11.

[10] S. Pearson *et al.*, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 609–620. doi: 10.1109/ICSE.2017.62.

[11] G. Candea and P. Godefroid, "Automated software test generation: some challenges, solutions, and recent advances," 2019, pp. 505–531. doi: 10.1007/978-3-319-91908-9_24.

[12] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge: Cambridge University Press, 2016. doi: DOI: 10.1017/9781316771273.

[13] E. Nikravan and S. Parsa, "Improving dynamic domain reduction test data generation method by Euler/Venn reasoning system," *Software Quality Journal*, vol. 28, no. 2, pp. 823–851, Jun. 2020, doi: 10.1007/s11219-019-09471-4.

[14] F. Belli, M. Beyazıt, A. T. Endo, A. Mathur, and A. Simao, "Fault domain-based testing in imperfect situations: a heuristic approach and case studies," *Software Quality Journal*, vol. 23, no. 3, pp. 423–452, Sep. 2015, doi: 10.1007/s11219-014-9242-6.

[15] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213–223, Jun. 2005, doi: 10.1145/1064978.1065036.

[16] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24.

[17] K. Luckow *et al.*, "JDart: A dynamic symbolic analysis framework," 2016, pp. 442–459. doi: 10.1007/978-3-662-49674-9_26.

[18] B. Korel and J. Laski, "Dynamic program slicing," *Inf Process Lett*, vol. 29, no. 3, pp. 155–163, Oct. 1988, doi: 10.1016/0020-0190(88)90054-3.

[19] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling Java programs for parallelism," in *2009 ICSE Workshop on Multicore Software Engineering*, May 2009, pp. 49–55. doi: 10.1109/IWMSE.2009.5071383.

[20] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*, 2010, p. 73. doi: 10.1145/1831708.1831717.

[21] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, 2011, p. 146. doi: 10.1145/2025113.2025136.

[22] H. Li, Y. Liu, Z. Zhang, and J. Liu, "Program structure aware fault localization," in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, Nov. 2014, pp. 40–48. doi: 10.1145/2666581.2666593.

[23] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*, May 2006, pp. 272–281. doi: 10.1145/1134285.1134324.

[24] N. Bayati Chaleshtari and S. Parsa, "SMBFL: slice-based cost reduction of mutation-based fault localization," *Empir Softw Eng*, vol. 25, no. 5, pp. 4282–4314, 2020, doi: 10.1007/s10664-020-09845-4.

[25] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*, 2008, p. 167. doi: 10.1145/1390630.1390652.

[26] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*, 2005, p. 273. doi: 10.1145/1101908.1101949.

[27] ben Liblit, *Cooperative Bug Isolation*, vol. 4440. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-71878-9.

[28] T. Chen, X. Zhang, S. Guo, H. Li, and Y. Wu, "State of the art: dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, Sep. 2013, doi: 10.1016/j.future.2012.02.006.

[29] F. Feyzi and S. Parsa, "A program slicing-based method for effective detection of coincidentally correct test cases," *Computing*, vol. 100, no. 9, pp. 927–969, Sep. 2018, doi: 10.1007/s00607-018-0591-z.

[30] A. Bandyopadhyay, "Mitigating the effect of coincidental correctness in spectrum based fault localization," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Apr. 2012, pp. 479–482. doi: 10.1109/ICST.2012.130.

[31] Y. MIAO, Z. CHEN, S. LI, Z. ZHAO, and Y. ZHOU, "A clustering-based strategy to identify coincidental correctness in fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 05, pp. 721–741, Jun. 2013, doi: 10.1142/S0218194013500186.

[32] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: coverage refinement with context patterns to improve fault localization," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 45–55. doi: 10.1109/ICSE.2009.5070507.

[33] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 37–61, Jan. 1985, doi: 10.1145/2363.2366.

[34] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, Oct. 2011, doi: 10.1016/j.jss.2011.05.016.

[35] E. Alpaydin, *Introduction to machine learning*, 4th edition. MIT Press, 2020. Accessed: Jul. 24, 2022. [Online]. Available: https://mitpress.mit.edu/books/introduction-machine-learning-fourth-edition

[36] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, Mar. 2014, doi: 10.1016/j.jss.2013.08.031.

[37] F. Feyzi and S. Parsa, "Inference: effective fault localization based on information-theoretic analysis and statistical causal inference," *CoRR*, vol. abs/1712.0, Dec. 2017, doi: 10.1007/s11704-017-6512-z.

[38] D. G. Kleinbaum and M. Klein, *Logistic regression*. New York, NY: Springer New York, 2010. doi: 10.1007/978-1-4419-1742-3.

[39] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, 2014, pp. 437–440. doi: 10.1145/2610384.2628055.

[40] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 332–347, Feb. 2021, doi: 10.1109/TSE.2019.2892102.

[41] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Trans Reliab*, vol. 63, no. 1, pp. 290–308, Mar. 2014, doi: 10.1109/TR.2013.2285319.

پاورقی‌ها:

[1] Test data

[2] Test data for causal-statistical analysis fault localization

[3] Dynamic slicing

[4] Bias

[5] Back-door

[6] Predicate switching

[7] Value replacement

[8] Interesting value mapping pair

[9] Fault candidate causes