نشریه علمی-ترویجی محاسبات نرم
شماره‌ی سوم/ بهار و تابستان ۱۳۹۲/ صفحه ۷۲-۸۵

تاریخ دریافت مقاله: تیر ۱۳۹۲

تاریخ پذیرش مقاله: بهمن ۱۳۹۲

مروری بر روش‌های تولید داده‌های آزمون در آزمون جهشی

حسن فرزانه کاجانی*[۱]، سینا بخشایشی تیل[۲]، رضا ابراهیمی آتانی[۳]، اسدالله شاه‌بهرامی[۴]

[۱]کارشناس ارشد، دانشکده فنی و مهندسی، گروه کامپیوتر، دانشگاه گیلان، گیلان، ایران

hfarzaneh@msc.guilan.ac.ir

[۲]دانشجوی کارشناسی ارشد، مؤسسه آموزش عالی مهرآستان، گیلان، ایران

sina.bakhshayeshi@gmail.com

[۳]استادیار، دانشکده فنی و مهندسی، گروه کامپیوتر، دانشگاه گیلان، گیلان، ایران

rebrahimi@guilan.ac.ir

[۴]دانشیار، دانشکده فنی و مهندسی، گروه کامپیوتر، دانشگاه گیلان، گیلان، ایران

shahbahrami@guilan.ac.ir

چکیده: رشد روزافزون توانمندی تجهیزات سخت‌افزاری موجب آن شده است که تولید محصولات نرم‌افزاری با جهش مواجه شود. این جهش، افزایش فشار کاری را برای مهندسان نرم‌افزار جهت تولید نرم‌افزارهای مطمئن در پی داشته است؛ ازاین‌رو، فرآیندی مستقل در چرخه‌ی تولید نرم‌افزار با عنوان آزمون نرم‌افزار ایجاد شده است. یکی از روش‌های فرآیند آزمون نرم‌افزار که تحقیقات بسیاری بر روی آن انجام گرفته است، آزمون جهشی می‌باشد. بیشتر تحقیقات انجام‌شده در آزمون جهشی، شامل کاهش هزینه‌های تولید نسخه‌های خطادار و کاربردی‌کردن آن در محیط‌های مختلف (مانند امنیت، وب و غیره) بوده است. آنچه به‌عنوان یک خلاء محسوب می‌شود، کم‌توجهی به اهمیت تولید داده‌های ورودی بهینه در آزمون جهشی است. این مقاله قصد دارد یک نقشه‌ی راه برای محققین علاقمند از طریق بررسی جامع و مقایسه‌ی تحقیقات انجام‌شده در زمینه‌ی تولید داده‌های آزمون فراهم نماید تا اهمیت تولید داده‌های آزمون را نمایان کند.

واژه‌های کلیدی: آزمون جهشی، روش‌های تولید داده‌ی آزمون، معیارهای استاندارد ارزیابی.

۱. مقدمه

با توجه به رشد فناوری اطلاعات و تجهیزات سخت‌افزاری در دهه‌ی اخیر، تولید محصولات نرم‌افزاری با رشد سریعی مواجه شد. این رشد روزافزون، مهندسان نرم‌افزار را با چالشی جدی به نام «سنجش کیفیت تولیدات نرم‌افزاری» مواجه کرد. بر این اساس، روش‌های مختلفی جهت آزمون و سنجش نرم‌افزار ارائه شد و درنهایت باعث شد آزمون نرم‌افزار به‌عنوان فرآیندی مستقل در چرخه‌ی تولید نرم‌افزار قرار گیرد.

آزمون جهشی یکی از روش‌هایی بود که تحقیقات بسیار زیادی بر روی آن صورت گرفت. این روش اولین بار توسط «Lipton»[۱] در مقاله‌ی دانش‌آموزی در سال ۱۹۷۱ معرفی شد؛ اما اندکی بعد به‌صورت رسمی در سال ۱۹۷۸ توسط «DeMillo»[۲] معرفی شد. در این روش، ابتدا نسخه‌های خطادار از برنامه‌ی اصلی ایجاد شده و سپس داده‌های ورودی آزمون جهت تشخیص خطاهای تزریق‌شده در نسخه‌های خطادار، بر روی آن‌ها اعمال می‌شود. هدف اصلی در این روش، تولید داده‌هایی است که بتواند خطاهای بیشتری را تشخیص دهد؛ زیرا بالارفتن کیفیت هر محصولی رابطه‌ی مستقیمی با داده‌های ورودی آزمون دارد.

در سال‌های اخیر، تحقیقات فراوانی بر روی این روش در زمینه‌های مختلف صورت گرفته است و بر اساس آن بسیاری از دانشمندان بر این باور شدند که این روش بسیار قدرتمندتر از سایر روش‌ها است؛ به‌طوری‌که «Walsh»[۳] در تحقیقی عملی نشان داد که این روش بسیار قدرتمندتر از سایر روش‌ها است؛ همچنین «Frankel»[٤] و «Offutt»[٥] اثبات کردند که این روش نسبت به آزمونی که در تشخیص خطاها جریان داده است، بسیار موفق‌تر عمل می‌کند؛ علاوه‌براین، تحقیقات اخیر «Andrew»[٦] نشان داد که تحلیل جهشی بسیار مقرون به صرفه و یک روش پرکاربرد در زمینه‌ی ارزیابی داده‌های آزمون جهشی است.

همان‌طور که بیان شد، تحقیقات فراوانی در حوزه‌های مختلف آزمون جهشی انجام شده است؛ اما خلائی که احساس می‌شود، نبود تحقیق جامع در زمینه‌ی فعالیت‌های انجام‌شده در حوزه‌ی تولید داده‌های باکیفیت و تلاش‌های صورت‌گرفته (نظری و عملی) در زمینه‌ی تشخیص (کشتن) نسخه‌های خطادار است. از طرفی دیگر، تولید داده‌های باکیفیت در آزمون جهشی از سوی کارشناسان و محققین، هنوز به‌عنوان مشکلی باز تلقی می‌شود که این مسئله خود به‌تنهایی اهمیت انجام این تحقیق و نگارش این مقاله را بیان می‌کند؛ ازاین‌رو، این مقاله تلاش می‌کند برای اولین بار تحقیقی جامع از تحقیقات انجام‌شده در موارد فوق را ارائه کرده و هریک را مورد بررسی قرار دهد تا زوایای آن‌ها برای محققان علاقمند به فعالیت در این زمینه بیشتر مشخص شده و درنهایت، نقشه‌ی راه کلی برای آن‌ها ترسیم شود.

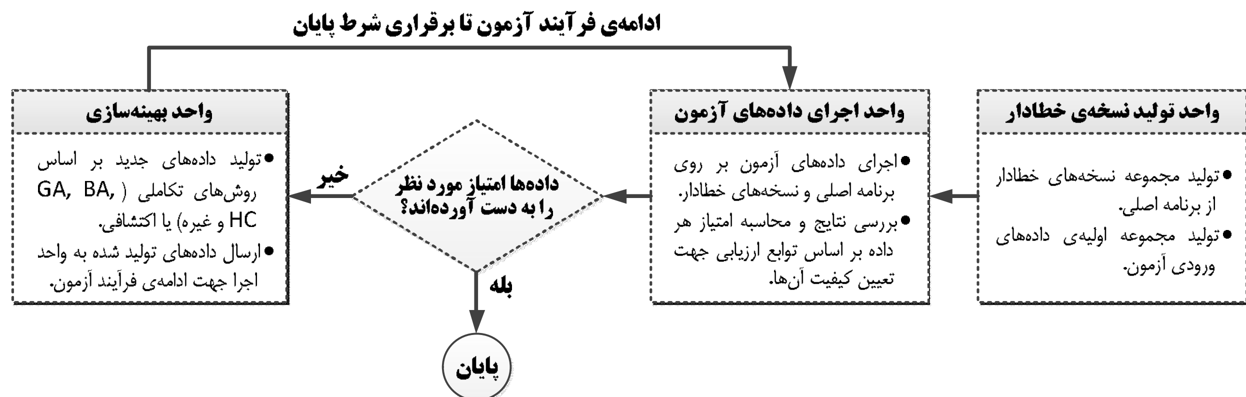
مقاله این‌گونه ادامه می‌یابد: در بخش ۲، ملزومات اولیه در آزمون جهشی (فرآیند کلی، عملگرهای جهشی و تحلیل جهشی) بیان شده است. بخش ۳ به تشریح مفاهیم کشتن ضعیف و قوی همراه با یک مثال می‌پردازد. بخش ٤ در ابتدا روش‌های مختلف تولید داده‌های آزمون در آزمون جهشی را همراه با مثال شرح داده و سپس در ادامه تحقیقات انجام‌شده در این زمینه را همراه با معیارهای استاندارد استفاده‌شده در آن‌ها مورد مقایسه قرار می‌دهد. در بخش‌های ٥ و ٦ نیز به‌ترتیب، نتیجه‌گیری و منابع مورد استفاده آورده شده است.

۲. پیش‌نیاز

ازآنجایی‌که این مقاله بر روی آزمون جهشی تمرکز دارد، اطلاعات پایه و زمینه‌ای مورد نیاز در زمینه‌ی آزمون جهشی در این بخش بیان شده است.

۱.۲. فرآیند کلی آزمون جهشی

فرآیند کلی این روش را می‌توان به سه بخش اصلی تقسیم کرد: ۱. واحد تولید نسخه‌ی خطادار؛ ۲. واحد اجرای داده‌های آزمون؛ ۳. واحد بهینه‌سازی. در واحد تولید نسخه‌های خطادار عملگرهای جهشی (در زیربخش ۲-۲ توضیح داده شده است) بر روی کدهای برنامه‌ی اصلی اعمال می‌شود و سپس نسخه‌هایی از برنامه‌ی اصلی که دارای خطاهای منفرد است، تولید می‌شود که اصطلاحاً به آن‌ها نسخه‌ی خطادار (جهش‌شده) می‌گویند. پس از تولید این نسخ خطادار، واحد اجرا موظف است داده‌های آزمون را بر روی برنامه‌ی اصلی و نسخه‌های خطادار اعمال و نتایج آن‌ها را با یکدیگر مقایسه کند.



شکل(۱): فرآیند کلی آزمون جهشی

در عملیات مقایسه، اگر نتایج حاصل از نسخه‌ی خطاداری با برنامه‌ی اصلی یکسان نباشد، اصطلاحاً گفته می‌شود که داده‌ی آزمون مورد نظر توانسته است خطای تزریق‌شده درون نسخه‌ی خطادار را کشف کند یا به عبارتی دیگر آن را بکشد.[1] بالعکس، اگر این نتایج یکسان باشد، اصطلاحاً گفته می‌شود نسخه‌ی خطادار مورد نظر معادل[2] با برنامه‌ی اصلی است [۷]. می‌توان نتیجه گرفت که داده‌ی مورد نظر از کیفیت مطلوب جهت کشف خطا برخوردار نبوده است؛ بنابراین، واحد بهینه‌سازی وظیفه دارد این داده‌ها را بهبود دهد و مجدداً مراحل قبل را تکرار کند.

در زیر، مراحل شکل (۱) به‌صورت خلاصه تشریح شده است:

• **واحد تولید نسخه‌های خطادار:** اجرای هر آزمون جهشی نیازمند تعدادی ملزومات اولیه است که عبارت‌اند از: مجموعه نسخه‌های خطادار (که از طریق اعمال عملگرهای جهشی تولید می‌شوند)، داده‌های اولیه‌ی ورودی آزمون، تعیین مقادیر ثابت‌ها و غیره که تمامی آن‌ها در این قسمت و قبل از شروع فرآیند آزمون تعیین می‌شوند.

• **واحد اجرای داده‌های آزمون:** در این مرحله، داده‌های اولیه بر روی برنامه‌ی اصلی و بعد از آن، به‌ترتیب بر روی تمامی نسخه‌های خطادار اجرا می‌شوند و نتایج حاصل از این اجراها با ساختاری مناسب ذخیره می‌شود. این نتایج جهت بررسی و تعیین میزان توانایی هریک از داده‌ها در تشخیص خطا استفاده خواهند شد. به‌منظور تعیین میزان توانایی هریک از داده‌ها در تشخیص خطا که نشان‌دهنده‌ی میزان کیفیت آن‌هاست، از توابع ارزیابی استفاده می‌شود. این توابع نتایج حاصل از اجرای داده‌ها بر روی برنامه‌ی اصلی و نسخه‌های خطادار را به‌عنوان پارامتر ورودی دریافت کرده و براساس فرمول‌های استاندارد خاصی (که فرمول‌های ارزیابی نام دارند) خروجی را محاسبه می‌نمایند. سپس مقادیر خروجی این توابع که نشان‌دهنده‌ی

1. Killed Mutant
2. Equivalent Mutant

میزان کیفیت هر داده است، به‌عنوان امتیاز به آن‌ها اختصاص داده می‌شود.

- **واحد بهینه‌سازی:** وظیفه‌ی این مرحله، بهینه‌سازی داده‌ها و تولید داده‌های جدید است. بهینه‌سازی تنها زمانی انجام خواهد شد که شرط تعیین‌شده برای اتمام مراحل آزمون ارضاء نشده باشد. بر این اساس، امتیاز داده‌ها پس از مرحله‌ی ارزیابی بررسی می‌شود و در صورتی که داده‌ها امتیاز مورد نظر را به دست نیاورده باشند، برای تغییر و بهینه‌شدن به این بخش ارسال می‌شوند. شرط تعیین‌شده برای اتمام آزمون متغیر است و در هر آزمون براساس اولویت‌های مدّنظر (معمولاً اولین اولویت) تعیین می‌شود. برای مثال، می‌توان تعداد مشخصی از اجراشدن یا رسیدن به امتیازی خاص را برای شرط پایانی درنظر گرفت.
- **پایان:** در این مرحله، آزمون به پایان رسیده و داده‌ها توانسته‌اند شرط پایانی را ارضاء کرده و کیفیت مورد نظر را به‌دست آورند. پس از پایان آزمون، روش‌های تکاملی از نظر سرعت رسیدن به مرحله‌ی پایان یا امتیاز ارزیابی داده‌هایی که تولید کرده‌اند، با یکدیگر مقایسه می‌شوند.

۲.۲. عملگرهای جهشی

همان‌طور که قبلاً بیان شد، تولید نسخه‌های خطادار از طریق عملگرهای جهشی صورت می‌پذیرد. برای مثال، عملگر AOR[1] یکی از عملگرهای جهشی پایه در آزمون جهشی است[۸،۷]. این عملگر وظیفه دارد عملگرهای محاسباتی (±، × ، ÷ و ...) مربوط به یک گزاره را با یکدیگر تعویض کند. عبارت زیر را به‌عنوان یک گزاره درنظر بگیرید:

$$A=B+C$$

در این صورت، پس از اعمال این عملگر می‌توان گزاره‌ای شبیه به گزاره‌ی زیر را تولید کرد:

$$A=B\times C$$

1. Arithmetic Operator replacement

۳.۲. تحلیل جهش

برای آنکه بتوان از کیفیت و قدرت توانایی تشخیص خطا توسط داده‌های آزمون آگاه شد، می‌توان از تحلیل جهشی استفاده نمود که به شرح زیر است [۹].

$$Mutation\ Score=\left(\frac{Killed}{All-Equivalent}\right)\times 100 \quad (1)$$

تابع فوق از سه پارامتر تشکیل شده است: پارامتر Killed تعداد نسخ خطاداری است که خطاهای تزریق‌شده در آن‌ها توسط داده‌ی آزمون مورد نظر کشف شده است. پارامتر Equivalent برابر است با تعداد نسخه‌های خطاداری که معادل با برنامه‌ی اصلی بوده‌اند و همچنین هیچ‌یک از داده‌های آزمون، توانایی کشف خطاهای موجود در آن‌ها را نداشته است. پارامتر All تعداد کل نسخه‌های خطادار تولید شده از برنامه‌ی اصلی را دربرمی‌گیرد.

۳. کشتن نسخه‌های خطادار

در این بخش، دو روش کشف خطای ضعیف[2] و قوی[3] که در حوزه‌ی کشتن نسخه‌های خطادار استفاده می‌شوند، همراه با مثال عددی شرح داده شده است. با توجه به شکل (۲)، برنامه‌ی یافتن بزرگ‌ترین مقدار از میان سه ورودی به‌عنوان برنامه‌ی اصلی درنظر گرفته شده است. از آنجایی‌که فرآیند آزمون جهشی نیازمند تولید نسخه‌های خطادار است، در این مثال چهار نسخه‌ی خطادار تولید شده است که جزئیات آن در شکل آمده است. قبل از ادامه‌ی توضیح مثال، آشنایی با دو مفهوم کشتن قوی و ضعیف ضروری است [۱۰].

2. Strongly Killing
3. Weakly Killing

	اجرا	مقایسه نتایج				تصمیم گیری	
	داده ورودی	نتایج ضعیف		نتایج قوی		کشتن ضعیف	کشتن قوی
		نسخه خطادار	برنامه اصلی	نسخه خطادار	برنامه اصلی		
نسخه‌ی خطادار ۱	a=10 , b=20 , c=30	30	30	20	30	×	✓
نسخه‌ی خطادار ۲	a=4 , b=7 , c=8	3	4	8	8	✓	×
نسخه‌ی خطادار ۳	a=4 , b=5 , c=6	false	true	5	6	✓	✓
نسخه‌ی خطادار ٤	b=2 , c=3 , a=1	2	2	3	3	×	×

ج) مراحل اجرایی نسخه‌های خطادار

```
1  public int Find_MAX(int a, int b,
2  int c)
3  {
4      int max=0;
5      if(a>b)
6          max=a;
7      else
8          max=b;
9      if(c>max)
10         max=c;
11     return max;
   }
```

الف) برنامه ماکزیمم

نام نسخه‌ی خطادار	دستور جهش شده	شماره دستور جایگزین شده
نسخه‌ی خطادار ۱	c=c;	9
نسخه‌ی خطادار ۲	max=a-1;	5
نسخه‌ی خطادار ۳	if(--c>max)	8
نسخه‌ی خطادار ٤	max=b-- ;	7

ب) جزئیات خطاهای به‌وجودآمده

شکل(۲): مثالی از کشتن ضعیف و قوی نسخه‌ی خطادار

- **کشتن ضعیف:** در کشتن ضعیف، نیاز به اجرای کامل نسخه‌ی خطادار نبوده و فقط نتیجه‌ی دستور جهش‌شده نسبت به همان دستور در برنامه‌ی اصلی مقایسه می‌شود.
- **کشتن قوی:** در کشتن قوی، نسخه‌ی خطادار به‌طور کامل اجرا گشته و نتیجه‌ی نهایی آن نسبت به برنامه‌ی اصلی مقایسه می‌شود.

اگر به شکل (۲) قسمت (ج) توجه داشته باشید، در ابتدا داده‌های ورودی در ستون داده‌ی ورودی قرار گرفته‌اند. این داده‌ها بر روی نسخه‌ی اصلی و نسخه‌ی خطادار اعمال شده و نتایج آن‌ها در دو حالت ضعیف و قوی مقایسه شده است. برای مثال، داده‌ی مورد نظر در نسخه‌ی خطادار ۲ توانسته است در حالت اجرایی ضعیف، خروجی متفاوتی را نسبت به برنامه‌ی اصلی تولید کند که درنهایت می‌توان نتیجه گرفت داده‌ی مورد نظر به‌صورت ضعیف نسخه‌ی خطادار ۲ را کشته است یا به عبارت دیگر، توانسته است خطای تزریق شده در آن را به‌صورت ضعیف تشخیص دهد؛ اما داده‌ی آزمون ورودی در نسخه‌ی خطادار ٤ نتوانسته است در هیچ‌یک از حالت‌های ضعیف و قوی باعث تولید خروجی متفاوت و کشته‌شدن آن شود؛ یعنی داده‌ی مورد نظر از کیفیت مطلوب جهت کشف خطاهای تزریق‌شده برخوردار نبوده است؛ درنتیجه، می‌توان آن را به‌عنوان نسخه‌ی خطادار معادل با برنامه‌ی اصلی درنظر گرفت.

با توجه به مثال فوق، هدفی که در این حوزه‌ی تحقیقی دنبال می‌شود، تولید داده‌هایی است که توانایی کشف خطاهای تزریق‌شده در نسخه‌های خطادار (به عبارت دیگر، توانایی کشتن نسخه‌های خطادار) را داشته باشد. آنچه به‌عنوان حالت ایده‌آل و قابل قبول در نظر گرفته می‌شود، توانایی داده‌های آزمون در کشتن نسخه‌های خطادار به دو حالت ضعیف و قوی است. برای آشنایی بیشتر، در بخش بعد مجموعه‌ی تحقیقات انجام‌شده در این حوزه‌ی تحقیقی ارائه شده است.

٤. کارهای انجام‌شده جهت تشخیص نسخ خطادار

در این بخش، کارها و مقالات انتشاریافته در زمینه‌ی تولید داده‌های آزمون جهت کشتن نسخه‌های خطادار معرفی شده است. از آنجایی‌که این مقالات از روش‌های مختلفی استفاده کرده‌اند، مناسب است قبل از بررسی و مقایسه‌ی آن‌ها، نگاهی اجمالی بر هرکدام از این روش‌ها داشته و هرکدام به‌طور مختصر توضیح داده شود.

١.٤. روش‌های مختلف تولید داده‌ی آزمون

به‌طور کلی، روش‌های مورد استفاده جهت تولید داده در آزمون جهشی به دو دسته تقسیم می‌شوند: ایستا[1] و پویا[2]. در حالت ایستا نیاز به اجرای کامل برنامه نبوده و آزمون‌گر با اعمال تغییراتی ثابت در کد مورد نظر، اقدام به تولید داده می‌کند؛ اما حالت پویا در تضاد با حالت ایستا می‌باشد؛ به عبارت دیگر، برنامه بدون هیچ‌گونه تغییرات ثابتی اجرا شده و در حین اجرا اقدام به تولید داده می‌شود.

١.١.٤. روش‌های ایستا

روش‌های ایستا شامل دو روش کاهش دامنه[3] و اجرای نمادینی[4] می‌باشد که هریک در زیر توضیح داده شده است.

- **کاهش دامنه:** این روش با استفاده از حذف متغیرهای مازاد یا قراردادن قیود مختلف، اقدام به کاهش دامنه‌ی تولید اعداد می‌نماید؛ برای مثال، در حالت حذف متغیرهای مازاد با نگاه‌کردن به کد شکل (۳) متوجه خواهید شد که متغیر d در تصمیم‌گیری دو دستور شرطی (دستور ٦ و ٨) تأثیر ندارد؛ بنابراین، می‌توان برای کاهش دامنه‌ی تولید داده‌ها متغیر غیر موثر d را حذف کرد [۱۱].

```
1  public   void   F1(int
2  a,b,c,d)
3  {
4        .
5        .
6        .
7        if(a>b)
8            a++;
9        if(b>c)
10           b++;
11       .
12       .
13       .
   }
```

شکل (۳): مثالی از کاهش دامنه

در حالت اعمال قیود نیز، تلاش بر این است که دامنه‌ی تولید اعداد با استفاده از قیود متفاوت[5] کاهش داده شود [۱۲]؛ برای نمونه، مثال زیر را در نظر بگیرید.

if(I+K)>=J	دستور اصلی (e)
if(3+K)>=J	دستور جهش شده (e′)
I≠3	قید لحاظ شده

همان‌طور که مشاهده می‌کنید، دستور جهش‌شده برای معادل‌نشدن با دستور برنامه‌ی اصلی ($e{\neq}e'$) نیاز به قید ($I{\neq}3$) دارد؛ به عبارتی دیگر، این روش از طریق حذف داده‌هایی که باعث تولید نسخه‌های خطادار معادل می‌شود، اقدام به کاهش دامنه‌ی تولید داده‌های ورودی آزمون می‌کند.

- **اجرای نمادینی:** در این روش، به‌جای آنکه اقدام به تولید داده‌های عددی شود، از داده‌های نمادین استفاده می‌شود [۱۳]. کد شکل (٤) را درنظر بگیرید.

```
1  public void SUM(int a,b,c){
2       int x=a+b;
3       int y=b+c;
4       int z=x+y-b;
5       return z;
6  }
```

شکل (٤): مثالی از اجرای نمادینی

فرض کنید مقادیر عددی پارامترها برابر ($a{=}1, b{=}3, c{=}5$) باشد. در این صورت، روند اجرای عددی کد شکل (٤) به‌صورت شکل (٥) است.

---

1. Static
2. Dynamic
3. Static Domain Reduction (SDR)
4. Static Symbolic Execution (SSE)
5. Constraint-Based Testing (CBT)

دستور	مقادیر متغیر ها					
	X	Y	Z	A	B	C
١	؟	؟	؟	١	٣	٥
٢	٤	–	–	–	–	–
٣	–	٨	–	–	–	–
٤	–	–	٩	–	–	–
٥	مقدار ٩ بازگردانده می‌شود					

شکل(٥): اجرای عددی

دستور	مقادیر متغیر ها					
	X	Y	Z	A	B	C
١	؟	؟	؟	a1	a2	a3
٢	a1+a2	–	–	–	–	–
٣	–	a2+a3	–	–	–	–
٤	–	–	a1+a2+a3	–	–	–
٥	مقدار a1+a2+a3 بازگردانده می‌شود					

شکل(٦): اجرای نمادین به صورت ایستا

در شکل(٥)، متغیرهای موجود هر دستور به‌صورت عددی محاسبه و اجرا شده است؛ اما روش نمادین آن کاملاً متفاوت و به‌صورت شکل (٦) می‌باشد. اگر توجه نمایید به‌جای تولید داده‌های ورودی به‌صورت عددی، از حروف نمادین جهت کاهش دامنه‌ی تولید اعداد استفاده شده است. نکته‌ای که می‌توان گفت آن است که در این روش می‌توان از روش کاهش دامنه نیز استفاده کرد.

٢.١.٤. روش‌های پویا

روش‌های پویا شامل سه روش کاهش دامنه[1]، اجرای نمادینی[2] و تولید داده‌ی مبتنی بر جستجو[3] می‌باشد که هریک در زیر توضیح داده شده است.

1. Dynamic Domain Reduction (DDR)

• **کاهش دامنه:** این روش یکی از روش‌های معروف در زمینه‌ی تولید داده‌های باکیفیت آزمون جهت کشتن نسخه‌های خطادار می‌باشد. به طور کلی، این روش ابتدا برای هریک از متغیرهای موجود در کد برنامه دامنه‌ای تعریف کرده و سپس گراف کنترل جریان[4] آن تشکیل می‌شود. پس از آن، مسیری دلخواه از گراف مورد نظر انتخاب می‌شود و برای آنکه عبور از مسیر انتخاب‌شده تضمین شده باشد، یعنی داده‌های ورودی بتوانند شرط‌های مسیر را ارضاء کنند، دامنه‌ی ورودی متغیرها را تغییر می‌دهد [١٤]. برای روشن‌ترشدن عملکرد این روش به مثال شکل (٧) توجه نمایید.

فرض کنید که دامنه‌ی اولیه‌ی تعریف‌شده برای متغیرهای A، B و C به‌صورت زیر است:

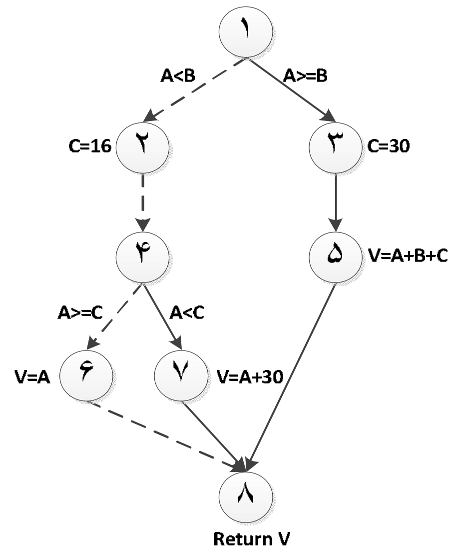
$$A: <0 \ldots 20>$$
$$B: <10 \ldots 40>$$
$$C: <0 \ldots 100>$$

قسمت (ب) در شکل (٧) گراف کنترل جریان و مسیر هدف انتخاب‌شده به‌صورت خطوط مقطع را نشان می‌دهد. برنامه در اولین مرحله‌ی اجرا عدد ١٤ را به‌عنوان نقطه‌ی تفکیک دامنه انتخاب می‌کند. بر این اساس، برای آنکه این دامنه توانایی تولید اعدادی جهت عبور از شرط یال ٢-١ را داشته باشد، دامنه‌ی متغیر A برابر $<0...14>$ و دامنه‌ی متغیر B برابر $<15...40>$ خواهد شد. در ادامه‌ی روند اجرای برنامه، با دستور انتساب خط ٧ در گره ٢ مقدار متغیر C به ١٦ تغییر می‌یابد و موجب می‌شود دامنه‌ی C به‌صورت ثابت، یعنی $<16...16>$ درنظر گرفته شود. این امر باعث ناتوانی داده‌ها در عبور از شرط یال ٦-٤ $(A>=C)$ می‌شود؛ زیرا حد بالای دامنه‌ی متغیر A برابر ١٤ است که از عدد ١٦ کوچک‌تر است. در این حالت، شرط یال ٦-٤ هرگز ارضاء نخواهد شد؛ بنابراین، برای عبور کامل از مسیر انتخاب‌شده باید به گره ١ بازگشت و نقطه‌ی تفکیک دیگری را برای تعیین دامنه‌ی مناسب انتخاب کرد.

2. Dynamic Symbolic Execution (DSE)
3. Search Based Test case Generation (SBTG)
4. Control Flow Graph (CFG)

```
1  public int Value(int A, int B, int C)
2  {
3      int V;
4      V=0;
5      if(A<B)
6      {
7          C=16;
8          if(A<C)
9              V=A+30;
10         else
11             V=A;
12     }
13     else
14     {
15         C=30;
16         V=A+B+C;
17     }
18     return V;
19 }
```

(الف): کد اصلی



(ب): گراف کنترل جریان

شکل(۷): مثالی از کاهش دامنه به طور پویا

اکنون فرض کنید عدد ۱۷ به‌عنوان نقطه‌ی تفکیک دامنه‌ی جدید انتخاب شود. بر این اساس، دامنه‌ی متغیر A برابر $<0...17>$ و دامنه‌ی متغیر B برابر $<18...40>$ می‌شود. این دامنه برای عبور از عبارت شرطی یال ۲-۱ مناسب است؛ چراکه تمامی اعداد انتخاب‌شده از این دامنه‌ها شرط مورد نظر $(A<B)$ را ارضاء می‌کند؛ اما جهت عبور از شرط یال ٦-٤ $(A>=C)$ باید دامنه‌ی جدیدی برای متغیر A محاسبه شود. از آنجاکه دامنه‌ی متغیر C پس از دستور انتساب در گره ۲ برابر $<16...16>$ خواهد شد، در اینجا نقطه‌ی تفکیک ١٦ به‌عنوان حد پایین دامنه‌ی A انتخاب می‌شود تا شرط مورد نظر ارضاء شود؛ بنابراین، دامنه‌ی نهایی متغیر A برابر با $<16...17>$ خواهد شد؛ درنتیجه، این دامنه توانایی تولید داده‌هایی را خواهد داشت که می‌توانند عبور از مسیر انتخاب‌شده را تضمین کنند. برای مثال، ورودی‌های $(A=17,\ B=25,\ C=15)$ از دامنه‌ی نهایی فوق توانایی عبور از مسیر انتخاب‌شده را دارند.

به‌طور کلی، می‌توان نتیجه گرفت که روش کاهش دامنه به‌طور پویا تلاش می‌کند تا از طریق تعیین پویای حد فاصل مناسب در دامنه‌ی متغیرها، داده‌های ورودی را به‌گونه‌ای تولید کند که عبور از مسیر انتخاب‌شده در گراف کنترل جریان تضمین شود.

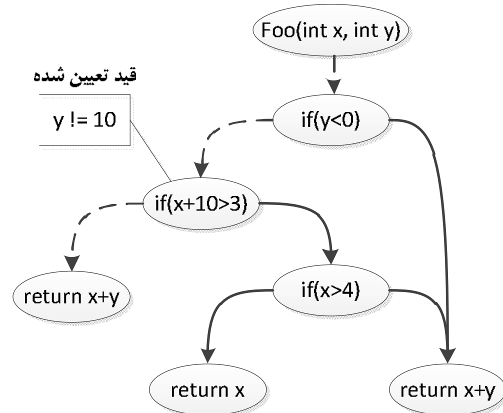
- **اجرای نمادینی:** روش دیگری که در زمینه‌ی تولید داده‌های آزمون ارائه شده است، اجرای نمادینی به‌صورت پویاست. به‌طور کلی، می‌توان گفت که این روش مانند روش اجرای نمادینی به‌طور ایستا است؛ اما در این روش از گراف کنترل جریان استفاده می‌شود. ابتدا از گراف مورد نظر مسیری دلخواه انتخاب می‌شود و سپس بر روی آن مسیر، قیدهایی که از تولید نتایج یکسان با برنامه‌ی اصلی جلوگیری می‌کند، تعیین می‌شود. داده‌ی مورد نظر برای آنکه بتواند از مسیر انتخاب‌شده عبور کند، بایستی قیدهای تعیین‌شده را ارضا کند [١٥،١٦]. همواره در انتخاب قیدها باید به این نکته توجه کرد که عبارت تغییریافته در نسخه‌ی خطادار با عبارت برنامه‌ی اصلی معادل نگردد؛ برای این منظور باید قانون زیر را در انتخاب قید مدّنظر داشت.

$$\text{original expression} \neq \text{mutated expression} \quad (۲)$$

برای سهولت فهمیدن این روش، می‌توان به مثال شکل (۸) توجه کرد.

```
1   public int F3(int x, int y, int C)
2   {
3       if(y<0)
4       {
5           if(x+y>3)
6               return x+y;
7           else
8           {
9               if(x>4)
10                  return x;
11              else
12                  return x+y;
13          }
14      }
15      else
16          return x+y;
17  }
```

(الف): کد اصلی



(ب): گراف نسخه‌ی خطادار

شکل(۸): مثالی از اجرای نمادینی به‌طور پویا – نسخه خطادار تولیدشده، برابر است با جایگزینی دستور ۵ با $if(x+10>3)$

فرض کنید در مثال فوق، داده‌ی اولیه برابر بـا $x=8$, $y=10$ و مسیر انتخاب‌شده نیز برابر خطوط مقطع باشد. بـه‌دلیل آنکـه مقدار پارامتر y برابر با ۱۰ است، عبارت برنامه‌ی اصلی با عبارت نسخه‌ی خطادار برابـر خواهـد شـد ($x+y>3 \neq x+10>3$). از آنجایی‌که هدف، جلوگیری از معادل‌شدن نسخه‌های خطـادار بـا برنامه‌ی اصلی است، باید طبق قانون ۱ قیدی تعیین گـردد تـا از تولید عدد ۱۰ برای پارامتر y جلوگیری شود ($y!=10$)؛ لذا بایـد به گره شروع بازگشت نمود و داده‌ای مطابق با قیـد تعیین‌شـده تولید کرد که این امر موجب متفاوت‌شدن مسیر اجرای نسـخه‌ی خطادار با مسیر اجـرای برنامـه‌ی اصـلی و درنهایـت، بـه‌دلیـل متفاوت‌بودن مسـیر اجـرا باعـث کشـته‌شـدن نسـخه‌ی خطـادار می‌شود.

• **تولید داده‌ی مبتنی بر جســتجو:** روش تولیـد داده‌ی آزمـون مبتنـی بـر جسـتجو، بـر جسـتجوی فضـای داده بـا اسـتفاده از روش‌های تکاملی مانند الگـوریتم ژنتیـک[1]، باکتریولوژیکـال[2] و تپه‌نوردی[3] تأکید دارد. این روش با به‌کاربستن روش‌های تکاملی تلاش می‌نماید فضای داده را به‌طور بهینه کاوش کند تا توانـایی تولید داده‌هایی را داشته باشد که بتوانند به دسـتور جهـش شـده رسیده و موجب اجـرای آن گردنـد [۹،۷]. بـه مثـال شـکل (۹) توجه نمایید:

```
1   public void F1(int x, int y)
2   {
3       if(x>y)
4       {
5           if(y>10)
6               x=100;
7       }
8       else
9           y=x+1;
10  }
```

شکل(۹): مثالی از تولید داده مبتنی بر جستجو– نسخه‌ی خطادار تولیدشده برابر است با جایگزینی دستور ۹ با $y=x-1$

با توجه به مثال فوق، فرض کنید داده‌ی اولیه $x=1111(15)$ و $y=1100(12)$ باشـد و دسـتور جهـش‌شـده‌ی مـورد هـدف، دستور خط ۹ باشد. هدف آن است که داده‌ی مـورد نظر بتوانـد به این دستور جهش‌شده برسد و موجب اجرای آن شود تا منجر به خروجی نادرست نسبت به برنامه‌ی اصلی گردد؛ اما اگر توجه نمایید این داده توانایی رسیدن به دستور مورد هـدف را نـدارد و در دستور خط ٦ متوقف می‌شود؛ لذا از روش‌های تکاملی ماننـد ژنتیک می‌توان بهره برد تا داده‌ای تولید نمود که بتواند به مقصود مورد نظر برسد.

1. Genetic
2. Bacteriological
3. Hill Climbing

الگوریتم ژنتیک از دو عملگر اصلی تشکیل شده است: ترکیب[1] و جهش[2]. در عملگر ترکیب داده‌ها ابتدا به معادل باینری تبدیل شده و نقطه‌ای به‌طور تصادفی جهت عمل ترکیب انتخاب می‌شود. فرض نمایید نقطه‌ی ترکیب مورد نظر ۲ باشد، در این حالت بعد از عمل ترکیب، معادل باینری دو متغیر به $x=1100$ و $y=1111$ تغییر می‌یابد. پس از اعمال عملگر ترکیب، نوبت به عملگر جهش می‌رسد. در عملگر جهش، یک بیت به‌طور تصادفی انتخاب و مقدار آن معکوس می‌شود؛ یعنی در صورت ۰ با ۱ و بالعکس در صورت وجود ۱ با ۰ جایگزین می‌شود. اگر در متغیر x بیت چهارم و در y بیت اول انتخاب شده باشد در این صورت داده‌ها به $x=0100$ و $y=1110$ تغییر می‌یابند. پس از اعمال عملگر جهش، داده‌ها به معادل دهدهی خود تبدیل می‌شوند ($x=4$ و $y=14$). اکنون داده‌های جدید تولید شده و مجدداً بر روی نسخه‌های خطادار اجرا می‌شوند. هدف از تولید داده‌ی جدید، بالابردن توانایی رسیدن به دستور جهش‌شده است که درنتیجه باعث اجرا و کشته‌شدن نسخه‌ی خطادار خواهد شد.

۲.٤. تحقیقات مربوطه

در زیربخش قبل، روش‌های مختلف در زمینه‌ی تولید داده‌های آزمون در آزمون جهشی ارائه شد. در این قسمت، مقاله‌های علمی که در این زمینه صورت گرفته است، به‌طور مختصر توضیح داده شده است.

«Offutt» [۱۷] در تحقیق خود توانست بر اساس نظریه‌های اولیه‌ای که بر روی آزمون جهشی در زمینه‌ی تولید داده‌های آزمون وجود داشت، ابزاری جدید به نام Godzilla مبتنی بر دو مفهوم اعمال قیود و تحلیل جهشی پیاده‌سازی نماید. Godzilla به‌دلیل تولید خودکار داده‌های آزمون، برای مدت زیادی مورد استفاده‌ی آزمون‌گران قرار گرفت. اندکی بعد « DeMillo and Offutt» [۱۲] توانستند Godzilla را با یک سیستم آزمون جهشی پیاده‌سازی‌شده به نام Mothra ترکیب کنند، به‌طوری که Godzilla وظیفه‌ی تولید داده و Mothra مسئولیت تولید نسخه‌های خطادار و مقایسه نتایج اجرا را بر عهده داشتند. آن‌ها همچنین توانستند هزینه‌ی تولید داده در سیستم Mothra را از طریق ترکیب قیود تولید شده برای هر مسیر کاهش دهند [۱۸]. نتیجه کار آن‌ها منجر به کشته‌شدن نسخه‌های خطادار بیشتر با توجه به تولید تعداد داده‌های کمتر شد.

«Offutt et al»[۱٤] الگوریتمی را برای کاهش دامنه تولید داده به‌صورت پویا معرفی کردند و سپس توانستند ایده‌ی خود را به‌صورت یک ابزار پیاده‌سازی کنند [۱۹]. یکی از مسائل سخت و پر اهمیت در آزمون جهشی، کاهش هزینه‌ی اجرایی داده‌ها بر روی نسخه‌های خطادار، از طریق کاهش تعداد داده‌هاست. برای مثال، مجموعه‌ای با ۲۰ داده را فرض نمایید که دارای ۳ داده‌ی بهینه (برای تشخیص خطاها) است. اگر به‌جای اجرای ۲۰ داده، تنها ۳ داده‌ی بهینه را بر روی نسخه‌های خطادار اجرا نمود، هزینه‌ی اجرایی آزمون جهشی، کاهش چشمگیری خواهد داشت. بر این اساس، «Offutt et al» [۲۰] توانستند چند استراتژی اکتشافی برای این منظور ارائه دهند.

«Baudry» [۷] برای اولین‌بار دو الگوریتم تکاملی ژنتیک و باکتریولوژیکال را به‌منظور تولید خودکار داده‌های آزمون در آزمون جهشی با استفاده زبان $C\#$ به‌کار برد. «Ayari et al» [۹] نیز توانستند الگوریتم تکاملی کلونی مورچگان را برای تولید خودکار داده‌ی آزمون مورد استفاده قرار دهند.

اکثر روش‌های تولید داده بر اساس وضعیت (قیود تولیدشده) هر نسخه‌ی خطادار بوده است؛ به عبارت دیگر، هر نسخه‌ی خطادار، داده‌ای مطابق با وضعیت خود دریافت می‌کند. با داشتن تعداد زیادی از نسخه‌های خطادار، تعداد داده‌های تولیدشده افزایش می‌یابد. یک روش برای کاهش تعداد داده‌ها، استفاده‌ی مشترک نسخه‌های خطادار با وضعیت یکسان از یک داده‌ی مشترک است که «Liu et al»[۲۱] روشی را برای رسیدن به این هدف معرفی کردند. «Zhang et al» [۱۵] به‌جای تولید و اجرای مستقل نسخه‌های خطادار، برای هر نسخه‌ی خطادار اطلاعات فراداده‌ای[3] تولید کرده و سپس داده‌هایی به‌صورت نمادینی که توانایی اجرا بر روی این فراداده‌ها را داشته باشند، تولید کردند. آن‌ها به این منظور ابزاری به نام PexMutator در زبان $C\#$ طراحی کردند.

«Papadakis et al» [۱٦] با نگرشی متفاوت اقدام به تولید داده‌ی آزمون نمودند. در این روش، ابتدا تمامی نسخه‌های خطادار تولید می‌شوند و سپس از بین نسخه‌های موجود، آن‌هایی که قابلیت کشته‌شدن توسط داده‌های تولیدشده را دارند، به‌صورت اکتشافی انتخاب می‌شوند. آن‌ها تحقیق خود را گسترش داده و از سه ابزار تولید خودکار داده‌ی آزمون در زبان Java به نام‌های JPF-SE، Concolic و Etoc استفاده کردند و نتایج حاصل را با یکدیگر مقایسه کردند [۲۲].

1. Crossover
2. Mutation
3. Meta-data

«Fraser and Zeller»[۲۳] آزمون جهشی را در زمینه‌ی شیءگرایی اجرا کردند. برای این منظور، نسخه‌های خطاداری در سطح کلاس تولیدشده و داده‌ها بر روی آن‌ها به‌صورت شیءگرا تولید و اجرا می‌شوند. «Mark Harman et al» [۱۰] یک معماری ترکیبی جهت تولید داده‌های آزمون به نام SHOM طراحی کردند که در آن از روش‌های *DSE* و *SBTG* به‌صورت پویا استفاده شده است. آن‌ها همچنین تأثیر کاهش دامنه‌ی داده‌های ورودی از طریق حذف متغیرهای کم اهمیت را در روش SBTG با استفاده از SDR بررسی کردند [۱۱].

با توجه به توضیحات ارائه‌شده، جدول (۱) تحقیقات انجام‌شده را به‌ترتیب سال با یکدیگر مقایسه کرده است. همان‌طور که مشهود است، این جدول در پنج زمینه مقاله‌های مورد نظر را ارزیابی کرده است. این زمینه‌ها به‌ترتیب عبارت‌اند از:

- «سال» انتشار مقاله؛
- «روش» استفاده‌شده در تولید داده؛
- «زبان» برنامه‌نویسی مورد استفاده در سیستم آزمون جهشی؛
- «روش تزریق خطا» که به‌صورت تک‌خطایی یا چندخطایی نسخه‌های خطادر را تولید می‌کنند؛
- «روش کشتن نسخه‌ی خطادار» که به‌صورت ضعیف یا قوی انجام می‌شود.

آنچه می‌توان از این جدول برداشت کرد آن است که تحقیقات بسیار کمی در این حوزه صورت گرفته است و تولید داده‌های آزمون بهینه، همچنان به‌عنوان چالشی در این زمینه محسوب می‌گردد.

جدول (۱): مقاله‌های انجام شده در زمینه تولید داده‌های آزمون جهشی

نویسنده	سال	روش	زبان	روش تزریق خطا	روش کشتن نسخه خطادار
Offutt [۱۷]	۱۹۸۸	SDR	Fortran	تک‌خطا	ضعیف
DeMillo and Offutt [۱۲]	۱۹۹۱	SDR	Fortran	تک‌خطا	ضعیف
DeMillo and Offutt [۱۸]	۱۹۹۳	SDR	Fortran	تک‌خطا	قوی
Offutt et al. [۱٤]	۱۹۹٤	DDR	Fortran	تک‌خطا	ضعیف
Offutt et al. [۲۰]	۱۹۹٥	SDR	Fortran	تک‌خطا	ضعیف
Offutt et al. [۱۹]	۱۹۹۹	DDR	Fortran	تک‌خطا	ضعیف
Baudry [۷]	۲۰۰۲	SBTG	C#	تک‌خطا	ضعیف
Liu et al. [۲۱]	۲۰۰٦	DDR	C	تک‌خطا	ضعیف
Ayari et al. [۹]	۲۰۰۷	SBTG	Java	تک‌خطا	ضعیف
Mark Harman et al. [۱۱]	۲۰۰۷	SDR / SBTG	C	تک	ضعیف
Zhang et al. [۱٥]	۲۰۱۰	DSE	C#	تک‌خطا	ضعیف
Papadakis et al. [۱٦]	۲۰۱۰	DSE	C	تک‌خطا	ضعیف
Papadakis et al. [۲۲]	۲۰۱۰	DSE	Java	تک‌خطا	ضعیف
Fraser and Zeller [۲۳]	۲۰۱۰	SBTG	Java	تک‌خطا	ضعیف
Mark Harman et al. [۱۰]	۲۰۱۱	DSE / SBTG	C	تک‌خطا / چندخطا	ضعیف / قوی

۳.٤. معیارهای استاندارد ارزیابی

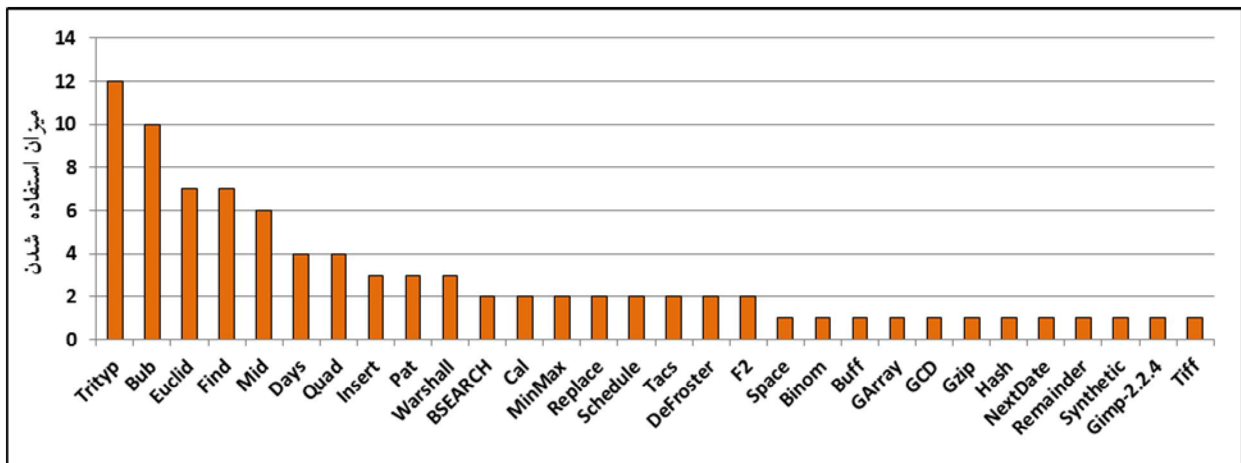
آنچه مسلم است، بسیاری از تحقیق‌های علمی، نیازمند پیاده‌سازی و ارزیابی نتایج می‌باشند؛ اما برای آنکه بتوان ارزیابی معتبری از نتایج داشت، عموماً معیارهای استانداردی در هر زمینه‌ی علمی استفاده می‌شود. تحقیق‌های صورت‌گرفته در زمینه‌ی تولید داده‌های آزمون در آزمون جهشی نیز نیازمند این‌گونه ارزیابی‌ها می‌باشد. در اینجا جهت آشنایی بیشتر با این معیارها، تعدادی از آن‌ها که توسط مقاله‌های جدول (۱) مورد استفاده قرار گرفته، در جدول (۲) نشان داده شده است.

یکی از سؤالاتی که ممکن است در ذهن ایجاد شود، آن است که معیار اصلی در انتخاب برنامه‌های استاندارد در آزمون جهشی چیست؟ برای پاسخ‌دادن به این سؤال، لازم است سیر تحقیقات انجام‌شده را به دو دوره‌ی ابتدایی و پیشرفته تقسیم نمود. در دوره ابتدایی، به‌دلیل نوپابودن آزمون جهشی، تأکید بر روی ایجاد خطا در گزاره‌های انتسابی ساده بود؛ ازاین‌رو، معیارهای

ارزیابی که در این دوران استفاده می‌شـد، معیارهـایی سـاده امـا دارای گزاره‌های انتسـابی متفـاوت بودنـد، ماننـد Tritypa ،Find، Euclid ،Bub و غیره؛ اما با گسترش تحقیقـات و ورود بـه دوره پیشرفته، نگرش اولیه تغییر کرد و نیاز بـه اسـتفاده از معیارهـای پیچیده‌تر، بیش از پیش احساس شد. بر این اساس، محققـین بـه اسـتفاده از معیارهـایی کـه دارای سـاختارهای شـرطی بیشـتر و پیچیده‌تری (دارای مسیرهای متعدد در ساختار فلوچارت) بودند روی آوردند، مانند Gimp (یک نرم‌افـزار مـتن‌بـاز تحـت GNU برای ویرایش تصاویر)، Synthetic (ترکیبی از ۲۰ داده‌ی ورودی و ۳۹ دستور شرطی مطابق با داده‌های ورودی، طراحی‌شده برای اجرای آزمایشی خـاص)، DeFroster (برنامـه‌ای دارای سـاختار شروطی متفاوت برای جلوگیری از تشکیل بخار بر روی شیشـه خودرو) و غیره. این امر سبب شد تا داده‌هـا عـلاوه بـر توانـایی تشخیص دستور خطادار، از جهت توانایی انتخاب مسیر صـحیح برای رسیدن به دستور خطادار مورد ارزیابی قرار گیرنـد. هـدف دیگر استفاده از برنامه‌های بزرگ‌تر به‌عنوان معیار ارزیـابی، ورود کاربردی آزمون جهشـی بـه فرآینـد تولیـد و توسـعه نـرم‌افـزار می‌باشد. لازم به ذکر است که اجباری بـرای انتخـاب معیارهـای استاندارد جدول (۲) توسط محققین وجـود نـدارد؛ امـا توصـیه می‌شود معیارهـای انتخـابی دارای شـروط پیچیـده و مسـیرهای انتخابی متعدد باشد. نمودار شکل(۱۰) نیز نسبت میزان استفاده‌ی معیارها توسط تحقیقات جدول (۱) را نشان می‌دهد.

جدول(۲): معیارهای استاندارد ارزیابی در آزمون جهشی

نام معیار	تعداد خطوط برنامه	تعداد عبارت‌های شرطی	توضیحات
Bub	۳۵	٦	مرتب سازی حبابی
Cal	۳۲	٦	چاپ کامل تقویم برای یک ماه یا سال خاص
Euclid	٤٣	٦	بزرگترین مقسوم علیه مشترک بین دو عدد
Find	۸۸	۲۲	جستجو
Insert	١٦	۷	درج در آرایه
Mid	٤٣	۱۰	میانه عدد
Pat	۲۳	٤	تشخیص الگو
Quad	۹	۹	پیدا کردن ریشه های درجه دو
Tritypa	۸۸	۳۲	تشخیص مثلث
Warshall	۳۰	٥	پیدا کردن کوتاهترین مسیر در یک گراف
BSEARCH	۳۳	٥	جستجوی دودویی
Days	٨٦	۲۸	محاسبه تعداد روز بین دو تاریخ خاص
MinMax	٤٤	٦	محاسبه بزرگترین و کوچکترین مقدار آرایه
Binom	۳٥	٤	حل معادله دو جمله ای
NextDate	٤٣	۱۰	محاسبه تاریخ روز بعدی یک تاریخ خاص
Remainder	٥۰	۹	رویه محاسبه باقیمانده تقسیم
Replace	٥۰۰	٤٣	جستجو و جایگزینی یک الگو
Tacs	۱۲۰	۲٥	سیستم کنترل و جلوگیری از برخورد هواپیما
Schedule	۲۰۰	۳۰	زمانبندی اولویتی
GCD	٤٣	٤	محاسبه بزرگترین مقسوم علیه مشترک در یک آرایه
DeFroster	۲۳۷	٥٦	برنامه‌ی سیتم ضد بخار ماشین
F2	٤۱۸	۲٤	برنامه‌ی سیتم ضد بخار ماشین
Hash	۱۰۱۱	٦٨	تابع درهم‌سازی (رمزنگاری)
Space	۹٥٦٤	نامشخص	سیستم آژانس فضایی اروپا
Buff	۱۳۷۱	۷٥	کنترل ربات
Garray	۸۰۸	٥۳	یک ساختار داده‌ای جدید تحت GNU
Gzip	۷۹۳۳	۲٦٦۱	نرم‌افزار فشرده سازی تحت GNU
Synthetic	۱۳۸	۷٨	یک برنامه برای اجرای آزمایشی خاص
Gimp-2.2.4	٨٦۷	٤٨	نرم‌افزار ویرایش تصویر تحت GNU
Tiff	۱۸۲	٤٨	برنامه گرافیکی – کارباتصاویر Tiff



شکل(۱۰): نمودار میزان استفاده معیارهای استاندارد توسط تحقیقات جدول (۱)

آنچه مشهود است، بیشتر تلاش‌های صورت‌گرفته بر روی برنامه‌های کوچک می‌باشد و تلاش برای استفاده از برنامه‌های بزرگ در آزمون جهشی بسیار کم بوده است؛ لذا در حال حاضر، تلاش بسیاری از محققین بر ارزیابی آزمون جهشی در برنامه‌های بزرگ و عملی‌کردن تولید داده‌های آزمون جهت استفاده‌ی آزمون‌گران در صنعت نرم‌افزار است.

۵. نتیجه‌گیری

این مقاله در ابتدا فرآیند کلی آزمون جهشی و ملزومات (تحلیل جهشی، عملگرهای جهشی) آن را معرفی کرد. سپس کشتن نسخه‌های خطادار همراه با یک مثال، به دو روش ضعیف و قوی بیان شد. در ادامه، فعالیت‌های صورت‌گرفته در زمینه‌ی تولید داده‌های آزمون همراه با مثالی از روش‌های مورد استفاده (SDR, SSE, DDR, DSE و SBTG) آورده شد.

هدف اصلی این مقاله، تهیه‌ی طرحی جامع از تحقیقات و روش‌های تولید داده‌ی آزمون در آزمون جهشی بوده است. از طرف دیگر، این مقاله قصد دارد تا با تأکید بر تعداد محدود تحقیقات صورت‌گرفته در زمینه‌ی تولید داده‌های آزمون، اهمیت انجام تحقیق و ارائه‌ی روش‌های نوین جهت تولید داده‌های بهینه با هزینه‌ی کم را نشان دهد.

با توجه به تحقیقات انجام‌شده در زمینه‌ی آزمون جهشی، می‌توان نتیجه گرفت که درصد کمی از این تحقیقات در رابطه با تولید داده‌های آزمون می‌باشد؛ درحالی‌که تولید داده‌ی بهینه در فرآیند آزمودن نرم‌افزار بسیار پر اهمیت است؛ زیرا هرچه داده‌ها توانایی آشکارسازی خطاهای بیشتر را داشته باشند، موجب افزایش کیفیت محصول نرم‌افزاری می‌شوند؛ ازاین‌رو، در سال‌های اخیر، محققین در تلاش هستند تا بتوانند این مورد را عملی‌تر ساخته و از آن در آزمودن نرم‌افزارهای بزرگ در صنعت تولید نرم‌افزار استفاده کنند.

با توجه به روند انتخابی معیارهای ارزیابی توسط محققین، می‌توان نتیجه گرفت که معیارها از برنامه‌های کوچک (دارای دستورات انتسابی ساده و اغلب غیر کاربردی و آزمایشی) در حال تبدیل به برنامه‌های کاربردی و بزرگ (دارای ساختار شرطی بزرگ و پیچیده) می‌باشد؛ لذا می‌توان امیدوار بود که آزمون جهشی در آینده به‌صورت کاربردی در فرآیند توسعه نرم‌افزار استفاده شود. محققین علاقمند نیز می‌توانند برای انجام تحقیقات خود، از برنامه‌های جدید استفاده کرده و معیارهای ارزیابی جدیدی را به عنوان استاندارد معرفی کنند.

از طرفی دیگر، با توجه به مقایسه‌ی تحقیقات انجام‌شده، می‌توان نتیجه گرفت که بیشتر آن‌ها بر روی کشتن ضعیف نسخه‌های خطادار، همراه با روش تزریق یک خطا (تک‌خطا) صورت گرفته و تحقیقات کمی بر روی روش تزریق چندخطا همراه با کشتن قوی انجام شده است. همچنین می‌توان دریافت که محققین در سال‌های اخیر به استفاده‌ی ترکیبی از روش‌های تولید داده برای تولید داده‌های آزمون روی آورده‌اند.

مراجع

[1] R. Lipton, "Fault Diagnosis of Computer Programs," student report, Carnegie Mellon Univ., 1971.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer", IEEE Computer, vol.11, issue.4, pp.34–41, 1978.

[3] P. J. Walsh. A measure of test completeness. PhD thesis, State University of New York at Binghamton, 1985.

[4] P. G. Frankl, S. N. Weiss, and C. Hu," All-uses vs. mutation testing: An experimental comparison of effectiveness", The Journal of Systems and Software, vol. 38, issue.3, pp.235–253, Sept. 1997.

[5] J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing", Software: Practice and Experience, vol.26, issue.2, pp.165–176, Feb. 1996.

[6] J. Offutt and R. H. Untch, Mutation 2000: Uniting the orthogonal. In Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, pp. 45–55, 2000.

[7] B. Baudry, F. Fleurey, J.M. Jezequel, and Y. Le Traon, "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment", Proc. 13th Int'l Symp. Software Reliability Eng., pp. 195-206, Nov. 2002.

[8] A.J. Offutt, J. Voas, and J. Payn, "Mutation Operators for Ada", Technical Report ISSE-TR-96-09, George Mason Univ., 1996.

[9] K. Ayari, S. Bouktif, and G. Antoniol. Automatic Mutation Test Input Data Generation via Ant Colony. InProceedings of the Genetic and Evolutionary Computation Conference (GECCO'07),pp.1074–1081, London, England, 7-11 July 2007.

[10] M. Harman, Y.Jia and W.Langdon, "Strong higher order mutation-based test data generation", Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp.212-222, Szeged, Hungary,2011.

[11] M. Harman,Y. Hassoun, K. Lakhotia, P. McMinn,J. Wengener , "The impact of input domain reduction on search-based test data generation", Proceeding ESEC-FSE '07 Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering,pp. 155-164, New York, USA, 2007.

[12] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation.IEEE Transactions on Software Engineering,vol.17, issue.9, pp.900–910,September1991.

[13] J.King, T.Watson , "Symbolic execution and program testing", Communications of the ACM, Vol.19 , Issue.7 , New York, USA, July 1976.

[14] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Approach for Test Data Generation: Design and Algorithms. Technical Report ISSE-TR-94-110, George Mason University, Fairfax, Virginia, 1994.

[15] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In Proceedings of the 26th International Conference on Software Maintenance (ICSM'10),Timisoara, Romania, September 2010.

[16] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. InProceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10), California, USA, November 2010.

[17] A. J. Offutt.Automatic Test Data Generation.Phd thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988.

[18] R.A. DeMillo and A.J. Offutt, "Experimental Results from an Automatic Test Case Generator,"ACM Trans. Software Eng. And Methodology,vol. 2, no. 2, pp. 109-127, Apr. 1993.

[19] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. Software:Practice and Experience, vol.29, issue.2, pp.167–193, February 1999.

[20] A.J. Offutt, J. Pan, and J.M. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets,"Proc. 12th Int'l Conf. Testing Computer Software, pp. 111-123, June 1995.

[21] M.-H. Liu, Y.-F. Gao, J.-H. Shan, J.-H. Liu, L. Zhang, and J.-S. Sun. An Approach to Test Data Generation for Killing Multiple Mutants. In Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06),pp.113–122,Philadelphia, Pennsylvania, USA, 24-27 September 2006.

[22] M. Papadakis, N. Malevris, and M. Kallia. Towards Automating the Generation of Mutation Tests. In Proceedings of the 5th Workshop on Automation of Software Teste (AST'10),pp.111–118,CapeTown, South Africa, 3-4 May 2010. ACM.

[23] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. InProceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10),pp147–158,Trento,Italy, 12-16 July 2010 ISSTA '10. ACM.