فصل ۱۷ فنون آزمون نرم افزار

مفاهیم کلیدی (مرتب بر حروف الفبا)

آزمون آرایه راستگوشه(متعامد) ، آزمون پذیری ، آزمون جعبه سیاه ، آزمون حلقه ، آزمون رفتاری ، آزمون ساختار کنترل ، آزمون مسیر پایه ، اهداف اصلی آزمون ، پیچیدگی چرخشی (سیکلوماتیک) ، تجزیه هم ارزی ، تحلیل مقادیر مرزی (BVA) ، گرافهای جریان

KEY CONCEPTS

basic path testing , behavioral testing , block-box testing , BVA , Control structure testing , cyclomatic complexity , equivalence partitioning , flow group , loop testing , OA testing , testability , testing objectives

نگاه اجمالی

آزمون نرم افزار چیست؟ وقتی کد منبع ایجاد گردید، باید نرم‌افزار آزمون شود تا خطاهای احتمالی موجود قبل از تحویل به مشتری رفع شوند. هدف شما طراحی یکسری موارد آزمون است که یافتن خطاها را به گونه ای مناسب پوشش دهد، اما چگونه؟ این‌جا جایی است که فنون آزمون نرم‌افزار وارد عمل می‌شوند. این تکنیک‌ها راهنمای سیستماتیکی برای طراحی آزمون‌هایی می‌کنند که: ۱) منطق درونی اجزای نرم‌افزاری را بررسی کرده و ۲) حوزه ورودی و خروجی برنامه را برای مشخص کردن خطاهای عملکرد برنامه، رفتار و عملیات، می آزمایند.

چه کسی عهده دار آزمون نرم افزار می باشد؟ در طول مراحل اولیه آزمون، مهندس نرم‌افزار کلیه آزمونها را انجام می‌دهد. با پیشرفت فرآیند ممکن است متخصصان آزمون نیز درگیر شوند.

چرا انجام این امر از اهمیت برخوردار است؟ بازبینی‌ها و دیگر فعالیتهای SQA می‌توانند خطاها را مشخص کرده و این کار را می‌کنند اما کافی نیستند. هر زمان که برنامه اجرا می‌شود، مشتری آن را می آزماید. بنابراین باید برنامه را قبل از این‌که به مشتری برسد با هدف خاص یافتن خطاها و از بین بردن آنها، آزمود. به‌منظور یافتن بیشترین مقدار احتمالی خطا، آزمونها باید به‌صورت نظام مند صورت گرفته و موارد آزمون با استفاده از فنون اصولی طراحی شوند.

مراحل کار چیست؟ نرم‌افزار از دو دیدگاه مختلف آزموده می‌شود: ۱) منطق درونی برنامه با استفاده از آزمون White Box و فنون آن، اجرا می‌شود. مقتضیات نرم‌افزاری با استفاده از فنون طراحی مورد

آزمون Black Box آزموده می‌شوند. در هر دو مورد، هدف یافتن حداکثر خطا با حداقل تلاش و زمان است.

محصول کار چیست؟ مجموعه‌ای از موارد آزمون، که برای اجرای مقتضیات برونی و منطق درونی طراحی شده‌اند، طرح و ثبت شده است، نتایج مورد انتظار مشخصند و نتایج واقعی ثبت می‌شوند.

چگونه مطمئن شوم که کار را به‌درستی انجام داده‌ام؟ در هنگام شروع آزمون، دیدگاه خود را تغییر دهید. سخت تلاش کنید تا موارد آزمون طراحی نرم‌افزاری را به شکلی اصولی در هم‌بریخته و موارد آزمون را که برای تأمل بودن ایجاد کرده‌اید، بازبینی کنید.

اهمیت آزمون نرم‌افزار و پیامدهای آن را با توجه به کیفیت نرم‌افزار نمی‌توان بیش از حد مورد تأکید قرار داد. به نقل از دوچ [DEU79][1]

توسعه سیستمهای نرم‌افزاری مستلزم یک‌سری فعالیتهای تولیدی است که در آن فرصتهایی برای به‌وجود آمدن خطاهای انسانی بسیار متعدد است. ممکن است در همان اوایل کار که اهداف ... خطا رخ دهد، ممکن است به‌طور اشتباهی یا ناقص مشخص شود، همان‌گونه که در مراحل طراحی و توسعه بعدی ممکن است رخ دهد ...

به‌خاطر ناتوانی انسان در انجام کارها و برقراری ارتباط به‌صورت کامل، توسعه نرم‌افزار با کار تضمین کیفی همراه است.

آزمون نرم‌افزار یک عنصر مهم برای تضمین کیفیت آن است و نمایان‌گر بازبینی نهایی مشخصه‌ها، طراحی و ایجاد کد است.

افزایش مواجهه با نرم‌افزار به‌عنوان عنصر سیستم و هزینه‌های همراه مربوط به عملکرد نادرست نرم‌افزار نیروهای متخصص را وادار می‌سازد که از طریق آزمون، طراحی را به‌خوبی انجام دهند. برای یک سازمان تولید نرم‌افزار، تخصیص ۳۰ تا ۴۰ درصد از نیروی کل پروژه برای انجام آزمون، کار غیر معمولی نیست. نهایتاً، آزمون نرم‌افزار ارزیابی شده توسط انسان (مثل کنترل پرواز، بررسی راکتور هسته‌ای) می‌تواند سه تا پنج برابر دیگر مراحل مهندسی نرم‌افزار مربوطه باشد.

در این فصل با اصول بنیادی آزمون نرم‌افزار و فنونی برای طراحی مورد آزمون آشنا می‌کنیم.

نقل قول

یک برنامه کاری، زیبایی فریبکار باقی خواهد ماند. روبرت دان

۱۷-۱ اصول و مبانی آزمون نرم افزار

آزمون، نمایان‌گر یک نا بهنجاری جالب برای مهندس نرم‌افزار است. در طول فازهای اولیه تعریف و تولید، مهندس تلاش می‌کند از یک مفهوم انتزاعی نرم‌افزاری ساخته و به یک محصول ملموس و مادی برسد. اکنون موقع آزمون است. مهندس یک‌سری آزمونهایی ارائه می‌دهد که به قصد منهدم کردن نرم‌افزار

1.Deutsch,M.

ساخته شده است. در واقع، آزمون، مرحله‌ای از فرآیند نرم‌افزاری است که می توان آن را بیشتر ویران‌کننده دانست تا سازنده.

مهندسین نرم‌افزار ذاتاً افراد سازنده‌ای هستند. آزمون مستلزم این است که تولیدکننده نکات و عبارات از پیش شناخته شده‌ای را که در مورد درست بودن نرم‌افزار تازه تولید دارد، دور ریخته و بر برخورد عقایدی که بعد از برملا شدن خطاها به‌وجود می‌آید، غلبه کند. بیزر این موقعیت را به‌طور مؤثر این‌گونه شرح می‌دهد:[BEI90] [1]

این فرض خیالی وجود دارد که اگر ما به هنگام برنامه‌نویسی واقعاً خوب عمل کنیم، هیچ‌گونه اشکالی پدید نمی‌آید.اگر می‌توانستیم کاملاً حواسمان را متمرکز کنیم، اگر همهٔ افراد از برنامه‌سازی ساختیافته استفاده می‌کردند و همین‌طور از طراحی بالا به پایین، جداول تصمیم‌گیری، اگر برنامه‌ها به SQUISH نوشته شده بودند، اگرگلولهٔ نقره‌ای (راه‌حل برجسته) درست را داشتیم، هیچ اشکالی به‌وجود نمی‌آمد. این افسانه است. اشکالات وجود دارند، چون ما کارمان را بد انجام می‌دهیم و اگر در این مورد بد باشیم، احساس گناه می‌کنیم. بنابراین آزمودن و طراحی مورد آزمون، مجوز شکست است که مقداری احساس گناه در ما القاء می‌کند. و یکنواختی و ملالت این آزمونها مجازات اشتباهات ماست. مجازات برای چه؟ برای انسان بودن؟ گناه برای چه؟ برای شکست در رسیدن به تکامل غیر انسانی؟ برای عدم تشخیص و تمایز بین آن‌چه برنامه‌نویس دیگر به آن فکر می‌کند و آن‌چه که بیان می‌دارد؟ برای شکست در نداشتن حالت تله‌پاتیک؟ برای حل نکردن مشکلات ارتباطات انسانی که به مدت چهل قرن دور ما گرفته‌اند؟

آیا باید آزمون حس گناه را در ما القاء کند؟ آیا آزمون واقعاً نابودگر است؟ پاسخ به این سؤالات کلمه "نه" است. در هر حال، اهداف آزمون تا حدی از آن‌چه که ممکن است انتظار داشته باشیم، متفاوت است.

۱۷-۱-۱اهداف اصلی آزمون

پرسش

اهداف اصلی، هنگام آزمون نرم افزار کدام هاست؟

در یک کتاب عالی که در مورد نرم‌افزار نوشته شده بود، گلن مایرز چند را قانون را برمی‌شمرد که می‌توانند به‌خوبی در خدمت اهداف آزمون قرار گیرند:[MYE79] [2]

۱- آزمون، فرآیند اجرای یک برنامه به قصد یافتن خطاهاست.

۲- یک مورد آزمون خوب آن است که به احتمال بالا یک خطای کشف نشده را بیابد.

۳- یک آزمون مؤفق، آزمونی است که یک خطای نامشخص را بیابد.

اهداف فوق نشان‌گر تغییر عمده‌ای در دیدگاه افراد است. آنها نظریه معمول قبلی را مبنی بر این‌که یک آزمون خوب آن است که در آن خطایی نمایان نشود را رد می‌کنند.

1.Beizer,B.

2.Myer,G.

نقل قول

در نرم افزار، خطاها بیش از دیگر فن آوریها، مشترک، نافذ و مشکل آفرین می باشند. دیوید پارنار

اگر آزمونی با موفقیت صورت گرفت (طبق اهداف بیان شده فوق)، آن وقت خطاهایی در نرم‌افزار پیدا می‌شوند. مزیت دوم این است که آزمون نشان می‌دهد که عملیات نرم‌افزاری طبق مشخصه منظور شده، عمل می‌کنند. یعنی آن‌دسته از شرایط عملکردی و رفتاری برآورده شده‌اند. علاوه بر آن، داده‌های جمع‌آوری شده بر اساس آزمون، یک نشان‌گر خوب برای قابلیت اطمینان به برنامه و خوب بودن کیفیت آن در مجموع است. اما آزمون نمی‌تواند نبود خطا و نقایص را نشان دهد، بلکه تنها خطاها و نواقصی را که موجودند نشان می‌دهد.

۱۷-۱-۲اصول آزمون

قبل از به‌کارگیری روش‌هایی در طراحی موارد مؤثر برای آزمون، مهندس نرم‌افزار باید اصول مقدماتی را در مورد هدایت کار آزمون نرم‌افزار بداند. دیویس[DAV95] مجموعه‌ای از اصول[۱] را بیان می‌دارد که برای استفاده در این کتاب مهیا شده‌اند:

[DAV95][۲]

- **تمام آزمونها باید تا رسیدن به نیازمندیهای مورد نظر مشتری و مصرف‌کننده، قابل پی‌گیری باشند.** همان‌گونه که دیده‌ایم، هدف از انجام آزمون برنامه مشخص کردن خطاهاست. این‌طور معلوم است که نقایص جدی‌تر از دیدگاه مصرف‌کننده آنهایی است که باعث عدم دسترسی کاربر به اهداف مورد نظرش می‌شود.
- **آزمونها باید مدت طولانی قبل از شروع آن، برنامه‌ریزی شوند. این کار (فصل ۸) را می‌توان به‌محض تکمیل شدن مدل شرایط مورد نظر، شروع نمود.** تعریف دقیق موارد آزمون می‌تواند به‌محض ریخته شدن مدل طراحی آغاز گردد. بنابراین تمام آزمونها می‌توانند قبل از ایجاد هر گونه برنامه‌ای، برنامه‌ریزی و طراحی شوند.
- **به‌کارگیری اصل پارتو در مورد آزمون نرم‌افزار.** اگر به‌صورت ساده بیان کنیم، اصل پارتو اشاره دارد که ۸۰ درصد تمام خطاهای مشخص شده در طول آزمون احتمالاً در ۲۰٪ از تمام جزء های برنامه قابل پی‌گیری هستند. البته مسئله این است که این جزءهای مورد شک را جدا کرده و آنها را کاملاً آزمود.
- **آزمون باید از جزء شروع شده و کم‌کم به‌طرف آزمون هایی در سطح وسیع برسد.** اولین آزمون هایی که معمولاً طراحی و اجرا می‌شوند، بر جزءهای منفرد، متمرکز می‌شوند. با پیشرفت

---

۳. تنها یک زیر مجموعه کوچک از اصول آزمون داویس در اینجا آورده شده است. برای اطلاعات بیشتر، به [DAV95] مراجعه نمایید.

2.Davis,A.

آزمون، این نقطه تمرکز روی تلاشی برای یافتن خطاهایی در خوشه‌های منسجم‌تر، معطوف می‌شود و نهایتاً به کل سیستم می‌رسد.

- **آزمون های جامع و فراگیر ممکن نیست.** تعداد دگرگونی‌های اساسی در مسیر، حتی برای یک برنامه متوسط بی‌اندازه زیاد است. به‌همین دلیل، ممکن نیست که هر یک از مسیرهای ترکیبی را در طول آزمون، بیازمائیم. ممکن است که به‌اندازه کافی منطق برنامه را آزموده و مطمئن شویم که همه شرایط در طراحی سطح جزءها رعایت شده‌اند.
- **برای این‌که آزمون بیشترین تأثیر را داشته باشد، باید توسط یک شخص ثالث مستقل صورت گیرد.** در مورد کلمات "بیشترین تأثیر" منظورمان این است که بیشترین احتمال یافتن خطاها وجود داشته باشد که هدف اصلی این کار است. بنا به دلایلی که پیش‌تر ذکر شد و به‌طور دقیق‌تر در فصل ۱۸ مورد بررسی قرار می‌گیرند، مهندس نرم‌افزاری که سیستم را خلق نموده بهترین گزینه برای انجام تمام آزمونهای نرم‌افزاری نیست.

۱۷-۱-۳ آزمون پذیری

در شرایط ایده‌آل، مهندس نرم‌افزار یک برنامه کامپیوتری، سیستم یا محصولی با قابلیت آزمون‌پذیری[1] در فکر خود طراحی می‌کند. این کار افرادی را که با آزمونها سر و کار دارند، قادر می‌سازد که موارد آزمون مؤثر[2] را به‌صورت ساده‌تر طراحی کنند. اما منظور جیمز باخ[3] از این کلمه به‌صورت زیر بیان شده است:

آزمون‌پذیری برنامه صرفاً چگونگی انجام آزمون بر روی برنامه کامپیوتری بدون زحمت است. از آن‌جا که این کار بسیار مشکل است، باید توجه داشت که بدانیم چه کارهایی برای مؤثرتر کردن و پر بازده کردن بیشتر آن می‌توان انجام داد. گاهی برنامه‌نویسان مشتاقند کارهایی انجام دهند که به فرآیند آزمون کمک کند و فهرستی از نکات طراحی، مشخصه‌ها و غیره می‌تواند در کارشان به آنها کمک کند.

مطمئناً متریک‌های مهمی وجود دارند که می‌توان از آنها برای ارزیابی آزمون‌پذیری در اکثر جنبه‌ها استفاده نمود. گاهی از قابلیت آزمون‌پذیری برای توجیه این امر استفاده می‌شود که چگونه مجموعه به‌خصوصی از آزمونها، محصول را بررسی و مشکلات آن را مشخص می‌کند. هم‌چنین در ارتش از آن برای نشان دادن این‌که چگونه یک وسیله به راحتی چک شده و در آن زمینه نگهداری می‌شود، استفاده

ارجاع به وب

مقاله ای سپید با عنوان " بهبود آزمون پذیری نرم افزار " در آدرس زیر قرار دارد:

www.stlabs.com/newsletters/testnet/docs/testability.htm

نکات کلیدی

آزمون پذیری حاصل یک طراحی خوب است. طراحی داده، معماری، رابط و جزئیات تفصیلی می توانند به راحتی تحت آزمون قرار گیرند یا آنکه کار را سخت کنند.

1.effective

2.testability

۳. پاراگرافی که به دنبال خواهد آمد، مطابق با نظریه آقای جیمز باخ ۱۹۹۴ می‌باشد، که نخستین بار درگروه خبری comp.software-eng دیده شد. این مطالب با کسب اجازه از ایشان به طبع رسیده است.

می‌گردد. این دو معنی مختلف با معنی «آزمون‌پذیری نرم‌افزار»[1] یکی نیستند. فهرست زیر مجموعه‌ای از مشخصه‌ها را بیان می‌دارد که نهایتاً به یک نرم‌افزار قابل آزمون می‌رسد:

قابلیت عمل. «هر چه بهتر کار کند، بهتر می‌توان آن را آزمون نمود.»

- سیستم دارای چند اشکال است (اشکال‌ها به تحلیل و گزارش هزینه‌های فرآیند آزمون می‌افزایند.)
- هیچ اشکالی اجرای آزمون ها را متوقف نمی‌کند.
- این محصول در مراحل عملکردی تکمیل می‌شود (امکان توسعه و آزمون همزمان وجود دارد.)

قابلیت مشاهده. «آن‌چه که می‌بینید، آن چیزی است که آزمون می‌کنید.»

- خروجی جداگانه‌ای برای هر ورودی ایجاد می‌شود.
- متغیرها و حالات سیستم قابل رؤیت و در طول اجرا قابل پرسش هستند.
- متغیرها و حالات گذشته سیستم مشهود و قابل پرسش هستند. (مثل ثبت تراکنش‌ها)
- تمام عواملی که بر خروجی تأثیر دارند قابل رؤیتند.
- خروجی غلط به راحتی شناسایی می‌شود.
- خطاهای داخلی به‌طور خودکار از طریق مکانیزم خود آزمایی مشخص می‌شوند.
- خطاهای داخلی به‌طور خودکار گزارش می‌شوند.
- کد مرجع در دسترس است.

قابلیت کنترل. «هر چه بهتر نرم‌افزار را کنترل کنیم، بهتر می‌توان آزمون را خودکار و بهینه ساخت.»

- تمام خروجی‌های احتمالی از طریق ترکیب داده‌ها، ایجاد می‌شوند.
- تمام کدها از طریق ترکیبی از داده‌ها، قابل اجرا هستند.
- متغیرها و حالات نرم‌افزاری و سخت‌افزاری را می‌توان مستقیماً به‌وسیله مهندس آزمون، کنترل نمود.
- قالب‌های ورودی و خروجی متناسب و دارای ساختارند.
- آزمون ها را می‌توان به راحتی مشخص، خودکار و باز آفرینی نمود.

قابلیت تجزیه‌پذیری. «با کنترل دامنه آزمون می‌توانیم سریع‌تر مشکلات را مجزا نموده و آزمون های هوشمند مجددی را انجام دهیم.»

- سیستم نرم‌افزاری از روی پیمانه‌های مستقلی ساخته می‌شود.

1.software testability

- می‌توان پیمانه‌های نرم‌افزاری را به‌طور مستقل مورد آزمون قرار داد.

سادگی. «هر چه چیز کم‌تری برای آزمون وجود داشته باشد، سریع‌تر می‌توانیم آن را بیازمائیم.»

- سادگی عملکرد (مثلاً مشخصه مجموعه حداقل چیز لازم برای نیازمندیهاست.)
- سادگی ساختاری (مثلاً سبک معماری پیمانه ای می‌شود تا ایجاد خطاها را محدود کند.)
- سادگی برنامه (مثلاً استاندارد برنامه‌نویسی برای راحتی بازرسی و نگهداری، تعیین و پذیرفته شده)

ثبات. «هر چه تغییرات کم‌تر باشد، اختلال در آزمون کم‌تر است.»

- تغییرات نرم‌افزاری نادرند.
- کنترل شده‌اند.
- این تغییرات آزمونهای موجود را بی‌اعتبار نمی‌کنند.
- نرم‌افزار از این شکستها سر بلند بیرون می‌آید.

قابلیت درک و شناخت. «هر چه اطلاعات بیشتری داشته باشیم، هوشمندانه‌تر آن را می آزماییم.»

- طرح به‌خوبی درک شده.
- ارتباطات بین جزءهای داخلی، خارجی و مشترک به‌خوبی شناخته شده‌اند.
- تغییرات طراحی مورد تبادل‌نظر قرار می‌گیرند.
- مدارک فنی فوراً در دسترسند.
- مدارک فنی به‌خوبی سازمان یافته‌اند.
- مدارک فنی مشخص و دارای جزییات هستند.
- مدارک فنی دقیقند.

این مراتب که توسط باخ پیشنهاد شده‌اند را می‌توان توسط مهندس نرم‌افزار برای تولید پیکربندی نرم‌افزاری به‌کار گرفت (مثل برنامه‌ها، داده‌ها و اسناد) که پاسخ‌گوی آزمون هستند.

در مورد خود آزمونها چه می‌توان گفت؟ کانر، فالک و گوین[KAN93][1] مراتب زیر را در مورد یک آزمون خوب بیان می‌کنند:

پرسش

صفات خاصه یک آزمون خوب کدام هاست؟

۱- **یک آزمون خوب دارای احتمال یافتن خطاهای بیشتری است.** برای نیل به این هدف، آزمون‌گر باید نرم‌افزار را شناخته و سعی کند تصویر ذهنی از چگونگی احتمال شکست ارائه دهد.

۲- **یک آزمون خوب دارای زواید و حاشیه نیست.** زمان و منابع موجود برای آزمون محدودند. هیچ امتیازی در مورد آزمونی که دارای هدفی مثل آزمون دیگر است، وجود ندارد. هر

1 Kaner,C.

آزمون دارای هدف متفاوتی است (حتی اگر به‌طور دقیق متناوب نباشد.) به‌طور مثال، یک پیمانه در نرم‌افزار خانهٔ امن (که در فصول پیش‌تر در مورد آن بحث شد) طوری طراحی شده که کلمه رمز کاربر را برای فعال‌سازی و خاموش کردن سیستم تشخیص دهد. به‌منظور مشخص نمودن خطا در ورودی کلمه رمز، آزمون‌گر یکسری آزمون طراحی می‌کند که آنها یک سری کلمه رمز را وارد می‌سازند. کلمات رمز معتبر و غیر معتبری باید یک مورد عدم موفقیت جداگانه را بررسی کند. مثلاً کلمه رمز غیر معتبر ۱۲۳۴ توسط سیستم برنامه‌ریزی شده پذیرفته نمی‌شود با این فرض که کلمه رمز معتبر ۸۰۸۰ قبول شود. اگر این کلمه رمز قبول شد، سیستم دارای مشکل است. مثلاً کلمه رمز دیگری مثل ۱۲۳۵ وارد شده که دارای همان منظور کلمه رمز ۱۲۳۴ می‌باشد و بنابراین زاید است. داده‌های غیر معتبر ۸۰۸۱ یا ۸۱۸۰ تفاوت زیرکانه‌تری است و تلاش می‌کند وجود خطایی در مورد کلمات رمز نزدیک به کلمه اصلی اما غیر مشابه با کلمه رمز معتبر را تشریح کند.

۳- **یک آزمون خوب باید بهترین از هر نظر باشد.** در یکسری از آزمونهایی که هدف مشابهی دارند، محدودیت‌های زمانی و مرجعی، ممکن است ما را به‌سوی انجام تنها زیر مجموعه‌ای از این آزمونها سوق دهد. در چنین مواردی، آزمونی که دارای بیشترین احتمال مشخص نمودن یکسری خطاست، باید به‌کار رود.

۴- **یک آزمون خوب نه باید زیاد ساده و نه زیاد سخت باشد.** گر چه گاهی ممکن است یکسری آزمونها را در یک مورد جای داد، اما ممکن است اثرات جانبی هر روش، باعث پنهان ماندن خطاها شود. به‌طور کلی، هر آزمون باید جداگانه انجام شود.

نقل قول

در طراحی موارد آزمون تنها یک قانون وجود دارد : تمام خصوصیات را پوشش دهید، اما در انجام آزمون افراط نکنید.

نیو نیو باماورا

۲-۱۷ طراحی موارد آزمون

طراحی آزمونها نرم‌افزار و دیگر محصولات مهندسی را می‌توان یک طراحی اولیه چالش برانگیز از خود محصول دانست. بنا به دلایلی که هم‌اکنون مورد بحث قرار دادیم، مهندسان نرم‌افزار اغلب با در نظر گرفتن آزمون به‌عنوان یک تفکر ثانویه، موارد آزمونی را طراحی می‌کنند که ممکن است حس درستی داشته، اما اطمینان کمی از نظر کامل بودن ایجاد می‌کنند. با به‌خاطر آوردن اهداف آزمون، باید آزمون‌هایی را طراحی کنیم که دارای بیشترین احتمال یافتن مهم‌ترین خطاها در حداقل زمان باشند.

هر محصول مهندسی‌ساز را می‌توان به یکی از دو روش زیر امتحان نمود:

۱- با آگاهی از کارکرد خاصی که این محصول برای آن تولید شده، آزمونهایی را می‌توان انجام داد که هر کارکرد را از نظر عملی بودن کاملاً تشریح کند. در حالی‌که در عین حال برای هر کارکرد جستجویی برای یافتن خطاها انجام می‌دهد.

ارجاع به وب

روزنامه فنون آزمون (TTN) منبعی ممتاز از اطلاعات مربوط به شیوه های آزمون می باشد : www.testworks.com/News/-tn-online/

۲- با آگاهی از کارهای صورت گرفته داخلی در هر محصول، می‌توان آزمونهایی را انجام داد که از جور شدن همه کارها اطمینان یابیم، یعنی عملیات داخلی طبق مشخصات بوده و همه اجزای درونی به اندازه کافی به‌کار گرفته شده‌اند. اولین آزمون، آزمون جعبه سیاه و دومی آزمون جعبه سفید است.

یا در نظر گرفتن نرم‌افزار کامپیوتری، آزمون جعبه سیاه اشاره دارد به آزمون که بر رابط نرم‌افزاری صورت می‌گیرند. گر چه از آنها برای مشخص کردن خطاها استفاده می‌شود، اما از آزمونهای جعبه سیاه برای قابلیت عمل، کارکردهای مختلف نرم‌افزار نیز استفاده می‌شود، یعنی ورودی به‌خوبی پذیرفته شده و خروجی کاملاً تولید شده یا این‌که انسجام اطلاعات برونی، به‌درستی پذیرفته شده است. یک آزمون جعبه سیاه[1] بعضی از جنبه‌های بنیادی سیستم را با توجه کمی به ساختار منطق درونی نرم‌افزار، می آزماید.

آزمون جعبه سفید[2] نرم‌افزاری در مورد بررسی دقیق جزییات رویه‌ای کار صورت می‌گیرد. مسیرهای منطقی در نرم‌افزار با ایجاد موارد آزمونی که مجموعه مشخصی از شرایط و/ یا حلقه‌ها را به‌کار می‌گیرند، مورد آزمون واقع می‌شوند. ممکن است وضعیت برنامه در نقاط مختلف معین کند که آیا وضعیت موردنظر یا برآورده شده یا وضعیت کنونی هم‌خوانی دارد یا خیر؟

در وهله اول به‌نظر می‌رسد که یک آزمون کامل جعبه سفید منجر به برنامه‌های ۱۰۰ درصد درست می‌شود. همه آن‌چه که باید انجام شود عبارتست از تعریف همه مسیرهای منطقی، ایجاد موارد آزمونی بسط آنها برای به‌کارگیری و ارزیابی نتایج یعنی تولید موارد آزمونی برای به‌کارگیری منطق برنامه به‌صورت گسترده. متأسفانه، آزمون جامع نمایان‌گر یکسری مشکلات لجستیکی معین می‌شود. حتی در مورد برنامه‌های کوچک، تعداد مسیرهای منطقی احتمالی می‌توانند بسیار زیاد باشند. مثلاً یک برنامه ۱۰۰ خطی به زبان C را در نظر بگیرید. بعد از یک شرح مقدماتی در مورد داده‌ها، برنامه دارای دو حلقه است که هر کدام ۱ تا ۲۰ بار اجرا می‌شوند که به شرایط مشخص شده در داده‌های ورودی بستگی دارند. در داخل حلقه داخلی، چهار ساختمان لازمند. تقریباً 10^{14} مسیر احتمالی وجود دارد که ممکن است در این برنامه اجرا شوند!

برای این‌که از این عدد ایده‌ای داشته باشید، فرض می‌کنیم که یک پردازشگر آزمونی جادویی وجود دارد ("جادویی" برای آن‌که چنین پردازشگری نیست) که برای آزمون جامع و کامل ارائه شده است. پردازشگر می‌تواند یک مورد آزمونی ارائه دهد، آن را اجرا نموده و نتایجش را در یک میلینیوم ثانیه ارزیابی کند. پردازشگر اگر ۲۴ ساعته و ۳۶۵ روز را کار کند، باید برای آزمون برنامه ۳۱۷۰ سال کار کند. این امر بدون تردید، باعث بی‌نظمی زیادی در جداول زمانی توسعه نرم‌افزار می‌شود. آزمون جامع برای سیستم‌های بزرگ نرم‌افزاری غیر ممکن است.

آزمون های جعبه – سفید تنها پس از وجود طراحی تفصیلی (یا برنامه نویسی) می توانند طراحی شوند. از آنرو که جزئیات منطقی برنامه باید در اختیار باشد

1.black-box

2.White-box

آزمون جعبه سفید را نباید به‌عنوان غیر عملی بودن کنار گذاشت. تعداد محدودی از مسیرهای منطقی مهم را می‌توان انتخاب و آزمون نمود. ساختارهای مهم داده‌ای را می‌توان از نظر اعتبار بررسی نمود. می‌توان روش‌های هر دو شیوه جعبه سیاه و سفید را با هم ترکیب کرد تا یک روشی به‌وجود آورد که رابط نرم‌افزاری را ارزیابی نموده و به‌صورت گزینشی مطمئن سازد که اعمال درونی نرم‌افزار درست می‌باشند.

۱۷-۳ آزمون جعبه سفید

ممکن نیست که تمام مسیرهای یک برنامه را مورد آزمون قرار داد. بدان دلیل که تعداد مسیرها می تواند بسیار زیاد باشد.

این آزمون که گاهی به آن آزمون جعبه شیشه‌ای[1] نیز می‌گویند، یک روش طراحی مورد آزمونی است که از ساختار کنترل طراحی رویه برای بدست آوردن موارد آزمون استفاده می‌کند. با استفاده از روش‌های آزمون جعبه سفید مهندس نرم‌افزار می‌تواند موارد آزمونی را بدست آورد که ۱) تضمین کنند که همه مسیرهای مستقل داخل یک پیمانه حداقل یکبار به‌کار گرفته شده‌اند. ۲) همه تصمیمات منطقی را در مورد طرفین درست و غلط آنها اجرا کند. ۳) همه لوپها (حلقه‌ها) را در سر حدات آنها و در داخل سر حدات عملیاتی آنها اجرا کند و ۴) تمام ساختارهای اطلاعاتی داخلی را برای تضمین اعتبارشان اجرا سازد.

ممکن است یک سؤال منطقی در این مرحله حادث شود: «چرا زمان و انرژی خود را صرف نگرانی در مورد جزییات منطقی کنیم، وقتی که ممکن است به‌نحو بهتری تلاش خود را بسط دهیم تا مطمئن شویم که نیازمندیهای برنامه مهیا شده‌اند؟» یا اگر بخواهیم طور دیگری آن را بیان کنیم، چرا تمام انرژی خود را صرف آزمونهای جعبه سیاه نکنیم؟ پاسخ در ماهیت اشکالات نرم‌افزاری نهفته است: [JON81][1]

نقل قول

خطاهای پنهان (bug ها) در زوایا و گوشه های سیستم پنهان می شوند و در مرزها اجتماع می یابند.

بوریس بیزر

- **خطاهای منطقی و فرضیات نادرست با امکان این که مسیر برنامه‌ای اجرا شود رابطه معکوس دارند.** وقتی خطاهایی در کارمان به‌وجود می‌آیند که ما کارکرد، شرایط یا کنترلی را طراحی و پیاده‌سازی کنیم که خارج از روند اصلی هستند. هر روزه از فرآیند انتظار می‌رود که به‌خوبی درک شود در حالی‌که فرآیند یک مورد ویژه در دام خطاها می‌افتد.
- **اغلب معتقدیم که یک مسیر منطقی احتمالاً وقتی اجرا نمی‌گردد که بر پایه معمول اجرا گردد.** جریان منطقی یک برنامه گاهی غیر حدسی است، یعنی که فرضیات غیر هوشیارانه ما در مورد جریان کنترل و داده‌ها ممکن است ما را به‌سوی ایجاد خطاهای طراحی هدایت کند که تنها یکبار در هنگام شروع آزمون مشخص می‌شود.
- **خطاهای مربوط به تایپ برنامه، به‌صورت تصادفی هستند.** وقتی برنامه‌ای به کد منبع زبان برنامه‌نویسی ترجمه می‌شود، احتمال دارد که چند خطای تایپی رخ دهد. بسیاری از این خطاها به‌وسیله

1.glass-box

مکانیزم‌های بازبینی تایپ و نحو پیدا می‌شوند، اما بقیه نامشخص می‌مانند تا وقتی آزمون شروع می‌شود، احتمال دارد که یک تیپو روی مسیر منطقی ناشناخته‌ای به‌عنوان مسیر اصلی جریان، وجود داشته باشد.

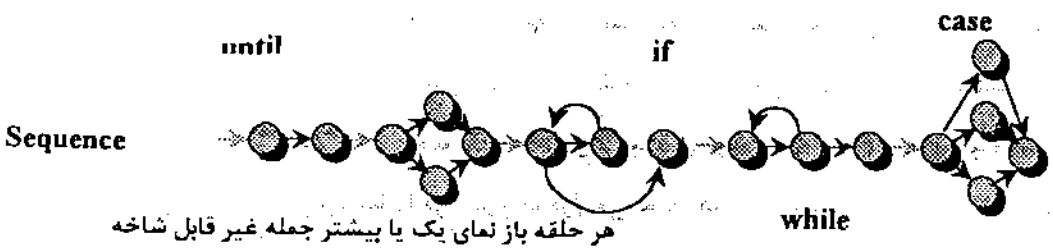
۱۷-۴ آزمون مسیر پایه

آزمون مسیر پایه[2] یک تکنیک آزمون جعبه سفید است که اولین بار توسط تام مک کیب [MCC76][3] به‌کار گرفته شد. این روش، طراح مورد آزمونی را قادر می‌سازد تا یک ارزیابی پیچیده منطقی را از طرح رویه داشته و از این ارزیابی به‌عنوان راهنمایی برای تعریف مجموعه مقدماتی مسیرهای اجرایی استفاده کند. موارد آزمونی بدست آمده برای به‌کارگیری در مجموعه مقدماتی از نظر اجرای هر جمله یا دستوالعمل برنامه، حداقل یکبار در طول آزمون تضمین شده‌اند.

۱۷-۴-۱علائم گراف جریان

قبل از معرفی روش مسیر مقدماتی یا پایه، از نشانه‌گذاری ساده‌تری برای نمایش جریان کنترل استفاده می‌شد که گراف جریان[4] (یا گراف برنامه) نام داشت که باید معرفی شود.[5] گراف جریان با استفاده از نشانه‌گذاری آمده در شکل ۱۷-۱، جریان منطقی کنترل را مشخص می‌کند. هر ساختمان (سازه) ساختیافته شده (فصل ۱۶) دارای یک نشانه بخصوص در نمودار جریان است.

ساختمانهای ساخت یافته در گراف روند :



شکل ۱۷ـ۱ علائم گراف روند

1.Jones,T.C.

2.Basis path testing

3.McCabe,T.

4.flow graph

۵. در عمل، شیوه مبتنی بر مسیر، بدون استفاده از گرافهای جریان نیز قابل استفاده است. با این وجود، آنها ابزاری مفید برای فهم کنترل جریان و درک رهیافت فراهم می سازند.

توضیح
هنگامیکه ساختار کنترل منطقی یک پیمانه پیچیده می نماید، اقدام به رسم یک گراف جریان نمایید.

به‌منظور تشریح کاربرد گراف جریان، طراحی رویه‌ای را در نظر می‌گیریم که در شکل (۱۷-۲ الف) آمده است. در این‌جا، از یک فلوچارت برای مشخص نمودن ساختمان کنترلی برنامه استفاده شده است. شکل (۱۷-۲ ب) فلوچارت را در گراف جریان مربوطه طرح می‌کند. (با فرض این‌که هیچ‌گونه شرایط ترکیبی برای خانه‌ها تصمیم‌گیری فلوچارت گنجانده نشده باشد.) با رجوع به شکل (۱۷-۲ب)، هر دایره که یک گره گراف جریان[1] نامیده می‌شود، نمایان‌گر یک یا چند جمله رویه‌ای است. یک رشته جعبه‌های فرآیند و یک لوزی مربوط به تصمیم‌گیری، می‌توان در هر گره طراحی کرد. پیکان‌های روی نمودار جریان به نام لبه‌ها[2] یا رابط‌ها[3] نمایان‌گر جریان کنترل بوده و با پیکان‌های فلوچارت قابل مقایسه‌اند. هر لبه باید در هر گره خاتمه یابد، حتی اگر گره نمایان‌گر هیچ جمله رویه‌ای نباشد. (مثلاً نشانه ساختار اگر - پس - و دیگر if then-else را ببینید.) مناطق محدود شده توسط لبه‌ها و گره‌ها را به نام محدوده[4] می‌نامند. در هنگام شمارش این مناطق حوزه خارج از نمودار را نیز در نظر گرفته و آن را به‌عنوان یک منطقه یا Region می‌نامیم.[5]

وقتی در طراحی رویه‌ای با شرایط پیچیده‌ای برخورد می‌کنیم، ایجاد گراف جریان کمی پیچیده‌تر می‌شود. وقتی یک وضعیت ترکیبی رخ می‌دهد که یک یا چند اپراتور دوارزشی (مثل NOR, NAND, AND, OR منطقی) در وضعیت شرطی وجود داشته باشند. با توجه به شکل ۱۷-۳، بخش PDL به نمودار وضعیت IF a OR b به‌وجود می‌آید. هر گره‌ای که دارای یک شرط است، یک گره گزاره‌ای[6] نامیده شده و دو یا سه لبه دارد که از آن بیرون زده‌اند.

۱۷-۴-۲ پیچیدگی چرخشی (سیکلوماتیک)[7]

این یک متریک نرم‌افزاری است که اندازه کمّی پیچیدگی منطقی یک برنامه را مهیا می‌سازد. وقتی از آن در بستر روش آزمون مسیر مقدماتی استفاده می‌شود، ارزش محاسبه شده برای پیچیدگی سیکلوماتیک مشخص‌کننده تعدادی از مسیرهای مستقل در مجموعه مقدماتی برنامه بوده و نیز حد بالائی برای تعدادی از آزمونها را در اختیار ما قرار می دهند که باید به‌منظور اطمینان از اجرای همه جملات مورد آزمون، حداقل برای یک‌بار انجام دهند.

1.flow graph node

2.edges

3.links

4.regions

۵. توضیح مفصل بیشتر از گرافها و کاربرد آنها در آزمون، در بخش ۱۷-۶-۱ آورده شده است.

6.predicate node

7.Cyclomatic complexity

یک مسیر مستقل[1] هر مسیری در برنامه است که حداقل یک مجموعه جدیدی از حالات پردازشی یا وضعیت جدیدی را معرفی می‌کند. وقتی مسیر مستقل از نظر شرایط گراف جریان بیان می‌شود، باید در طول حداقل یک لبه‌ای حرکت کند که قبل از تعریف مسیر، پیمایش نشده باشند. مثلاً، مجموعه‌ای از مسیرهای مستقل برای نمودار جریان در شکل (۱۷-۲ب) آمده‌اند:

مسیر۱ : ۱-۱۱

مسیر۲ : ۱-۲-۳-۴-۵-۱۰-۱-۱۱

مسیر ۳: ۱-۲-۳-۶-۸-۹-۱۰-۱-۱۱

مسیر ۴: ۱-۲-۳-۶-۷-۹-۱۰-۱-۱۱

توجه داشته باشید که هر مسیر جدید یک لبه تازه را معرفی می‌کند. مسیر

۱-۲-۳-۴-۵-۱۰-۱-۲-۳-۶-۸-۹-۱۰-۱۱

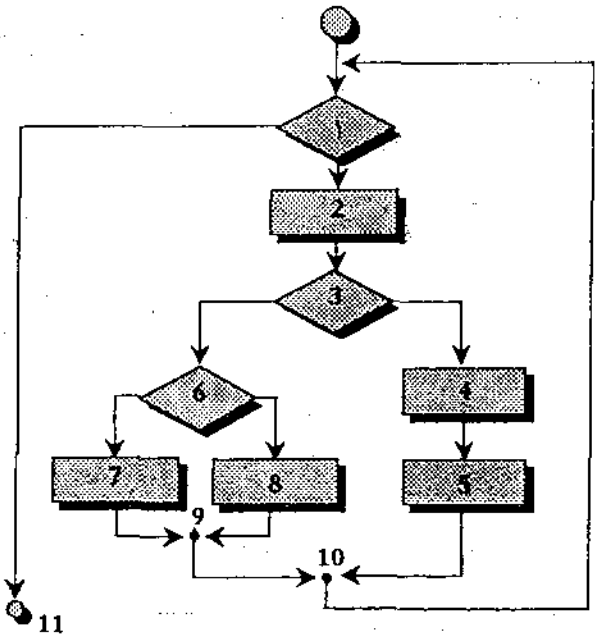
به عنوان یک مسیر مستقل در نظر گرفته نمی‌شود، زیرا صرفاً ترکیبی از مسیرهای مشخص شده کنونی بوده و هیچ لبه جدیدی را پیمایش نمی‌کند.

مسیرهای ۱، ۲، ۳ و ۴ که در بالا تعریف شده‌اند، مجموعه پایه[2] را برای نمودار جریان در شکل (۱۷-۲ب) تشکیل می‌دهند. یعنی اگر بتوان آزمون‌هایی را طراحی کرد که اجرای این مسیرها را به اجبار انجام دهند، هر حالت از برنامه به‌طور تضمینی، حداقل یکبار و تمام شرطی از نظر درست و غلط بودنش اجرا می‌گردد. باید توجه داشت که مجموعه پایه، منحصر به فرد نیست. در حقیقت، چندین مجموعه مختلف مقدماتی را می‌توان برای یک طراحی رویه‌ای فرضی در نظر گرفت.

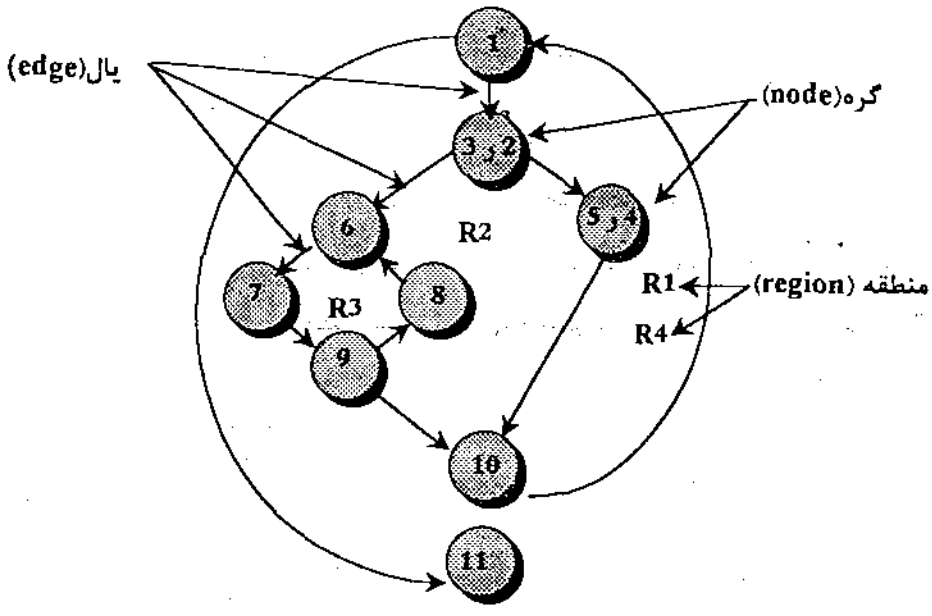
پیچیدگی چرخشی (سیکلوماتیک)، متریک مناسبی برای پیش بینی پیمانه هایی است که مستعد خطا می باشند. می توان آن را به همان خوبی برای آزمون طراحی به کار برد که برای طرح ریزی آزمون مورد استفاده قرار می گیرد.

1.independent path

2.basis set



(الف)



(ب)

شکل ۱۷ـ۲ روند نما (الف) وگراف روند (ب)

چگونه می‌فهمیم در جستجوی چند مسیر باشیم؟ محاسبه پیچیدگی سیکلوماتیک پاسخ این سؤال را مهیا می‌کند.

پیچیدگی سیکلوماتیک پایه و اساس در تئوری گراف دارد و یک متریک نرم‌افزاری بسیار مفید در اختیار ما می‌گذارد. این پیچیدگی به یکی از سه شکل زیر محاسبه می‌شود:

پرسش

پیچیدگی چرخشی (سیکلوماتیک) چگونه محاسبه می شود؟

۱- تعداد مناطق نمودار جریان که با پیچیدگی سیکلوماتیک ارتباط دارند.

۲- پیچیدگی سیکلوماتیک، V(G) برای گراف جریان G به‌صورت زیر تعریف شده:

$$V(G) = E - N + 2$$

در آن E تعداد لبه‌های نمودار جریان و N تعداد گره‌ها می‌باشد.

۳- V(G) برای نمودار جریان (G) به‌صورت زیر تعریف می‌شود:

$$V(G) = P + 1$$

در آن P تعداد گره‌های مورد اسناد در نمودار جریان G است.

اگر یکبار دیگر به نمودار جریان در شکل (۱۷-۲ب) اشاره کنیم، می‌توان پیچیدگی سیکلوماتیک را با استفاده از الگوریتم‌های اشاره شده فوق محاسبه نمود:

۱- نمودار جریان دارای چهار منطقه است.

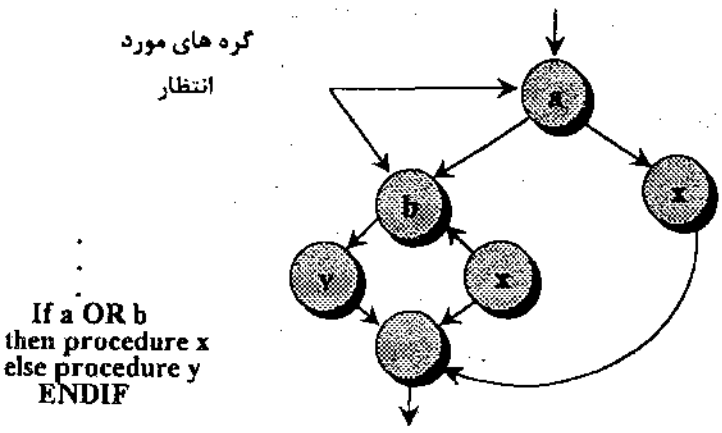
۲- ۴=۲+گره ۹- لبه ۱۱= V(G)

۳- ۴=۱+گره اسنادی ۳=V(G)

بنابراین پیچیدگی سیکلوماتیک در نمودار جریان شکل (۱۷-۲ب) می‌شود ۴.

پیچیدگی چرخشی، حد بالایی برای برخی موارد آزمون فراهم می سازد. هنگامی که هدف، اطمینان از اجرای جملات یک جزء (حداقل برای یک بار) باشد.

مهم‌تر این‌که، مقدار V(G) سرحد بالایی را برای تعدادی از مسیرهای مستقل در اختیار ما قرار می‌دهد که مجموعه مقدماتی را تشکیل می‌دهند و به‌طور تلویحی سرحد بالایی در مورد تعداد آزمونهایی که باید طراحی و اجرا شوند تا در برگرفتن همه جملات برنامه را تضمین کنند نیز به ما می‌دهد.



شکل ۱۷ـ۳ منطق مرکب

نقل قول

انسان ممکن الخطاست. برای کشف یک خطا باید رفتاری خداگون داشته باشید.

روبرت دان

۱۷-۴-۳ دستیابی به موارد آزمون

روش آزمون مسیر ابتدایی را می‌توان در طراحی رویه یا کد منبع به‌کار گرفت. در این بخش، ما آزمون مسیر ابتدایی را به‌صورت یک‌سری مراحل ارائه می‌کنیم. شیوه یا روال معدل[1] (میانگین) که در PDL در شکل ۱۷-۴ آمده به‌عنوان نمونه‌ای برای تشریح هر مرحله از روش طراحی مورد آزمون، استفاده می‌شود. توجه داشته باشید که میانگین با این‌که یک الگوریتم بسیار ساده است اما حاوی شرایط و حلقه‌های پیچیده‌ای است. مراحل زیر را می‌توان برای نتیجه‌گیری مجموعه مقدماتی به‌کار گرفت:

۱- استفاده از طرح یا کد به‌عنوان پایه و کشیدن نمودار مربوطه جریان.

نمودار جریان با استفاده از نمادها و قواعد ساختاری ارائه شده در بخش ۱۶-۴-۱ ایجاد می‌شود. با اشاره به PDL برای میانگین در شکل ۱۷-۴، یک نمودار جریان با عددگذاری حالت PDL ایجاد می‌شود که میانگین رویه خواهد بود.

این روال یا رویه به‌طور میانگین ۱۰۰ یا تعداد کمتری را که بین مقادیر سرحد قرار گرفته‌اند، محاسبه می‌کند. هم‌چنین، مجموع کل و ارزش رقمی کل را نیز می‌دهد. گراف نگاشت آن به گره‌های گراف جریان دیده می‌شود. گراف جریان مربوطه در شکل ۱۷-۵ است.

۲- تعیین پیچیدگی سیکلوماتیک (Cyclomatic Complexity) نمودار جریان بدست آمده.

پیچیدگی سیکلوماتیک V(G) با به‌کارگیری الگوریتم‌های آمده در بخش ۱۷-۵-۲ تعیین می‌شود. باید توجه داشت که V(G) را می‌توان بدون بسط نمودار جریان با شمارش تمام حالات ترکیبی در PDL و افزودن 1 تعیین نمود.

۲- با توجه به شکل ۱۷-۵ :

V(G) = ۶ منطقه

V(G) = ۱۷ لبه - ۱۳ گره +۲=۶

V(G) = ۵ گره اسنادی + ۱ =۶

```
PROCEDURE average;

. This procedure computes the average of 100 or fewer
  numbers that lie between bounding values: it also computes the
  sum and the total number valid.

INTERFACE RETURNS average. total.Input. total.valid:
INTERFACE ACCEPTS value, minimum. maximum:
```

1. average

```
    TYPE  value[I:100] IS SCALAR ARRAY;
TYPE  average, total.Input. total. valid;
    minimum. maximum. Bum IS SCALAR;
TYPE i IS INTEGER;

    { i=1;
1   { total.Input = total.valid 2 0;
      sum=0;
      DO WHILE value[i] < > -999 AND total.Inp 3 < 100
4         Increment total.Input by 1;
          IF value[i] > == minimum AND value[i] < = 6 aximum.
5     7 { THEN Increment total.valid by 1;
              sum = s sum + value[i]
          ELSE sklp
8     {   ENDIF
              Increment i by 1;

  9 ENDDO
    IF total.valid >10
12  11  THEN average = sum / total.valid;
    ELSE average = -999;
13  ENDIF
END average
```

۳- تعیین مجموعه اولیه مسیرهای مستقل خطی.

مقدار V(G) با تعداد مسیرهای مستقل خطی در ساختار برنامه کنترلی بدست می‌آید. در مورد میانگین رویه، انتظار داریم، شش مسیر مشخص شود:

مسیر۱: ۱۳-۱۱-۱۰-۲-۱

مسیر۲: ۱۳-۱۲-۱۰-۲-۱

مسیر۳: ۱۳-۱۱-۱۰-۳-۲-۱

مسیر۴: .....-۲-۹-۸-۵-۴-۳-۲-۱

مسیر ۵ : .....-۲-۹-۸-۶-۵-۴-۳-۲-۱

مسیر۶ : .....-۲-۹-۸-۷-۶-۵-۴-۳-۲-۱

قسمت‌هایی که با نقطه‌چین به‌دنبال شماره‌های ۴، ۵ و ۶ آمده حذف به قرینه بوده و نشان‌گر این است که هر مسیری در باقی ساختمان کنترل پذیرفته است. اغلب شناسایی گره‌های اسنادی به‌عنوان یک کمک در رسیدن به موارد آزمون، ارزشمند است. در این مورد گره‌های ۲، ۳، ۵، ۶ و ۱۰ گره‌های اسناد هستند.

۴- آماده‌سازی مواردی که اجرای هر مسیر را در مجموعه مقدماتی، اجباری می‌سازند.

دادهها باید انتخابی باشند به‌طوری‌که شرایط در گره‌های اسنادی به درستی تنظیم شده و هر مسیر آزمون می‌شود. موارد آزمونی که مجموعه اولیه را برقرار می‌سازند در بالا توصیف شده‌اند.

نقل قول

(مهندسین نرم افزار) در تعیین تعداد آزمون های مورد نیاز برای یک برنامه ساده، به طور قابل ملاحظه ای برآورد اندکی دارند. مارتین اولد و چالز الوین

ـ مورد آزمون مسیر ۱ :

مقدار (K) = ورودی معتبر، که در آن K<i برای ۱۰۰≤ i ≤ ۲

مقدار (i) برابر ۹۹۹- که ۱۰۰≤ i ≤ ۲

نتایج منتظره: میانگین درست بر اساس مقدار K و مقادیر کل درست.

توجه: مسیر ۱ را نمی‌توان به تنهایی آزمود، اما باید به‌عنوان بخشی از مسیرهای ۴، ۵ و ۶ مورد آزمون واقع شود.

ـ مورد آزمون مسیر ۲:

مقدار (i) برابر ۹۹۹-

نتایج موردنظر: میانگین ۹۹۹ـ ، سایر مقادیر کل در مقادیر اولیه.

ـ مورد آزمون مسیر ۳:

ـ تلاش به‌منظور پردازش مقدار ۱۰۱ یا بیشتر

ابتدا ۱۰۰ رقم اول باید با ارزش باشند.

نتایج موردنظر: مثل آزمون مورد ۱

ـ مورد آزمون مسیر ۴:

مقدار (i) ورودی معتبر برای ۱۰۰ >i مقدار (K) کمتر از کمینه برای K<i

نتایج موردنظر: میانگین درست بر اساس مقادیر K و مجموع کل‌های مناسب

ـ مورد آزمون مسیر ۵:

مقدار (i) ورودی معتبر برای ۱۰۰ >i مقدار (K) بزرگتر از بیشینه برای K<=i

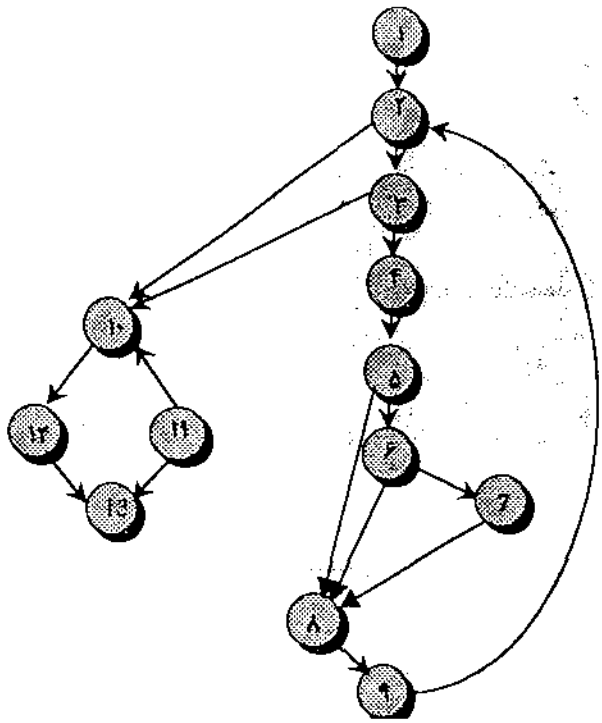
نتایج موردنظر: میانگین درست بر اساس مقادیر N و مجموع کل‌های مناسب

ـ مورد آزمون مسیر ۶ :

مقدار (i) ورودی معتبر برای ۱۰۰ >i

نتایج موردنظر: میانگین درست بر اساس مقادیر N و مجموع کل‌های مناسب

هر مورد آزمون اجرا شده و با نتایج پیش‌بینی شده مقایسه شد. زمانی‌که همه موارد تکمیل گردید، فرد آزمون‌گر می‌تواند مطمئن باشد که همه جملات برنامه حداقل یکبار اجرا شده است.



شکل ۱۷ـ۵ گراف روند برای رویه avreage (معدل گیری)

نکته مهم این است که بعضی از مسیرهای مستقل (مثل مسیر ۱ در مثال ما) را نمی‌توان به‌تنهایی آزمود. یعنی ترکیب داده‌های لازم برای پیمودن مسیر را با جریان و روال عادی برنامه نمی‌توان بدست آورد. در چنین مواردی، این مسیرها به‌عنوان بخشی از آزمون مسیر دیگر آزمون می‌شوند.

۱۷-۴-۴ ماتریس های گراف

پرسش

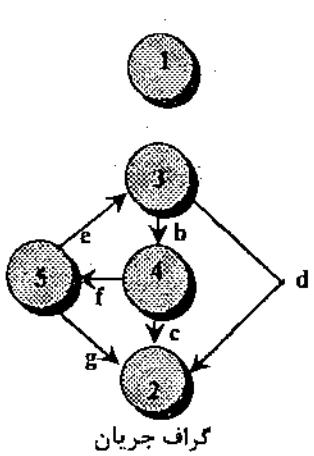
یک ماتریس گراف چیست و چگونه می توان آن را در آزمون به کار برد؟

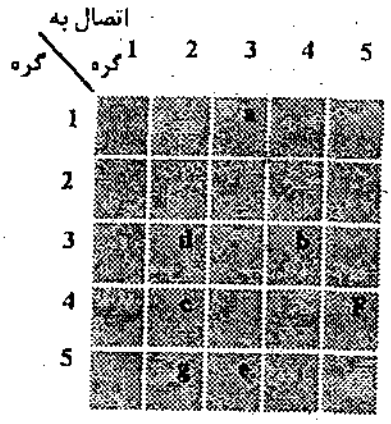
روال بدست آوردن نمودار جریان و حتی تعیین مجموعه‌ای از مسیرهای مقدماتی، قابل مکانیزه‌سازی است. برای تولید یک ابزار نرم‌افزاری که در آزمون مسیر اولیه ما را یاری کند، یک ساختار داده‌ای به‌نام ماتریس گراف[1] می‌تواند کاملاً مفید باشد.

ماتریس گراف یک ماتریس مربعی است که اندازه‌اش (یعنی تعداد ردیف‌ها و ستون‌ها) برابر تعداد گره‌های روی نمودار جریان است. هر ستون و ردیف با گره‌ای مشخص مرتبط است و مقادیر ورودی ماتریس با ارتباطات بین گره‌ها مرتبطند. مثال سادهٔ ای از نمودار جریان و ماتریس گراف مربوطه آن در شکل ۱۷-۶ آمده است.[BEI90] [2]

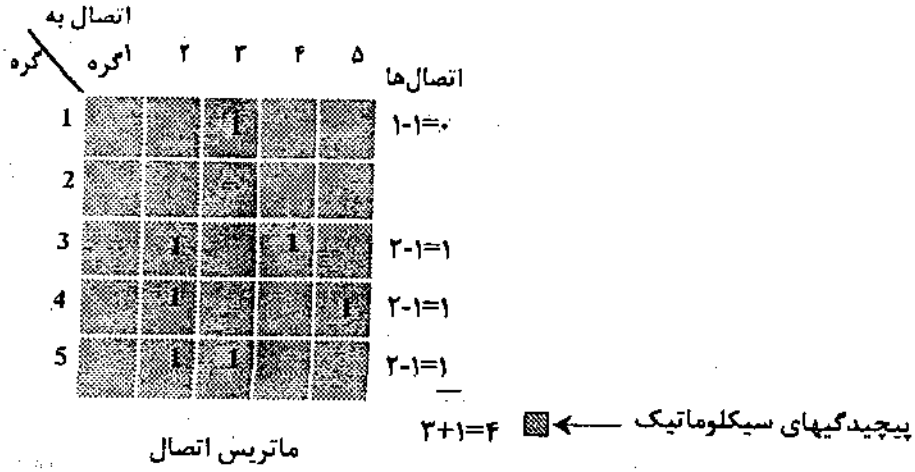
1.graph matrix

2.Beizer,B.





شکل ۱۷-۶ ماتریس گراف



شکل ۱۷-۷ ماتریس اتصال

با رجوع به شکل، هر گره در نمودار جریان به‌وسیله ارقام شناسایی می‌شود، در حالی‌که هر لبه به‌وسیله حروف مشخص می‌شود. یک مدخل حرفی در ماتریس مرتبط با اتصال میان دو گره به‌وجود می‌آید. مثلاً گره ۳ به‌وسیله لبه b به گره ۴ متصل است.

تا این نقطه، ماتریس گراف چیزی بیشتر از یک بازنمایی جدولی از گراف جریان نیست. با افزودن یک وزن اتصال به هر مدخل ماتریس، ماتریس گراف یک ابزار دقیق و قدرتمند برای ارزیابی ساختار کنترلی برنامه در طول آزمون می‌شود. در ساده‌ترین حالت وزن اتصال ۱ (یک اتصال وجود دارد) یا صفر است (که ارتباط وجود ندارد). اما این وزن اتصال‌ها، می‌توانند دارای خصوصیات جالب‌تری باشند:

- احتمال این‌که یک Link (لبه) اجرا شود.
- زمان پردازش که در طول پیمایش یک اتصال سپری می‌شود.
- حافظه لازم در طول پیمایش یک اتصال
- منابع لازم در طول پیمایش یک اتصال

به‌منظور تشریح گفته‌های فوق از ساده‌ترین ارزیابی برای اشاره به ارتباط‌ها (صفر یا ۱) استفاده می‌کنیم.

ماتریس گراف شکل ۱۷-۶ به‌صورتی‌که در شکل ۱۷-۷ آمده، دوباره کشیده شده است. به‌جای هر حرف یک عدد ۱ گذاشته شده که نشان‌دهنده وجود ارتباط است. (به‌خاطر وضوح تصویر صفرها گذاشته نشده‌اند).

ماتریس گرافی که به این شکل نشان داده می‌شود، ماتریس اتصال خوانده می‌شود.

با رجوع به شکل ۱۷-۷، هر ردیف با دو یا چند مدخل نمایان‌گر یک گره استادی است. بنابراین، با انجام عملیات ریاضی نشان داده شده در سمت راست ماتریس، روش دیگری برای تعیین پیچیدگی سیکلوماتیک در اختیار ما قرار می‌گیرد.

بیزر یک روش کامل از الگوریتم‌های اضافی ریاضی را در اختیار ما می‌گذارد که می‌توان آن را در ماتریس‌های گراف به‌کار گرفت. با استفاده از این تکنیک‌ها، می‌توان تحلیل لازم برای طراحی موارد آزمون را، به‌طور نسبی یا به طور کامل، خودکار طراحی کرد.

۱۷-۵ آزمون ساختار کنترل

تکنیک آزمون مسیر مقدماتی که در بخش ۱۷-۴ توصیف شده، یکی از چند تکنیک آزمون کنترل ساختار است. گرچه این آزمون ساده و بسیار مؤثر است، اما این به‌خودی خود کافی نیست. در این بخش انواع دیگری از آزمون ساختار مورد بحث قرار می‌گیرند. این آزمون گسترش‌یافته، کیفیت آزمون جعبه سفید را بهبود بخشیده و آن را به‌طور کامل در برمی‌گیرد.

خطا ها بیشتر در همسایگی شرایط منطقی دیده می شوند تا سلسله مراتبی از جملات پردازشی.

۱۷-۵-۱ آزمون شرط[1]

آزمون وضعیت[2] یک روش طراحی مورد آزمون است که شرایط منطقی موجود در پیمانه برنامه را می آزماید. یک وضعیت ساده، یک متغیر بولین یا عبارت رابطه‌ای است که احتمالاً با یک اپراتور NOT (" ¬ ") همراه است. عبارت ربطی شکل زیر را به خود می‌گیرد:

$$E_1\text{<Relational-Operator>}E_2$$

۱. بخشهای ۱۷-۵-۱ و ۱۷-۵-۱ مطابق مطالب پروفسور کی.سی.تای. [TAI89] آورده شده است.

2.Condition testing

که در آن E_1 و E_2 عبارت جبری و قسمت < عملگر رابطه‌ای> یکی از موارد زیر است: "<" ، "≤" ، "≠" (غیرمساوی)، ">" یا "≥" وضعیت مرکب[1] متشکل از دو یا چند وضعیت ساده، اپراتورهای بولین و پرانتزهاست. فرض می‌کنیم که اپراتورهای بولین که در وضعیت مرکب مجازندشامل ("|") OR، (&) AND و ("¬") NOT باشد. شرط بدون عبارات ربطی را عبارت بولین[2] می‌نامند.

بنابراین، انواع احتمالی عناصر یک شرط شامل موارد زیر هستند: اپراتور بولین، متغیر بولین، یک جفت پرانتز بولین (که یک شرط ساده یا مرکب را در برمی‌گیرد)، یک اپراتور رابطه‌ای یا یک عبارت جبری. اگر شرطی درست نباشد، حداقل یک جزء شرط درست نیست. بنابراین انواع خطاهای یک شرط شامل موارد زیر هستند:

- خطای عملگر بولین (نادرست/ از قلم افتاده/ اپراتورهای اضافی بولین)
- خطای متغیر بولین
- خطای پرانتز بولین
- خطای اپراتور رابطه‌ای
- خطای عبارت محاسباتی

روش آزمون شرطی روی آزمون هر شرط در برنامه متمرکز می‌شود. معمولاً راهبردهای آزمون شرط (که بعداً در این بخش مورد بحث قرار می‌گیرد) دارای دو مزیت هستند. اولی، اندازه‌گیری شمول و پوشش آزمون هر شرط ساده است. دوم، پوشش دادن شروط برنامه، راهنمایی برای ایجاد آزمونهای اضافی برنامه مهیا می‌کند.

هدف از آزمون شرط این است که نه تنها خطاهای شرایط موجود برنامه را تشخیص دهد، بلکه دیگر خطاهای برنامه را نیز مشخص کند. اگر یک مجموعه آزمون برای برنامه "P" برای تشخیص خطاها در شرایط موجود در P مؤثر باشد، این احتمال وجود دارد که این مجموعه آزمون برای تشخیص دیگر خطاهای p نیز مؤثر باشد. علاوه بر این، اگر این راهبرد آزمون برای تشخیص خطاها در یک شرط مؤثر باشد پس برای شناسایی خطاهای برنامه نیز مؤثر است.

چند شیوه آزمون شرط پیشنهاد می‌شود. آزمون شاخه[3] احتمالاً ساده‌ترین شیوه است. در مورد وضعیت مرکب C، شاخه‌های درست و غلط C و هر شرط ساده در C باید حداقل یکبار اجرا شوند. [MYE79][4]

1.compound condition

2.Boolean expression

3.Branch testing

4.Myer, G.

آزمون میزان[1] نیازمند سه یا چهار آزمون است که باید برای عبارت ربطی بدست آیند. برای عبارت ربطی به شکل :[WHI80] [2]

$$E_1<\text{Relational-Operator}>E_2$$

سه مجموعه .لازمندتا ارزش E_1 را بیشتر، مساوی یاکمتر از E_2 کنند. اگرعبارت <Relational-Operator>نادرست باشد و E_1 و E_2 درست، پس این سه آزمون، تشخیص خطای اپراتور رابطه‌ای را تضمین می‌کنند. برای شناسایی خطاهای E_1 و E_2 آزمونی که ارزش E_1 را بیشتر یا کمتر از E_2 می‌کنند. [HOW82][3] باید تفاوتی میان این دومقدار با حداقل ممکن ایجاد کنند.

در مورد عبارت بولین باn متغیر دوتایی، همه آزمونهای احتمالی ۲n لازمند. (n>0) این شیوه می‌تواند خطاهای اپراتور، متغیر و پرانتز بولین را شناسایی کند اما عملاً تنها وقتی n مقدار کوچکی دارد، عمل می‌کند.

آزمون‌های حساس به خطا در مورد عبارات بولین را نیز می‌توان بدست آورد. در مورد یک عبارت بولین (یک عبارت بولین که در آن هر متغیر بولین تنها یک بار واقع می‌شود) همراه با n متغیر بولین (n>0)، می‌توانیم به‌راحتی مجموعه آزمونی با آزمون‌های کمتر از ۲n ایجاد کنیم که این مجموعه شناسایی خطاهای چندگانه در اپراتور بولین را تضمین نموده و برای مشخص کردن دیگر خطاها نیز مؤثرند.[FOS84,TAI87] [4]

تای [TAI89][5] یک آزمون شرط را ارائه می‌دهد که فنون تشریح شده فوق را به‌کار می‌بندد. در آزمون به اصطلاح BRO (عملگر رابطه‌ای و شاخه)، این تکنیک شناسایی خطاهای شاخه و اپراتور ربطی را در یک وضعیت تضمین می‌کند به‌شرط این‌که تمام متغیرهای بولین و عملگر رابطه‌ای در این موقعیت تنها یکبار واقع شده و متغیرهای مشترکی نداشته باشند.

راهبرد BRO از محدودیت‌های شرطی برای وضعیت C استفاده می‌کند. محدودیت شرطی برای C با n تا شرط ساده به‌صورت $(D_1, D_2, D_3, \ldots, D_n)$ تعریف شده که در آن D_i $(0<i\leq n)$ نشانه‌ای است که محدودیتی را در مورد نتیجه شرط ساده iأم در وضعیت C، مشخص می‌کند. محدودیت شرطی D در مورد وضعیت C تحت موقعیت اجرای C است اگر در طول این اجرای C نتیجه هر شرط ساده در C محدودیت مربوطه را در D برقرار سازد.

حتی هنگامی که در قبال آزمون شرایط به تصمیم قطعی رسیده اید، لازم است زمانی را به هر شرط اختصاص دهید تا از پوشش خطاها اطمینان حاصل شود. از آنرو که اینجا محل امنی برای خطاهای پنهان (باگ ها) می باشد.

1.Domain testing

2.White , L.J. and E.I.Cohen

3.Howden,W.E.

4.Foster,K.A. / Tai,K.C.

5.Tai,K.C.

در مورد متغیر بولین B، محدودیتی را در مورد نتیجه B مشخص می‌کنیم که بیان می‌دارد B باید یا درست یا غلط باشد. از این‌رو، در مورد عبارت ربطی، علایم <, = , > برای مشخص کردن محدودیت‌های نتیجه عبارت استفاده می‌شوند.

به‌طور مثال، وضعیت زیر را در نظر بگیرید:

$$C_1: B_1 \& B_2$$

که در آن B_1 و B_2 متغیرهای بولین هستند. محدودیت شرطی در مورد C_1 به‌صورت (D_1, D_2) است که در آن هر یک از موارد D_2, D_1 درست (T) یا غلط (F) هستند. ارزش درست یا غلط بودن برای C_1 یک محدودیت شرطی است و به‌وسیله آزمون درست بودن ارزش B_1 یا غلط بودن ارزش B_2 حاصل می‌گردد. شیوه آزمون BRO مستلزم مجموعه محدودیت {(T,T), (F,T), (T,F)} است که با اجرای C_1 بدست می‌آید. اگر C_1 به‌خاطر یک یا چند خطای اپراتور بولین نادرست باشد، حداقل یک مجموعه محدود باعث شکست C_1 می‌شود.

در دومین مثال وضعیتی به‌صورت زیر داریم:

$$C_1: B_1 \& (E_3 = E_4)$$

که در آن B_1 یک عبارت بولین و E_3 و E_4 عبارات جبری‌اند. محدودیت شرطی برای C_2 به‌صورت (D_1, D_2) است که در آن هر D_1 درست یا غلط و D_2 > ، = ، < است. از آن‌جا که C_2 مثل C_1 است. به‌جز این‌که دومین شرط ساده در C_2 یک عبارت ربطی است، می‌توانیم مجموعه محدودی برای C_2 با تغییر مجموعه {(T,T), (F,T), (T,F)} که برای C_1 تعریف شده، بسازیم.

توجه کنید که "T" برای $(E_3=E_4)$ نشان‌گر "=" و "F" برای $(E_3=E_4)$ نشان‌گر > یا < است. با جایگزین کردن (T, =) و (F, =) به‌جای (T,T) و (F, T) و قرار دادن (T, <) و (T,>) به‌جای (T, F)، مجموعه محدودیت برای C_2 می‌شود {(T, =), (F, =), (T,<), (T,>)}، تمام مجموعه محدودیت فوق شناسایی خطاهای اپراتور ربطی و بولین را در C_2 تضمین می‌کند.

به‌عنوان سومین مثال وضعیتی را به شکل زیر در نظر می‌گیریم:

$$C_3: E_1 > E_2) \& (E_3 = E_4)$$

که در آن E_1, E_2, E_3 و E_4 عبارات محاسباتی هستند. محدودیت شرطی برای C_3 عبارتست از فرم (D_1, D_2) که در آن هر کدام از D_2, D_1 عبارتست از >,=,<. از آن‌جا که C_3 مثل C_2 است به‌جز این‌که اولین شرط ساده در C_3 یک عبارت جبری است، می‌توانیم یک مجموعه محدودیتی توسط تغییر مجموعه محدودیت برای C_2، برای C_3 نیز بسازیم که داریم:

{(>,=), (=,=), (<,=), (>,>), (>,<)}

با در نظر گرفتن مجموعه فوق شناسایی خطاهای اپراتور ربطی در C_3 تضمین می‌شود.

۱۷-۵-۲ آزمون جریان داده[1]

این روش مسیرهای آزمونی یک برنامه را طبق محل تعاریف و کاربرد متغیرها در برنامه، انتخاب می‌کند. چندین شیوه مورد بررسی و مقایسه قرار گرفته‌اند.[FRA88] و[NTA88][2] و[FRA93][3] برای تشریح رهیافت آزمون جریان داده‌ها فرض کنید که به هرجمله در یک برنامه یک عدد منحصر به‌فرد اختصاص یافته و هر عملکرد یا تابع، پارامترها یا متغیرهای کلی خود را تغییر نمی‌دهند. برای حالتی که در آن S به‌عنوان عدد جمله می‌باشد:

DEF(S)={X |جمله S دارای تعریفی از X است}

USE(S) = {X|جمله S دارای کاربردی از X است}

اگر جمله S وضعیت IF یا LOOP باشد، مجموعه DEF آن تهی است و مجموعه USE آن براساس شرط یا جمله S است. تعریف متغیر X در جمله S را زندگی کردن در جمله Ś می‌گویند، اگر که مسیری از جمله S به جمله Ś وجود داشته باشد که دارای تعریف دیگری از X نباشد.

زنجیره تعریف ـ کاربرد[5] (یا زنجیره D-U) متغیر X به‌صورت {X, S,Ś} است که در آن S,Ś ارقام جمله، X در DEF(S) و USE(S) بوده و تعریف X در جمله S عبارتست از زندگی در جمله Ś.

یک شیوه ساده آزمون جریان داده‌ها عبارتست از درخواست این‌که هر زنجیره DU حداقل یکبار پوشش داده‌می‌شود. ما به این راهبرد، راهبرد آزمون DU می‌گوئیم. به نظر می‌رسد که آزمون DU پوشش تمام شاخه‌های برنامه را تضمین نمی‌کند. در هر حال یک شاخه به‌جز مواقع نادر مثل ساختارها IF- Then- Else که در آن Then تعریفی از هرگونه متغیری نداشته و بخش else وجود ندارد، با آزمون DU پوشش داده نمی‌شود.

راهبرد آزمون جریان داده‌ها برای انتخاب مسیرهایی از برنامه که حالات حلقه[6] و IF را دارا می‌باشند، مفیدند. برای تشریح این امر، کاربرد آزمون DU را برای انتخاب مسیرهایی برای PDL در نظر بگیرید که شامل:

```
proc x
Bl;
do while C1
 If C2
   then
```

توصیه

این تصور که آزمون جریان داده ها تنها برای سیستم های بزرگ کاربرد دارد، تصوری باطل است. به کلام دیگر، آزمون مذکور برای هر نقطه از نرم افزار که مضنون به خطا باشد، کاربرد دارد.

1.data flow

2.Frankl,P.G.and E.J.weyuker

3.Ntafo,.C.

4.Frankl,P.G. and S.Weiss

5.definition-use chain

6.loop

```
        if C4
          then B4;
          else B5;
        endif;
      else
        if C3
        then B2;
         else B3;
        endif;
      endif;
    enddo;
    B6;
  end proc;
```

برای به‌کارگیری شیوه آزمون DU در انتخاب مسیرهای آزمون نمودار کنترل جریان، لازم است تعاریف و کاربرد متغیرها را در هر وضعیت یا بلوک در PDL بدانیم. فرض کنید که متغیر X در آخرین جمله بلوک B_1, B_2, B_3, B_4, B_5 و B_5 تعریف شده و در اولین جمله بلوک‌های B_1, B_2, B_3, B_4, B_5 و B_6 استفاده می‌شود. شیوه آزمون DU مستلزم اجرای کوتاه‌ترین مسیر از هر یک از $0<i\leq5$ تا $1<j\leq6$, B_j می‌باشد. (چنین آزمونی هرگونه استفاده از متغیر X در وضعیت‌های C_1, C_2, C_3 و C_4 را نیز در برمی‌گیرد). گرچه زنجیره ۲۵ تایی متغیر X وجود دارد. اما ما تنها به پنج مسیر برای پوشش دادن این زنجیره‌های DU نیاز داریم. دلیلش این است پنج مسیر برای پوشش دادن زنجیره B_i از X, $0<i\leq5$ تا B_6 لازم بوده و دیگر زنجیره‌های DU می‌توانند با ساختن این پنج مسیر مشتمل بر تکرار حلقه‌ها بدست آیند.

از آن‌جا که وضعیت‌های یک برنامه طبق تعاریف و کاربرد متغیرها به یکدیگر مرتبطند، روش آزمون جریان داده‌ها برای شناسایی خطاها مؤثر است. در هر حال، مشکلات ارزیابی میزان پوشش برنامه امتحان و انتخاب مسیرهای آزمونی برای امتحان جریان داده‌ها مشکل‌تر از مشکلات مربوط به آزمون وضعیت است.

توصیه

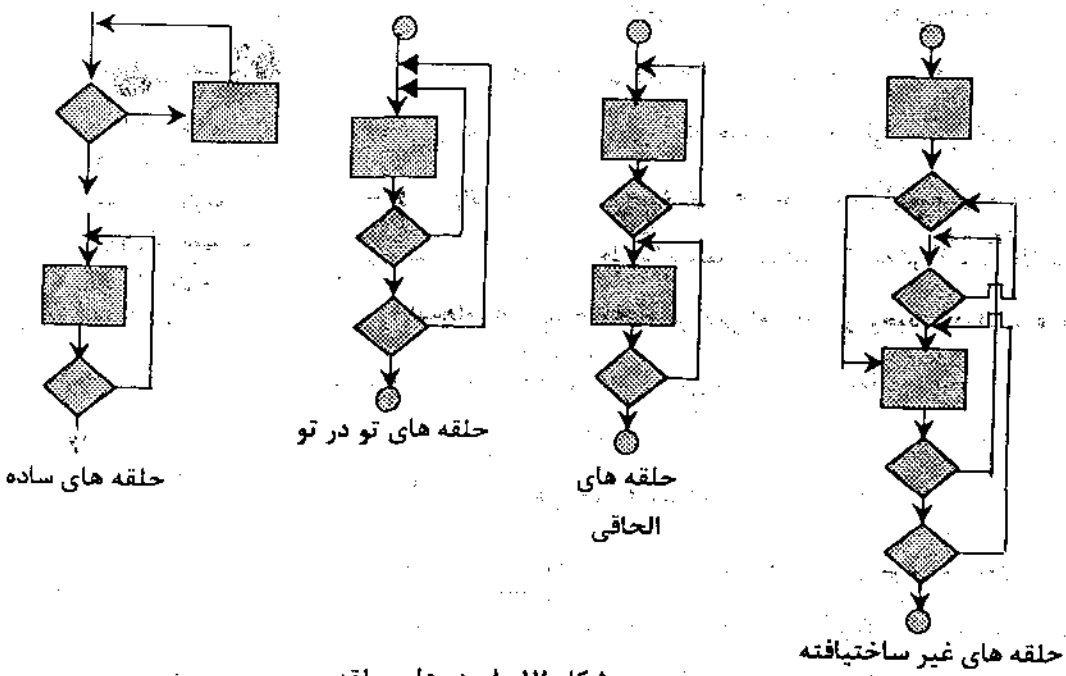
ساختارهای پیچیده حلقه، محل امن دیگری برای خطاهای پنهان است. تخصیص زمانی برای طراحی آزمون هایی که آزمایش ساختارهای حلقه را عهده دار شوند، ارزشمند خواهد

۱۷-۵-۳ آزمون حلقه

حلقه‌ها اساس و بنیان اکثریت عمده همه الگوریتم‌های پیاده‌سازی شده در نرم‌افزارند. ما اغلب در حین انجام آزمون توجه کمی به آنها می‌کنیم.

آزمون حلقه[1]، یک تکنیک آزمون جعبه سفید است[BEI90] که منحصراً روی اعتبار ساختمان‌های حلقه متمرکز می‌شود. می‌توان چهار کلاس از حلقه‌ها را تعیین نمود: حلقه‌های ساده، حلقه‌های تو در تو (Nested)، متسلسل (به‌هم پیوسته یا Concatened) و غیر ساختیافته.

1 Loop testing



شکل ۱۷ـ۸ رده های حلقه

حلقه یا لوپ ساده. مجموعه آزمونهای زیر را می‌توان در این نوع حلقه به‌کار گرفت که در آن n حداکثر تعداد عبورهای مجاز از حلقه است.

۱- بطور کامل از حلقه چشمپوشی کنید.

۲- تنها یک گذر از حلقه انجام دهید.

۳- دو گذر از حلقه انجام دهید.

۴- در جایی‌که m<n است m گذر از حلقه انجام دهید.

۵- n+1, n, n-1 گذر از حلقه انجام دهید.

حلقه‌های تو در تو یا Nested اگر قرار بود روش حلقه‌های ساده را به‌صورت حلقه‌های تو در تو بسط دهیم، تعداد آزمونها ممکن بود به‌صورت تصاعد هندسی با افزایش تودرتویی زیاد شود. بیزر[1] رهیافتی را بیان می‌دارد که به کاهش تعداد دفعات آزمون کمک می‌کند:

۱- از داخلی‌ترین حلقه شروع کنید ـ بقیه حلقه‌ها را روی حداقل مقدار تنظیم کنید.

۲- آزمونهای ساده حلقه را، برای داخلی‌ترین حلقه انجام دهید در عین‌حال حلقه‌های خارجی را در حداقل مقدار پارامتر تکرار قرار دهید. آزمونهایی دیگررا برای مقادیر خارج از دامنه یا مجزا اضافه کنید.

۳- کار را به طرف لایه‌های بیرونی‌تر ادامه دهید، آزمونهایی برای حلقه بعدی انجام داده اما دیگر حلقه‌های خارجی‌تر را در حداقل مقدار و دیگر حلقه‌های تو در تو را در مقادیر معمول نگه‌دارید.

1.Beizer,B.

۴- کار را ادامه دهید تا وقتی که همه حلقه‌ها مورد آزمون واقع شوند.

شما نمی توانید حلقه های ساخت نایافته را بگونه ای مؤثر مورد آزمون قرار دهید. (لذا) آنها را از اول طراحی کنید.

حلقه‌های پیوسته. می‌توان حلقه‌های پیوسته را با استفاده از روش تعریف شده برای حلقه‌های ساده، آزمون نمود و این در صورتی‌که حلقه‌ها مستقل از یکدیگر باشند. اگر دو حلقه به یکدیگر متصل شدند و حلقه‌ای با حلقه ۱ برخورد کرد، به عنوان مقدار اولیه برای حلقه ۲ انتخاب شده سپس حلقه‌ها دیگر مستقل نیستند. وقتی حلقه‌ها مستقل نیستند، روش به‌کار رفته در حلقه‌های تو در تو توصیه می‌شود.

حلقه‌های غیر ساختیافته. در جایی‌که امکان آن وجود دارد، این نوع حلقه‌ها را باید مجدداً طراحی نمود تا استفاده از سازه‌های برنامه‌نویسی ساختیافته را منعکس سازند.

۶-۱۷ آزمون جعبه سیاه

آزمون جعبه سیاه[1] که به آن آزمون رفتاری نیز می‌گویند، روی نیازمندیهای کارکردی نرم‌افزار متمرکز می‌شود. یعنی آزمون جعبه سیاه، مهندس نرم‌افزار را قادر می‌سازد مجموعه‌ای از وضعیتهای ورودی را بدست آورد که بطور کامل همه نیازمندیهای کارکردی را برای برنامه اجرا خواهند کرد. آزمون جعبه سیاه جایگزینی برای فنون جعبه سفید نیست. بلکه روش مکملی است که احتمالاً نوع متفاوتی از خطاها را نسبت به روش جعبه سفید، برملا می‌سازد.

آزمون جعبه سیاه سعی دارد خطاهایی را در گروههای زیر پیدا کند:

۱) کارکردهای نادرست یا گم‌شده.

۲) خطاهای رابط.

۳) خطاهای ساختارهای داده‌ای یا دسترسی به پایگاه‌های داده‌ای بیرونی.

۴) خطاهای رفتاری یا عملکردی.

۵) خطاهای شروع و خاتمه.

برخلاف آزمون جعبه سفید که در اوایل فرایند آزمون صورت می‌گیرد، آزمون جعبه سیاه در طول مراحل آخر آزمون انجام می‌گیرد. از آن‌جا که آزمون جعبه سیاه عمداً کنترل ساختار را مدنظر قرار نمی‌دهد، توجه بر دامنه اطلاعات معطوف می‌شود. آزمونهایی برای پاسخ‌گویی به سؤالات زیر صورت می‌گیرند:

- اعتبار کارکردی چگونه آزموده می‌شود؟
- چگونه رفتار و عملکرد سیستم مورد آزمون واقع می‌گردد؟
- چه نوع داده‌های ورودی، موارد آزمونی خوبی ایجاد می‌کنند؟
- آیا سیستم بطور خاص نسبت به مقادیر ورودی معینی حساس است؟

---

1. Black-box testing

- چگونه سرحدات یک کلاس داده مجزا می‌گردند؟
- چه مقدار داده و چه حجمی را سیستم می‌تواند تحمل نماید؟
- ترکیبات به‌خصوصی از داده‌ها چه اثری روی عملیات سیستم دارد؟

با به‌کارگیری فنون جعبه سیاه، مجموعه‌ای از موارد آزمونی که معیارهای زیر را برقرار می‌سازند، بدست می‌آوریم [MYE79]

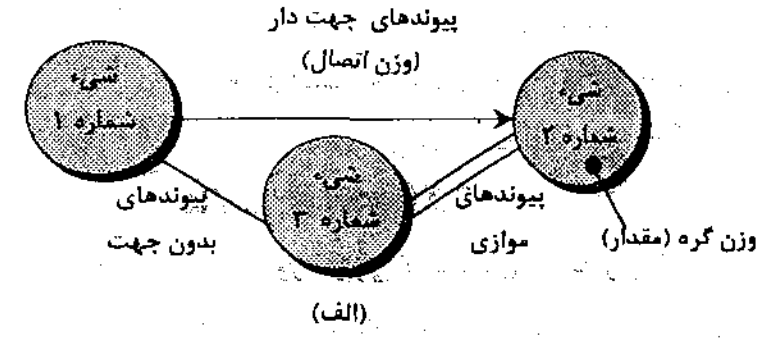
۱) موارد آزمونی که با عددی که بیشتر از یک است، تعداد موارد آزمونی اضافی را که باید طراحی شوند تا به یک آزمون منطقی برسیم، کاهش دهد و ۲) موارد آزمونی که گاهی در مورد وجود یا عدم وجود انواع خطاها به ما اطلاعاتی می‌دهند، به‌جای خطایی که تنها با آزمون به‌خصوصی که انجام می‌دهیم ارتباط دارد.

۱۷-۶-۱شیوه های آزمون مبتنی بر گراف

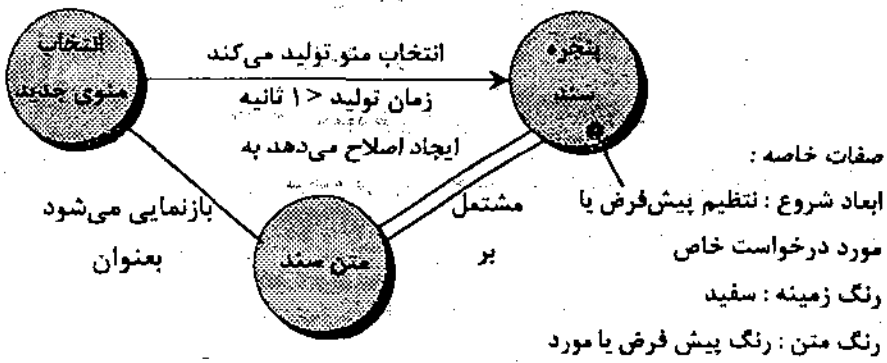
اولین مرحله در آزمون جعبه سیاه عبارتست از شناخت اشیایی[۲] که در نرم‌افزار مدل‌سازی شده و ارتباطاتی که این اشیاء را به هم مرتبط می‌کند. وقتی این کار صورت گرفت، مرحله بعدی عبارتست از تعریف یک‌سری آزمونهایی که تمام اشیایی را که دارای رابطه موردنظر با یکدیگر هستند، شناسایی می‌کنند. اگر بخواهیم به شکل دیگری بیان کنیم، آزمون نرم‌افزار با ایجاد نموداری از اشیای مهم و ارتباطاتشان آغاز شده و سپس یک‌سری آزمون تعبیه می‌شود که نمودار را تحت‌پوشش قرار می‌دهد به‌طوری‌که هر شی و رابطه آن به اجرا درآمده و خطاها مشخص می‌شوند. [BEI95]

۱. در این متن، اصطلاح شیٔ به اشیاء داده ای اطلاق می گردد که ما در فصل ۱۱ و۱۲ آن را توضیح داده ایم. اشیاء برنامه مانند پیمانه یا جملات زبان برنامه سازی خواهند بود.

2.Beizer,B.



(الف)



(ب)

شکل ۱۷ـ۹ (الف) علائم گراف
(ب) مثالی ساده

برای موفقیت این مراحل، مهندس نرم‌افزار کار را با ایجاد یک گراف[1] آغاز می‌کند یعنی مجموعه‌ای از گره‌ها[2] یعنی مجموعه‌ای از گرهها که بازنمای اشیاء می‌باشند؛ اتصال‌ها[3] (رابطها) که بازنمای رابطه میان اشیاء هستند؛ وزن گرهها[4] که خواص یک گره را مشخص می‌کنند و وزن رابطها[5] که بعضی از خصوصیات یک رابط را معین می‌کنند.[6]

1.graph
2.nodes
3.links
4.node weights
5.link weights

۶ این مفاهیم اندکی مبهم می نمایند. به خاطرآورید که گرافها در بخش ۱۷-۴-۱ نیز مورد استفاده قرار گرفتند تا گرافی برای شیوه اولیه آزمون مسیر، ایجاد شود. گره های گراف برنامه مشتمل بر دستورالعمل ها (اشیاء برنامه) بود. خواه در طراحی رویه ای یا برنامه منبع، و یالهای جهت دار، جریان کنترل میان اشیاء برنامه را نشان می داد. در اینجا، استفاده از گراف جهت آزمون جعبه سیاه به خوبی دیده می شود.

نمایش سمبولیکی از یک نمودار در شکل ۱۷-۹الف نشان داده شده است. گره‌ها به‌صورت دوایری نشان داده شده‌اند که به‌وسیله رابطها به‌هم متصلند و فرم‌های مختلفی دارند. یک رابط جهت‌دار[1] (به‌وسیله فلش نشان داده شده) نشان‌گر این است که ارتباط تنها به‌صورت یک طرفه حرکت می‌کند. در رابطه دو طرفه[2] یا رابط متقارن[3] نشان داده می‌شود که ارتباط در هر دو طرف به‌کار گرفته می‌شود. رابطهای موازی[4] وقتی استفاده می‌شوند که چند رابطه مختلف بین گره‌های نمودار ایجاد شده باشند. به‌عنوان یک مثال ساده، بخشی از نمودار را در مورد یک برنامه کاربردی کلمه‌پرداز (شکل ۱۷-۹ب)در نظر بگیرید که در آن:

انتخاب منوی فایل جدید = شی شماره ۱

پنجره سند = شی شماره ۲

متن سند = شی شماره ۳

است.

با رجوع به شکل، از انتخاب منو روی **فایل جدید** یک **پنجره سند** یا مستند ایجاد می‌شود. اندازه وزن گره **پنجره سند** فهرستی از ویژگیهای مختلف پنجره را که به هنگام تولید پنجره انتظار آنها را داریم، ارائه می‌دهد. اندازه رابط نشان‌گر این است که پنجره باید در کمتر از یک ثانیه ایجاد شود. یک رابط بدون جهت، ارتباطی متقارن را بین **گزینه انتخاب منوی فایل جدید** و **متن سند** ایجاد نموده و رابطهای موازی نشان‌دهنده ارتباطاتی بین **پنجره سند** و **متن** آن می‌باشد. در واقع، باید یک گراف بسیار دقیق‌تر به‌عنوان مقدمه ایجاد شود تا طراحی مورد آزمون را انجام دهد. سپس مهندس نرم‌افزار با پیمایش گراف و کنترل کردن ارتباطات نشان داده شده، موارد آزمونی را بدست می‌آورد.

بیزر[BEI95][5] چند روش آزمون رفتار را توصیف می‌کند که می‌توانند از گرافها استفاده کنند:

مدل‌سازی جریان تراکنش. گره‌ها نمایان‌گر مراحلی از تراکنش بوده (مثل مراحل لازم برای انجام رزرو بلیط هواپیما با استفاده از یک سرویس on-line) و رابطها نمایان‌گر ارتباط منطقی بین مراحل هستند (مثل ورودی‌های **اطلاعات پرواز** که به دنبال پردازش اعتبار/ دسترسی[6] صورت می‌گیرد.) می‌توان از نمودار جریان داده‌ها برای کمک به ایجاد نمودارهایی از این نوع کمک کرد.

1.directed link

2.bidirectional link

3.symmetric link

4.parallel links

5.Beizer,B.

6.validation/availability processing

مدل‌سازی حالت محدود و معین. گره‌ها نمایان‌گر حالات قابل مشاهده کاربر نرم‌افزار است (مثل صفحاتی که با ارائه سفارش جنس توسط فروشنده وارد می‌شوند) و رابطه‌هایی که نمایان‌گر انتقالاتی هستند که برای تغییر از حالتی به حالت دیگر رخ می‌دهند. (مثل اطلاعات سفارش که در طول پیدا کردن دسترسی به موجودی[1] مشخص شده و به دنبال آن اطلاعات صورت‌حساب مشتری داده می‌شود.) نمودار تغییر حالت را می‌توان برای کمک به تولید نمودارهایی از این نوع استفاده نمود.

مدل‌سازی جریان داده‌ها. گره‌ها اشیای داده‌ای بوده و رابطه‌ها تغییراتی هستند که برای تغییر یک شی داده‌ای به شی‌دیگر رخ می‌دهند. مثلاً،گره FICATax.Withheld (FTW) از روی (GW) Gross.Wages با استفاده از رابطه (FTW)= ۰/۶۲× GW محاسبه می‌شود.

مدل‌سازی زمان‌بندی. این گره‌ها اشیای برنامه بوده و رابطه‌ها ارتباطات سریالی و پیوسته‌ای میان این اشیاء هستند. اندازه رابطه‌ها برای مشخص کردن زمان لازم اجرا به هنگام شروع برنامه، استفاده می‌شود.

مبحث مفصلی از هر یک از این روش‌های آزمونی فراتر از حوزه مباحث این کتاب است. خوانندگان علاقه‌مند را برای بحث دقیق‌تر به [BEI95][2] ارجاع می‌دهیم. تهیه یک گزارش کلی از رهیافت آزمون مبتنی بر نمودار ارزشمند است.

پرسش

راهبرد کلی آزمون نرم افزار کدام است؟

آزمون مبتنی بر نمودار با تعریف همه گره‌ها و ارزش آنها آغاز می‌شود. یعنی اشیاء و صفاتشان شناسایی می‌شوند. از مدل داده‌ای می‌توان به‌عنوان نقطه شروع استفاده نمود، اما توجه به این نکته مهم است که بسیاری از گره‌ها ممکن است اشیاء برنامه باشند (که صریحاً در مدل داده‌ای نمایش داده نشده‌اند.) به‌منظور اشاره‌ای به نقاط شروع و پایان، تعریف گره‌های ورودی و خروجی مفید است. وقتی گره‌ها شناسایی شدند، رابطه‌ها و ارزش آنها باید معین گردد. به‌طور کلی، رابطه‌ها باید نام‌گذاری شدند، گرچه رابطه‌هایی که نمایان‌گر جریان کنترلی بین اشیای برنامه هستند، نیازی به نام‌گذاری ندارند.

در بسیاری از موارد، مدل گراف ممکن است دارای حلقه باشد (یعنی مسیری در نمودار که در آن یک یا چند گره، بیشتر از یکبار با هم برخوردارند.) آزمون حلقه را می‌توان در سطح رفتاری به‌کار گرفت (جعبه سیاه). نمودار به شناسایی حلقه‌هایی که باید آزموده شوند، کمک می‌کند.

هر رابطه به‌صورت جداگانه بررسی می‌شود، به‌طوری‌که می‌توان موارد آزمون را بدست آورد. قابلیت انتقال[3](گذار) در روابط متسلسل و پیوسته برای تعیین چگونگی تأثیر اشاعه روابط بر اشیای تعریف شده در گراف، بررسی می‌شود. این خاصیت را می‌توان با در نظر گرفتن سه شی Z,Y,X تشریح کرد. روابط زیر را در نظر بگیرید:

X برای محاسبه Y لازم است.

1.inventory-availability look-up

2.Beizer,B.

3.transitivity

Y برای محاسبه Z لازم است.

بنابراین، رابطه تراگذری و انتقالی بین X و Z وجود دارد.

X برای محاسبه Z لازم است.

براساس این رابطه تراگذری، آزمونهایی برای یافتن خطا در محاسبات Z، باید یک‌سری مقادیر X و Y را مدنظر داشته باشند.

تقارن یک رابطه[1] نیز یک راهنمای مهم برای طراحی موارد آزمون است. اگر رابطی دو طرفه بود (متقارن)، نکته مهم، آزمون این خاصیت است. مشخصه UNDO در بسیاری از برنامه‌های کامپیوترهای شخصی نشان‌گر تقارن محدود است. یعنی UNDO امکان این را فراهم می‌کند که عملی بعد از تکمیل شدن، باطل گردد. این کار را باید کاملاً آزمود و باید به همه استثناهاً توجه نمود (یعنی جاهایی که در آن نمی‌توان از عمل UNDO استفاده کرد.) نهایتاً هر گره گراف باید دارای رابطه‌ای باشد که به خودش برمی‌گردد. در واقع هیچ عملی یا عمل بیهوده‌ای وجود ندارد. این رابطه‌های بازتابی[2] باید آزمون شوند.

با شروع طراحی مورد آزمون، اولین هدف، دستیابی به پوشش کامل گره[3] است. منظور ما از این گفته این است که آزمون‌هایی باید طراحی شوند که نشان‌دهند هیچ گره‌ای به‌صورت تصادفی حذف نشده و وزن گره‌ها درستند.

بعد از آن نوبت به پوشش دادن اتصال[4] می‌رسد. هر ارتباط بر اساس خاصیت‌هایش آزمون می‌شود. مثلاً، یک رابطه متقارن برای این‌که مشخص شود در واقع دو طرفه است، آزمون می‌گردد. یک رابطه زودگذر برای مشهود بودن خاصیت گذرا بودن، مورد آزمون واقع می‌شود. رابطه انعکاسی برای اطمینان از این‌که یک حلقه تهی وجود دارد صورت می‌گیرد. وقتی وزن اتصال مشخص شد، آزمونهایی صورت می‌گیرد تا نشان دهد که این مقادیر معتبرند دارند. در آخر، آزمون حلقه صورت می‌گیرد.

۱۷-۶-۲ تجزیه هم ارزی

تقسیم و تجزیه هم ارزی[5] یک روش از آزمون جعبه سیاه است که قلمرو ورودی برنامه را در گروه‌های مختلفی از داده‌ها تقسیم می‌کند که از آنها موارد آزمون بدست می‌آیند. یک مورد آزمون ایده‌آل به تنهایی یک گروه از خطاهایی را مشخص می‌کند. (مثل پردازش نادرست همه داده‌های کاراکتر) که ممکن است در غیر این‌صورت نیازمند اجرای موارد آزمونی بسیاری قبل از مشاهده خطاهای کلی باشند.

1.symmetry
2.reflexive
3.node coverage
4.link coverage
5.Equivalence partitioning

این کار سعی دارد مورد آزمونی را تعریف کند که گروههایی از خطاها را مشخص می‌کنند که به وسیله آن تعداد کلی موارد آزمونی را که باید ارائه شوند، کاهش می‌دهند.

طراحی مورد آزمون برای تقسیم هم ارزی بر اساس ارزیابی گروههای هم ارز برای یک شرط ورودی است. با استفاده از مفاهیم معرفی شده در بخش قبلی، اگر مجموعه‌ای از اشیاء را بتوان به‌وسیله ارتباطاتی متقارن، انتقال‌پذیر و انعکاسی به‌هم مرتبط کرد، یک رده هم ارزی به‌وجود می‌آید. یک رده[1] هم‌ارزی نمایان‌گر مجموعه‌ای از حالات با ارزش و غیر معتبر برای وضعیت‌های ورودی است. معمولاً، یک شرط ورودی یک مقدار رقمی خاص، یک‌سری مقادیر، مجموعه‌ای از ارزش‌های مربوطه یا شرط دوارزشی است. ممکن است گروههای هم ارزی طبق رهنمودهای زیر تعریف شوند:

۱- اگر یک شرط ورودی یک طیف[2] را تعریف کند، یک کلاس (رده) هم ارزی معتبر و دو کلاس هم‌ارزی غیر معتبر تعریف می‌شوند.

۲- اگر یک شرط ورودی نیازمند یک ارزش[3] خاص باشد، یک کلاس هم‌ارزی معتبر و دو کلاس هم‌ارزی غیر معتبر تعریف می‌شوند.

۳- اگر یک شرط ورودی عضوی از یک مجموعه[4] را مشخص کند، یک کلاس هم ارزی معتبر و یک کلاس هم ارزی ارزش تعریف می‌گردند.

۴- اگر یک شرط ورودی بولین[5] (دوارزشی) باشد، یک کلاس هم‌ارزی معتبر و یک کلاس هم‌ارزی غیر معتبر تعریف می‌شوند.

به‌عنوان یک مثال، داده‌های استفاده شده به‌عنوان بخشی از برنامه خودکار بانکداری را در نظر بگیرید. کاربر می‌تواند با استفاده از کامپیوتر شخصی خود به بانک دسترسی داشته، یک کلمه رمز شش رقمی را وارد کند و بعد یک‌سری فرمان تایپ شده ارائه دهد که عملکردهای مختلف بانکی را آغاز می‌کنند.

در طول زمان اتصال نرم‌افزاری که برای کار بانکی تعبیه شده، داده‌ها را به‌صورت زیر قبول می‌کند:

کد ناحیه ـ جای خالی یا عدد سه رقمی

پیشوند ـ عدد سه رقمی که با صفر یا یک شروع نمی‌شود.

پسوند ـ عدد چهار رقمی

کلمه رمز ـ یک رشته الفبای شش تایی

فرمان‌ها ـ Check (چک کردن)، Deposit (سپرده)، BillPay (پرداخت صورت‌حساب) و غیره.

1.equivalence class

2.range

3.value

4.set

5.Boolean

شرایط ورودی مربوط به هر یک از عناصر داده‌ای برای کاربرد بانکی را می‌توان به‌صورت زیر تعریف نمود:

کد ناحیه : شرط ورودی، دوارزشی- کد ناحیه ممکن است موجود نباشد.

شرط ورودی ـ طیف ـ مقادیر تعریف شده بین ۲۰۰ تا ۹۹۹ با استثنائات معینی

پیشوند : شرط ورودی، دامنه ـ مقدار خاصی > ۲۰۰ بدون رقم‌های صفر

شرط ورودی، مقدار ـ چهار رقم طول

کلمه عبور : شرط ورودی، بولین ـ ممکن است کلمه رمز وجود داشته یا نداشته باشد.

شرط ورودی، مقدار ـ یک رشته ۶ کاراکتری

فرمان : شرط ورودی، مجموعه ـ شامل فرمان‌های مورد اشاره فوق.

با به کارگیری این رهنمودها برای بدست آوردن کلاس‌های هم ارز، موارد آزمونی برای هر مورد داده‌ای در دامنه ورودی را می‌توان بدست آورد و اجرا نمود. موارد آزمون انتخاب می‌شوند، به‌طوری‌که بیشترین تعداد صفات خاصه یک کلاس هم ارزی در یک لحظه اجرا می‌گردند.

۱۷-۶-۳ تحلیل مقادیر مرزی

بنا به‌دلایلی که کاملاً روشن نیست، بیشتر خطاها در سرحدات دامنه ورودی رخ می‌دهند تا در مرکز. به‌همین دلایل است که تحلیل مقدار سرحد[1] (BVA) به‌عنوان یک تکنیک آزمونی ارائه شده است. BVA منجر به انتخاب موارد آزمونی می‌شود که مقادیر سرحد را می‌آزمایند.

BVA یک تکنیک طراحی مورد آزمون است که تقسیم‌بندی هم ارزی را تکمیل می‌کند. علاوه بر انتخاب هر عنصری از کلاس هم ارزی، BVA منجر به انتخاب موارد آزمونی در لبه‌های کلاس می‌شود. به‌جای این‌که منحصراً بر شرایط ورودی متمرکز شویم، BVA موارد آزمونی بدست می‌آورد که از دامنه خروجی نیز هستند. [MYE79][2]

پرسش

چه هنگام آزمون را به پایان برده ایم؟

رهنمودهای مربوط به BVA از بسیاری جهات مشابه موارد مربوط به تقسیم‌بندی هم ارزی هستند:

۱- اگر یک شرط ورودی، طیفی را مشخص کند که به‌وسیله مقادیر a و b محدود شده‌اند، موارد آزمونی باید طراحی شوند که به‌ترتیب بالا و پایین مقادیر a و b باشند.

۲- اگر یک شرط ورودی چند مقدار را مشخص کند، موارد آزمونی باید ارائه شوند که حداقل و حداکثر ارقام را آزمایش نمایند. مقادیری که درست بالا و پایین مقادیر حداقل و حداکثر هستند نیز، آزمون می‌شوند.

---

1 Boundary Value Analysis

2 Myer, G.

۳- رهنمودهای ۱ و ۲ شرایط ورودی را در شرایط خروجی به‌کار گیرید. مثلاً، فرض کنید که جدول دما در برابر فشار نیازمند خروجی از برنامه تحلیل مهندسی است. موارد آزمونی باید طراحی شوند که گزارش خروجی ارائه دهند که حداقل مقدار مجاز (و حداکثر) را در مقادیر عناصر جدول ایجاد کند.

۴- اگر ساختار داده‌ای برنامه داخلی سرحدات را مشخص نموده باشد. (مثل یک آرایه که دارای حد تعریف شده ۱۰۰ ورودی است)، مطمئن باشید که یک مورد آزمونی برای آزمون ساختار داده‌ای در سرحد آن طراحی می‌شود.

اکثر مهندسان نرم‌افزار ذاتاً تا حدی BVA را انجام می‌دهند. با به‌کارگیری رهنمودهای فوق‌الذکر، آزمون سرحد کامل‌تر می‌شود، بدین‌وسیله احتمال بیشتری برای یافتن خطا وجود دارد.

۱۷-۶-۴ آزمون مقایسه ای

چند موقعیت وجود دارد (مثل الکترونیک هوابرد، کنترل نیروگاه هسته‌ای) که در آن قابلیت اطمینان نرم‌افزار بسیار مهم و حساس است. در چنین مواردی اغلب از نرم‌افزار و سخت‌افزار افزونه برای به حداقل رساندن احتمال خطا استفاده می‌شود. وقتی نرم‌افزار افزونه ارائه شد، تیم‌های مهندسی نرم‌افزاری جداگانه‌ای نسخه‌های مستقل از برنامه کاربردی را با استفاده از همان مشخصه‌ها تولید می‌کنند. در چنین مواقعی، هر نسخه با همان داده‌ها آزمون می‌شود تا مطمئن شویم که همگی خروجی یکسانی دارند. سپس تمام نسخه‌ها به‌صورت موازی با مقایسه زمان واقعی نتایج به اجرا درمی‌آیند تا از سازگاریشان مطمئن شویم.

با استفاده از آموخته‌های خود از سیستم‌های افزونه، محققان بیان داشته‌اند[BRI87][1] که نسخه‌های مستقلی از نرم‌افزار برای کاربردهای مهم حتی وقتی که یک نسخه در سیستم مبتنی بر کامپیوتر استفاده می‌شود، تولید گردد. این نسخه‌های مستقل اساس کار آزمون جعبه سیاه را تشکیل می‌دهند که آزمون مقایسه[2] یا آزمون پشت به پشت[3] نامیده می‌شود.[KNI89][4]

با اجراهای متعددی از یک مشخصه یکسان، موارد آزمونی با استفاده از فنون دیگر جعبه سیاه طراحی می‌شوند که به‌عنوان ورودی برای هر نسخه از نرم‌افزار هستند. اگر خروجی هر نسخه یکسان باشد، فرض می‌شود که همه اجراها درست هستند. اگر خروجی مختلف بود، هر برنامه مورد بررسی قرار می‌گیرد تا

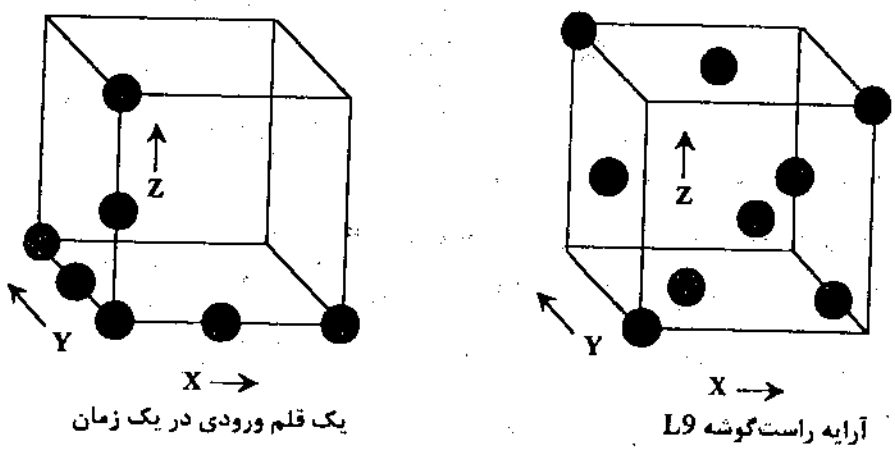
1.Brilliant,S.S.
2.comparison testing
3.back-to-back testing
4.Knight,J. and P.Ammann

معلوم شود آیا نقصی در یک یا چند نسخه وجود دارد که مسئول این اختلاف هست یا خیر. در اکثر موارد، مقایسه خروجی ها را می‌توان با ابزار خودکار انجام داد.

۱۷-۶-۵ آزمون آرایه راستگوشه (متعامد)

برنامه‌های کاربردی بسیاری وجود دارند که در آنها میدان ورودی نسبتاً محدود است. یعنی، تعداد پارامترهای ورودی کم است و مقادیر هر یک از پارامترها مشخصاً محدود می‌باشد. وقتی این اعداد بسیار کم باشند (مثل ۳ پارامتر که هر کدام سه مقدار مجزا می‌گیرند)، ممکن است هر تغییر ورودی را در نظر گرفته و به‌طور کامل پردازش میدان ورودی را بیازماییم. با افزایش تعداد مقادیر ورودی و تعداد مقادیر مجزا برای هر یک از قلم‌های داده، آزمون جامع غیر عملی و غیر ممکن می‌گردد.

می‌توان آزمون آرایه متعامد[1] را در مسایلی به‌کار گرفت که در آن دامنه ورودی نسبتاً کوچک اما برای انطباق با آزمون جامع بسیار بزرگ باشند. این روش به‌طور خاص برای یافتن خطاهای مربوطه به خطاهای منطقه‌ای[2] مفید است، یعنی یک دسته خطا که مربوط به منطق نادرست در جز نرم‌افزاری است.



یک قلم ورودی در یک زمان — آرایه راست‌گوشه L9

شکل ۱۷ـ۱۰ یک دید ژئو متری از موارد آزمون

1.Orthogonal array testing

2.region faults

به‌منظور تشریح تفاوت میان آزمون آرایه متعامد و رهیافت های قراردادی‌تر "یک مورد ورودی در هر بار" سیستمی را در نظر بگیرید که دارای سه مورد ورودی X, Y و Z است. هر کدام از این موارد دارای سه مقدار مجزا مربوط به آن است.

۲۷۰=۳^۳ مورد آزمونی ممکن وجود دارد. فادکه[PHA97] [1] نظریه هندسی از موارد آزمون احتمالی مربوط به X, Y و Z را بیان می‌دارد که در شکل ۱۷-۱۰ آمده‌اند. با توجه به شکل، ممکن است یک داده در هر بار در طول یک رشته از محور ورودی تغییر کند. این کار منجر به شمول نسبتاً محدودی از دامنه ورودی می‌شود (که به‌وسیله مکعب دست چپ در شکل مشخص شده است.)

به هنگام آزمون آرایه متعامد، یک آرایه متعامد[2] L۹ از موارد آزمونی ایجاد می‌شود. این آرایه دارای خاصیت توازن است.[PHA97] [3] یعنی موارد آزمونی به‌صورت یکنواخت در سراسر دامنه آزمون پخش می‌شوند. همان‌گونه که در مکعب سمت راست شکل ۱۷-۱۰ آمده است. پوشش آزمون در طول دامنه ورودی کامل‌تر است.

به‌منظور تشریح استفاده از آرایه متعامد L۹ عملکرد Send (فرستادن) را برای برنامه فکس در نظر بگیرید.

چهار پارامتر P_1, P_2, P_3 و P_4 انجام می‌گیرند تا به عمل ارسال[4] بروند، هر کدام سه مقدار مجزا دارند. مثلاً، P_1 مقدار زیر را دارد:

الان بفرست, $P_1=1$

آن را یکساعت دیگر بفرست, $P_1=2$

آن را بعد از نیمه شب بفرست, $P_1=3$

P_3P_2 و P_4 نیز مقادیر ۱ و ۲ و ۳ را نیز دارند که دیگر اعمال عمل فرستادن را مشخص می‌کند. اگر شیوه «یک قلم ورودی در هر بار» انتخاب شد، رشته آزمونهای زیر (P_4, P_3, P_2, P_1) مشخص می‌شود. (۱، ۱، ۱، ۱)، (۲، ۱، ۱، ۱)، (۳، ۱، ۱، ۱)، (۱، ۲، ۱، ۱)، (۱، ۳، ۱، ۱)، (۱، ۱، ۲، ۱)، (۱، ۱، ۳، ۱)، (۱، ۱، ۱، ۲) و (۱، ۱، ۱، ۳). *فادکه* این موارد آزمون را به‌صورت زیر برآورد می‌کند:[PHA97] [5]

چنین موارد آزمونی تنها وقتی مفید هستند که مطمئن باشید که این پارامترهای آزمونی با هم تداخل ندارند. آنها می توانند خطاها و عیوب منطقی را در جایی تشخیص دهند که یک پارامتر به تنهایی

1.phadke,M.
2.orthogonal array
3.phadke,M.
4.send
5.phadke,M.

باعث عدم کارکرد نرم‌افزار می‌شوند. این اشتباهات، خطاهای تک مورد[1] هستند. این روش نمی‌تواند خطاهای منطقی را که باعث عدم کارکرد درست می‌شود، زمانی که دو یا چند پارامتر به‌طور هم‌زمان مقادیر معینی به‌خود می‌گیرند، تشخیص دهد. یعنی نمی‌تواند هیچ‌گونه ارتباطی را تشخیص دهد. بنابراین توانایی‌اش در تشخیص نقص محدود است.

با فرض تعداد نسبتاً کم پارامترهای ورودی و مقادیر مجزای آن، آزمون جامع و کامل ممکن می‌شود. تعداد آزمونهای لازم $۳^۴=۸۱$ است که زیاد، اما قابل سازمان‌دهی است. تمام عیوب مربوط به جایگشت داده‌ها پیاد می‌شوند. اما انجام این کار مستلزم تلاش زیادی است.

روش آزمون آرایه متعامد ما را قادر می‌سازد با آزمون‌های کمتری نسبت به شیوه جامع، به آزمون خوبی برسیم. یک آرایه متعامد L۹ برای عملکرد ارسال فکس در شکل ۱۷-۱۱ نشان داده شده است.

مورد آزمون	پارامترهای آزمون			
	P1	P2	P3	P4
۱	۱	۱	۱	۱
۲	۱	۲	۲	۲
۳	۱	۳	۳	۳
۴	۲	۱	۲	۳
۵	۲	۲	۳	۱
۶	۲	۳	۱	۲
۷	۳	۱	۳	۲
۸	۳	۲	۱	۳
۹	۳	۳	۲	۱

شکل ۱۷ ـ ۱۱ یک آرایه راست‌گوشه L9

1 single mode faults

فادکه نتایج آزمونها را با استفاده از آرایه متعامد L_9 به شکل زیر، ارزیابی می‌کند:[PHA97] [1]

شناسایی و جداسازی تمام غلطهای تک حالته. یک غلط تک حالته در همه سطوح هر گونه پارامتری، به‌طور مستمر مشکل‌زاست. مثلاً، اگر همه موارد آزمونی عامل $P_1=1$ باعث ایجاد خطا شوند، این یک خطای تک حالته است. در این مثال آزمون‌های ۱، ۲ و ۳ [شکل ۱۷-۱۱] خطاها را نشان می‌دهند. با تحلیل اطلاعات موجود در مورد این‌که کدام آزمونها خطاها را نشان می‌دهند، می‌توان شناسایی کرد که کدام مقادیر پارامتری باعث خطا می‌گردند. در این مثال با توجه به این‌که آزمونهای ۱، ۲ و ۳ باعث خطا می‌شوند، می‌توان پردازش منطقی مربوط به ارسال را جدا نمود که منبع خطا است. چنین جداسازی برای از بین بردن خطا بسیار مهم است.

شناسایی تمام خطاهای دو حالته. اگر یک مشکل دائمی وجود داشته باشد و این در هنگامی باشد که دو پارامتر با هم اجرا می‌گردند، آن را خطای دو حالته[2] می‌نامند. در واقع، خطای دو حالته نشانه عدم سازگاری دوگانه و تأثیرات متقابل مضری است که بین دو پارامتر وجود دارد.

خطاهای چند حالته. آرایه‌های متعامد می‌توانند کار شناسایی خطاهای یک یا دو حالته را حتمی سازند. بسیاری از خطاهای چند حالته نیز با این آزمونها شناسایی می‌شوند.

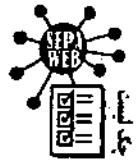
بحث مفصلی از مورد آزمون آرایه متعامد در قسمت [PHA89][3] دیده می‌شود.

۱۷-۷ آزمون برای محیط ها، معماری ها، و کاربردهای خاص

رهنمودهایی در خصوص طراحی رابط گرافیکی کاربر (GUI) در فصل ۱۶ ارائه شده اند.

با پیچیده‌تر شدن نرم‌افزار کامپیوتری، نیاز به آزمونهای تخصصی نیز افزایش می‌یابد. روش‌های آزمون جعبه سیاه و سفید، که در بخش‌های ۱۷-۵ و ۱۷-۶ در تمام برنامه‌های کاربردی، محیط‌ها و معماری‌ها قابل اجرا هستند. اما رهنمودها و رهیافتهای منحصر به فردی گاهی برای آزمون مورد نیازند. در این بخش ما راهنمایی‌هایی را در مورد محیط‌ها، معماری‌ها و برنامه‌های کاربردی خاص بررسی می‌کنیم که معمولاً مهندسان کامپیوتری با آن برخورد دارند.

۱۷-۷-۱ آزمون رابط های گرافیکی کاربر (GUI)



آزمون های GUI

رابطهای کاربر گرافیکی یا GUIها نمایان‌گر چالش‌های جالبی برای مهندسین نرم‌افزار هستند. به‌خاطر اجزای قابل استفاده مجدد که به‌عنوان بخشی از محیط‌های تولیدی GUI مهیا شده‌اند، ایجاد رابط کاربر زمان کم‌تری گرفته و دقیق‌تر نیز هست. اما در عین حال پیچیدگی GUI نیز افزایش یافته که منجر به مشکلات بیشتری در طراحی و اجرای موارد آزمون شده است.

1.phadke,M.

2.double mode fault

3.phadke,M.

از آن‌جا که بیشتر GUIهای مدرن ظاهر و حالات یکسانی دارند، یک‌سری آزمون‌های استاندارد ارائه شده‌اند. گراف های مدل‌سازی حالت محدود، ممکن است برای بدست آوردن یک‌سری آزمون‌ها استفاده شوند که اشیای برنامه‌ای و داده‌ای خاصی را که مربوط به GUI هستند، موردنظر قرار می‌دهد.

به‌خاطرات تغییرات عمده متعددی که مربوط به عملیات GUI است آزمون‌ها باید با ابزارهای خودکار صورت گیرند. یک‌سری از ابزارهای آزمون GUI به‌صورت انبوه در سال‌های اخیر به بازار آمده‌اند. برای بحث بیشتر، به فصل ۳۱ مراجعه شود.

۱۷-۷-۲ آزمون معماری های خادم / مخدوم (C/S)

مباحث مربوط به مهندسی نرم افزار خادم / مخدوم در فصل ۲۸ ارائه شده اند.

معماری یا ساختارهای C/S نمایان‌گر چالش بزرگی برای آزمون‌کنندگان نرم‌افزار است. ماهیت توزیع شدهٔ محیط‌های C/S، موضوعات عملکردی مربوط به پردازش تراکنش ها، حضور بالقوه یک‌سری پایگاه‌های سخت‌افزاری مختلف، پیچیدگی‌های ارتباط شبکه‌ای، نیاز خدمات به کاربران متعدد از یک پایگاه داده‌ای مرکزی و نیازمندیهای هماهنگی که برروی خادم اعمال شده، همگی کار آزمون معماری C/S و نرم‌افزاری که در آن قرار گرفته را نسبت به برنامه‌های متکی (Standalone) مشکل‌تر می سازند. در واقع، بررسی‌های صنعتی که اخیراً صورت گرفته نشان‌گر افزایش قابل توجهی در زمان و هزینهٔ آزمون، هنگامی است که محیط‌های C/S ارائه شده‌اند.

۱۷-۷-۳ آزمون مستندات و تسهیلات کمکی (راهنما)

عبارت «آزمون نرم‌افزار»[1] تصویرهایی از موارد آزمونی متعددی را که برای اجرا روی برنامه‌های کامپیوتری آماده شده و داده‌هایی که باید دستکاری شوند را، به ذهن ما تداعی می‌کنند. با یادآوری تعریف نرم‌افزار[2] که در اولین فصل این کتاب ارائه شد، نکته مهم توجه به این مطلب است که آزمون باید تا حد سومین عنصر پیکربندی، یعنی مستندات بسط یابد.

چه رهنمودهایی برای یک راهبرد موفق آزمون، وجود دارند؟

خطاهای مستندسازی در هنگام قبول برنامه، مثل خطاهای داده‌ای یا کد منبع می‌تواند مخرب باشد. هیچ چیز ناامیدکننده‌تر از پیروی از دستورالعمل کاربر یا تسهیلات کمکی به‌صورت دقیق و رسیدن نتایجی برخلاف نتایج پیش‌بینی شده، نیست. به همین دلیل است که آزمون مستندسازی باید بخش با مفهومی از هرگونه طرح آزمون نرم‌افزار باشد.

1.software testing.

2.software

این آزمون می‌تواند در دو مرحله صورت گیرد. اولین مرحله، بازنگری و بازرسی[1] است که اسناد را از نظر ویرایشی بازبینی می‌کند. دومین مرحله، آزمون زنده[2] است که از مستندات در ارتباط با استفاده از برنامه واقعی استفاده می‌کند.

در کمال تعجب آزمون زنده برای مستندسازی می‌تواند با استفاده از فنونی صورت گیرد که در مورد بسیاری از روش‌های جعبه سیاه که در بخش ۶-۱۷ بحث شد، مبهم و پیچیده باشند. آزمون مبتنی بر گراف را می‌توان برای توصیف کاربرد برنامه، استفاده نمود. تقسیم بندی هم ارزی و تحلیل ارزش سرحد را می‌توان برای تعریف دسته‌های مختلفی از ارتباطات متقابل مربوطه و داده‌های ورودی به‌کار گرفت.

۴-۷-۱۷ آزمون سیستم های زمان واقعی

ماهیت وابستگی زمانی و ناهمگامی[3] بسیاری از برنامه‌های کاربردی زمان واقعی یک موجودیت جدید و تقریباً مشکل را به ترکیب آزمونی اضافه می‌کند که زمان است. نه‌تنها طراح آزمون باید موارد آزمونی جعبه سیاه و سفید را در نظر بگیرد، بلکه برخورد با حوادث (مثل قطع پردازش)، زمان‌بندی داده‌ها و هدایت کارها به موازات یکدیگر را در پردازش داده‌ها نیز باید مدنظر قرار دهد. در بسیاری از مواقع، هنگامی که داده‌های یکسانی در زمانی که سیستم در حالتی متفاوت قرار دارد نیز ممکن است منجر به خطا شود.

مثلاً، نرم‌افزار زمان واقعی که دستگاه فتوکپی جدیدی را کنترل می کند وقفه‌های عملیاتی را بدون خطا در هنگام کپی گرفتن، می‌پذیرد. این وقفه‌های عملیاتی یکسان مثلاً وقتی‌که کاغذ در دستگاه گیر کرده، باعث ایجاد یک کد تشخیص‌دهنده می‌شود که نشان‌دهنده محل گیر کردن کاغذ است (که یک خطاست).

علاوه بر آن، ارتباط درونی که بین نرم‌افزار زمان واقعی و محیط سخت‌افزاری‌اش وجود دارد، نیز می‌تواند باعث یک‌سری مشکلات آزمونی شود. آزمون نرم‌افزار باید تأثیر خطاهای سخت‌افزاری روی فرآیند نرم‌افزاری را مدنظر داشته باشد. چنین خطاهایی از نظر شبیه‌سازی واقع بینانه بسیار مشکل هستند.

روش‌های جامع در طراحی مورد آزمون باید هنوز تکمیل گردند. می‌توان یک راهبرد چهار مرحله‌ای را پیشنهاد نمود:

آزمون وظیفه. اولین مرحله در آزمون نرم‌افزار بدون وقفه عبارتست از آزمون هر وظیفه به‌طور مستقل، یعنی آزمون‌های جعبه سفید و سیاه طراحی شده و برای همان وظیفه به اجرا درمی‌آیند. هر وظیفه به‌طور مستقل در طول این آزمونها صورت می‌گیرد. آزمون وظیفه خطاهای منطقی و کارکردی را مشخص می‌سازد، اما خطاهای زمان‌بندی یا وظیفه را نشان می‌دهد.

1.review and inspection

2.live test

3.asynchronous

آزمون رفتاری. با استفاده از مدل‌های سیستمی که توسط ابزارهای CASE ایجاد شده‌اند، می‌توان کارکرد سیستم بدون وقفه را شبیه‌سازی نموده و عملکرد آن را در اثر حوادث خارجی بررسی نمود. این فعالیت‌های تحلیلی می‌توانند به‌عنوان پایه و اساس طراحی آزمونها مورد استفاده در هنگام ساخت نرم‌افزار عمل کنند. با استفاده از تکنیکی که مشابه تقسیم بندی هم ارزی است (بخش ۱۷-۶-۱)، حوادثی (مثل وقفه‌ها، علائم کنترلی) برای آزمون طبقه‌بندی می‌شوند. مثلاً اتفاقاتی که برای دستگاه فتوکپی ممکن است رخ دهند: وقفه‌های کاربر، وقفه‌های مکانیکی (مثل جمع شدن کاغذ)، وقفه‌های سیستمی (مثل کاهش تونر) و وضعیتهای خطا می‌باشند (مثل داغ شدن رولر)، هر کدام از این موارد به‌طور منفرد آزمون شده و حالت رفتاری سیستم قابل اجرا از نظر شناسایی این خطاها بررسی می‌شوند، یعنی خطاهایی که در نتیجه پردازش مربوط به این حوادث رخ می‌دهند.

رفتار مدل سیستمی (که در طول وظیفه تحلیل تولید شده) و نرم‌افزار قابل اجرا را می‌توان از نظر هماهنگی و انطباق مقایسه کرد. وقتی هر گروه از این حوادث مورد آزمون واقع شدند، آنها را به‌صورت تصادفی روی سیستم با تکرار وقوع تصادفی اجرا می‌کنند. رفتار نرم افزار آزمایش می شود تا خطاهای رفتاری کشف شوند.

آزمون بین وظائف. وقتی خطاهای هر وظیفه و خطاهای رفتار سیستم مجزا شدند، آزمون در مورد خطاهای مربوط به زمان صورت می‌گیرد. کارهای ناهمگام که از نظر برقراری ارتباط با یکدیگر شناسایی شده‌اند با پردازش و ارزیابی مختلف داده‌ها صورت می‌گیرند که نهایتاً منجر به این امر می‌شود که آیا خطاهای ناهمگام بین وظیفه ای رخ می‌دهند یا خیر. علاوه بر این، وظائفی که از طریق صف کردن پیام‌ها یا ذخیره داده‌ها ارتباط برقرار می‌کنند از نظر مشخص کردن خطاها در اندازه‌بندی حوزه‌های ذخیره داده‌ها آزمون می‌شوند.

آزمون سیستم. نرم افزار و سخت‌افزار با هم تلفیق می شوند و یک‌سری آزمونهای سیستم روی آنها صورت می‌گیرد تا خطاهای موجود در رابط نرم‌افزاری/ سخت‌افزاری مشخص شود.

بسیاری از فرآیندهای سیستم‌های زمان واقعی وقفه ها را پردازش می کند. بنابراین، آزمون رفع و رجوع این حوادث بولین ضروری است. با استفاده از دیاگرام انتقال حالت و مشخصات کنترل، آزمون‌گر می‌تواند فهرستی از همه وقفه‌های ممکن و پردازشی که در نتیجه وقفه رخ می‌دهد، ارائه دهد. سپس آزمونهایی طراحی می‌شوند که مشخصه‌های زیر را در مورد سیستم ارزیابی می‌کنند:

- آیا اولویت‌های وقفه به درستی تخصیص یافته و اجرا می‌شوند؟
- آیا پردازش هر وقفه به درستی اجرا می‌شود؟
- آیا علمکرد (مثل زمان پردازش) اجرای وقفه با نیازمندیهای آن هماهنگی دارد؟
- آیا مقدار زیادی از وقفه‌ها که در زمان‌های بحرانی ایجاد می‌شوند، در عملکرد یا کار سیستم مشکل ایجاد می‌کنند؟

علاوه بر این‌ها، حوزه عمومی داده‌هایی که برای انتقال اطلاعات به‌عنوان بخشی از پردازش وقفه استفاده می‌گردند، باید آزمون شود تا پتانسیل لازم برای تولید اثرات جانبی ارزیابی گردد.

۸-۱۷ خلاصه

هدف اولیه طراحی مورد آزمون عبارتست از بدست آوردن مجموعه‌ای از آزمون‌ها که دارای ضریب احتمال بالایی در مشخص کردن خطاهای نرم‌افزاری باشند. برای نیل به این هدف، دو نوع مورد آزمون مختلف از نظر تکنیکی استفاده می‌شوند: آزمون جعبه سفید و آزمون جعبه سیاه.

آزمون جعبه سفید روی ساختار کنترل برنامه تمرکز دارد. موارد آزمون برای اطمینان از این‌که همه جملات برنامه حداقل یکبار اجرا شده و همه حالات منطقی نیز در طول آزمون به اجرا درآمده‌اند، به‌کار گرفته می‌شود. آزمون مسیر پایه که یک تکنیک جعبه سفید است، از نمودارهای برنامه برای بدست آوردن مجموعه‌ای از آزمونهای مستقل خطی استفاده می‌کند که پوشش کامل برنامه را تضمین می‌کند. آزمون وضعیت و جریان داده‌ها منطق برنامه را به اجرا گذاشته و آزمون حلقه یا لوپ، دیگر فنون جعبه سفید را با مهیا کردن رویه‌ای برای اجرای حلقه‌هایی با درجات پیچیدگی مختلف، تکمیل می‌کند.

هتزل [HET84] آزمون جعبه سفید را یک آزمون در مقیاس کوچک معرفی می‌کند. نکته موردنظر او این است که، آزمونهای جعبه سفیدی که ما در این فصل در نظر گرفتیم معمولاً در جزءهای برنامه‌های کوچک استفاده می‌شوند (مثل پیمانه‌ها یا گروه‌های کوچکی از پیمانه‌ها). به‌عبارت دیگر، آزمون جعبه سیاه نقطه تمرکز ما را وسعت بخشیده و گاهی «آزمون در مقیاس بزرگ» نامیده می‌شود.

آزمونهای جعبه سیاه برای اعتبار بخشیدن به نیازمندیهای کارکردی بدون توجه به کارهای داخلی برنامه طراحی شده‌اند. فنون آزمون جعبه سیاه روی قلمرو اطلاعات نرم‌افزار، بدست آوردن موارد آزمونی با تقسیم‌بندی قلمرو ورودی و خروجی برنامه متمرکز می‌شوند بطوری‌که پوشش آزمون بصورت کامل باشد. تقسیم‌بندی هم ارزی قلمرو ورودی را به کلاسهایی از داده‌ها تقسیم می‌کند که احتمالاً کارکرد نرم‌افزاری خاصی را اجرا می‌کنند. تحلیل مقدار سرحد، توانایی برنامه را در اجرای داده‌ها در سرحدات قابل پذیرش ممکن می‌سازد. آزمون آرایه متعامد یک روش نظام مند و مؤثر برای آزمون سیستم‌هایی مهیا می‌کند که پارامترهای ورودی کمی دارند.

روش‌های تخصصی آزمون، در برگیرنده یک طیف وسیعی از توانایی‌های نرم‌افزار و حوزه‌های کاربردی است. آزمون رابطهای گرافیکی کاربر، ساختار خادم / مخدوم (C/S)، مستندسازی و تسهیلات کمکی و سیستم‌های بدون وقفه هر کدام نیازمند رهنمودها و فنون تخصصی است که برای آزمون نرم‌افزار لازمند.

تولیدکنندگان با تجربه نرم‌افزار اغلب بیان می‌دارند که «آزمون هرگز پایان نمی‌پذیرد، فقط از شما به مشتری منتقل می‌شود» هر زمان که مشتری شما از برنامه استفاده می‌کند، یک آزمون صورت می‌گیرد. با به‌کارگیری طراحی مورد آزمون، مهندس نرم‌افزار می‌تواند به یک آزمون کامل رسیده و بدین‌وسیله بیشترین میزان خطا را قبل از شروع آزمون توسط مشتری، پیدا و اصلاح نماید.

مسایل و نکاتی برای تفکر و تعمق بیشتر

۱۷-۱ مایرز [MYE79] برنامه زیر را به‌عنوان یک خودآزمایی جهت توانایی شما در مشخص کردن آزمون‌های مناسب استفاده می‌کند: یک برنامه‌ای سه عدد صحیح را می‌خواند. این سه مقدار بیانگر طول اضلاع یک مثلث خواهند بود. برنامه با چاپ پیامی معلوم می کند که این مثلث متساوی‌الاضلاع، متساوی‌الساقین، یا مختلف‌الاضلاع می باشد. مجموعه‌ای از موارد آزمون را توسعه دهید که گمان دارید این برنامه را به اندازه کافی مورد آزمایش قرار می‌دهند.

۱۷-۲ برنامه مشخص شده در سوال ۱ را طراحی و پیاده‌سازی کنید (با رفع و رجوع مناسب خطاها). یک گراف جریان برای برنامه رسم کنید و آزمون مسیرهای پایه را برای توسعهٔ موارد آزمون چنان به‌کار برید که آزمون تمام دستورات برنامه را تضمین کنند. موارد را اجرا نموده و نتیجه کار خود را نشان دهید.

۱۷-۳ آیا اهداف دیگری در خصوص آزمون به نظرتان می رسد که در بخش ۱۷-۱-۱ توضیح داده نشده اند؟

۱۷-۴ تکنیک مسیرهای پایه را برای تمام برنامه‌هایی که برای مسائل ۱۶-۴ تا ۱۶-۱۱ ساخته اید، به‌کار برید.

۱۷-۵ برای محاسبه پیچیدگی سیکلوماتیک یک زبان برنامه سازی دلخواه، یک ابزار نرم‌افزاری را مشخص، طراحی و پیاده‌سازی کنید. در طراحی خود، از ماتریس گراف به‌عنوان ساختمان داده های کاربردی استفاده کنید.

۱۷-۶ کتاب بیزر [BE195] را مطالعه نموده، تعیین کنید چگونه برنامه‌ای که در مسئله ۱۷-۵ ساخته‌اید، برای لحاظ کردن وزن پیوندهای گوناگون قابل توسعه است. ابزار خود را برای پردازش احتمالات اجرا یا زمان پردازش پیوند و اتصال، توسعه دهید.

۱۷-۷ رهیافت آزمون شرط را که در بخش ۱۷-۵-۱توضیح داده شده، برای طراحی مجموعه ای از موارد آزمون جهت برنامه‌ای که در مسئله ۱۷-۲ ساخته‌اید، استفاده کنید.

۱۷-۸ با استفاده از رهیافت آزمون جریان داده‌ها که در بخش ۱۷-۵-۲ توضیح داده شد، لیستی از زنجیره‌های تعریف ـ کاربرد را برای برنامه‌ای که در مسئله ۱۷-۲ ساخته‌اید، بسازید.

۱۷-۹ یک ابزار خودکار طراحی کنید که مطابق بخش ۱۷-۵-۳ حلقه‌ها را شناسایی و طبقه بندی نماید.

۱۷-۱۰ ابزار شرح داده شده در مسئله ۱۷-۹ را به گونه ای توسعه دهید که با رسیدن به هر حلقه، موارد آزمونی برای آن تولید کند. در این عملیات محاوره با آزمون کننده ضروری است.

۱۷-۱۱ حداقل سه مثال بیاورید که طی آنها آزمون جعبه سیاه پاسخ دهد : " همه چیز درست است"، درحالیکه آزمون‌های جعبه سفید احتمال حضور خطایی را هشدار دهند. حداقل سه مثال بیاورید

که در آنها آزمون جعبه سفید اعلام دارد " همه چیز درست است"، در صورتی که آزمون جعبه سیاه احتمال وجود خطایی را گوشزد کرده است.

۱۷-۱۲ آیا آزمون کل و جامع (حتی اگر برای برنامه‌های بسیار کوچک امکان‌پذیر باشد) صحت برنامه ۱۰۰ درصد برنامه را تضمین می نماید؟

۱۷-۱۳ با استفاده از روش تقسیم و افراز هم‌ارزی، یک مجموعه از موارد آزمون برای سیستم خانه امن که در فصلهای نخستین کتاب توضیح داده شده، به‌دست آورید.

۱۷-۱۴ با استفاده از تحلیل مقادیر مرزی، یک مجموعه از موارد آزمون برای سیستم PHTRS مسئله به‌دست آورید.

۱۷-۱۵ اندکی پژوهش کنید و مقاله کوتاهی بنویسید که مکانیزم تولید آرایه‌های متعامد (راستگوشه) را برای داده‌های آزمون توضیح دهد.

۱۷-۱۶ یک رابط کاربر گرافیکی GUI خاص برای برنامه‌ای که با آن آشنا هستید انتخاب و یک سری آزمون برای تمرین با این GUI طراحی کنید.

۱۷-۱۷ پژوهشی بر یک سیستم خادم / مخدوم که با آن آشنا هستید انجام داده . مجموعه‌ای از سناریوی کاربر را توسعه دهید و سپس یک نمودار عملیاتی برای سیستم تهیه کنید.

۱۷-۱۸ یک راهنمای کاربر (یا تسهیلات راهنما) را برای برنامه‌ای که با غالبا" مورد استفاده قرار می دهید، مورد آزمون قرار دهید و حداقل یک خطا در مستندات آن بیابید.

فهرست منابع و مراجع

[BEI90] Beizer, B., *Software Testing Techniques*, 2nd ed., Van Nostrand-Reinhold, 1990.

[BEI95] Beizer, B., *Black-Box Testing*, Wiley, 1995.

[BRI87] Brilliant, S.S., J.C. Knight, and N.G. Levenson, "The Consistent Comparison Problem in N-Version Software," *ACM Software Engineering Notes*, vol. 12, no. I, January 1987, pp. 29-34.

[DAV95] Davis, A., *201 Principles of software Development*, McGraw-Hill, 1995.

[DEU79] Deutsch, M., "Verification and Validation," in *Software Engineering* (R. Jensen and C. Tonies, eds.), Prentice-Hall, 1979, pp. 329-408.

[FOS84] Foster, K.A., "Sensitive Test Data for Boolean Expressions," *ACM Software Engineering Notes*, vol. 9, no. 2, April 1984, pp. 120--125.

[FRA88] Frankl, P.G. and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Engineering*, vol. SE-14, no. 10, October 1988, pp. 1483-1498.

[FRA93] Frankl, P.G. and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow," *IEEE Trans. Software Engineering*, vol. SE-19, no. 8,August 1993,pp. 770-787.

[HET84] Hetzel, W., *The Complete Guide to Software Testing*, QED Information Sciences, 1984.

[HOW82] Howden, W.E., "Weak Mutation Testing and the Completeness of Test Cases," *IEEE Trans. Software Engineering*, vol. SE-8, no. 4, July 1982, pp. 371-379.

[JON81] Jones, T.C., *Programming Productivity: Issues for the 80s*, IEEE Computer Society Press, 1981.

[KAN93] Kaner, C., J. Falk, and H.Q. Nguyen, *Testing Computer software*, 2nd ed., Van Nostrand-Reinhold, 1993.

[KNI89] Knight, J. and P. Ammann, "Testing Software Using Multiple Versions," Software Productivity Consortium, Report No. 89029N, Reston, VA, June 1989.

[MCC76] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering*, vol. SE-2, December 1976, pp. 308-320.

[MYE79] Myers, G., *The Art of Software Testing*, Wiley, 1979.

[NTA88] Ntafos, S.C., A Comparison of Some Structural Testing Strategies," *IEEE Trans. Software Engineering*, vol. SE-14, no. 6, June 1988, pp. 868-874.

[PHA89] Phadke, M.S., *Quality Engineering Using Robust Design*, Prentice-Hall, 1989.

[PHA97] Phadke, M.S., It Planning Efficient Software Tests," *Crosstalk*, vol. 10, no. 10, October 1997, pp. 11-15.

[TAI87] Tai, K.C. and H.K. Su, "Test Generation for Boolean Expressions," *Proc. COMPSAC* '87, October 1987, pp. 278-283.

[TAI89] Tai, K.C., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, no. 2, April 1989, pp. 58-61.

[WHI80] White, L.J. and E.I. Cohen, "A Domain Strategy for Program Testing," *IEEE Trans. Software Engineering*, vol. SE-6, no. 5, May 1980, pp. 247-257.

خواندنیهای دیگر و منابع اطلاعاتی

Software engineering presents both technical and management challenges. Books by Black *(Managing the Testing Process*, Microsoft Press, 1999); Dustin, Rashka, and Paul *(Test Process Improvement: Step-by-Step Guide to Structured Testing*, Addison-Wesley, 1999); Perry *(Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*, Dorset House, 1997); and Kit and Finzi *(Software Testing in the Real World: Improving the Process*, Addison-Wesley, 1995) address management and process issues.

A number of excellent books are now available for those readers who desire additional information on software testing technology. Kaner, Nguyen, and Falk *(Testing Computer Software*, Wiley, 1999); Hutcheson *(Software Testing Methods and Metrics: The Most Important Tests*, McGraw-Hill, 1997); Marick *(The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*, Prentice-Hall, 1995); Jorgensen *(Software Testing: A Craftsman's Approach*, CRC Press, 1995) present treatments of the subject that consider testing methods and strategies.

Myers [MYE 79] remains a classic text, covering black-box techniques in considerable detail. Beizer [BEI90] provides comprehensive coverage of white-box techniques, introducing a level of mathematical rigor that has often been missing in other treatments of testing. His later book [BEI95] presents a concise treatment of important methods. Perry *(Effective Methods for Software Testing*, Wiley-QED, 1995) and Friedman and Voas *(Software Assessment: Reliability, Safety, Testability*, Wiley, 1995) present good introductions to testing strategies and tactics. Mosley *(The Handbook of MIS Application Software Testing*, Prentice-Hall, 1993) discusses testing issues for large information systems, and Marks *(Testing Very Big Systems*, McGraw-Hill, 1992) discusses the special issues that must be considered when testing major programming systems.

Software testing is a resource-intensive activity. It is for this reason that many organizations automate parts of the testing process. Books by Dustin, Rashka, and Poston *(Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 1999) and Poston *(Automating Specification-Based Software Testing*, IEEE Computer Society, 1996) discuss tools, strategies, and methods for automated testing. An excellent source of information on automated tools for software testing is the *Testing Tools Reference Guide* (Software Quality Engineering, Jacksonville, FL, updated yearly). This directory contains descriptions of hundreds of testing tools, categorized by testing activity, hardware platform, and software support.

A number of books consider testing methods and strategies in specialized application areas. Gardiner *(Testing Safety-Related Software: A Practical Handbook*, Springer-Verlag, 1999) has edited a book that addresses testing of safety-critical systems. Mosley *(Client/Server Software Testing on the Desk Top and the Web*, Prentice-Hall, 1999) discusses the test process for clients, servers, and network components. Rubin *(Handbook of Usability Testing*, Wiley, 1994) has written a useful guide for those who must exercise human interfaces.

A wide variety of information sources on software testing and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to testing concepts, methods, and strategies can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/sepa/resources/test - techniques.mhtml