ارایه روشی برای تولید خودکار موارد تست با استفاده از Viewchart

مجتبی وطنی* ، بهروز ترک لادانی، ناصر نعمت بخش

*دانشجوی کارشناسی ارشد مهندسی نرم افزار دانشگاه اصفهان

عضو هیئت علمی گروه کامپیوتر دانشگاه اصفهان

عضو هیئت علمی گروه کامپیوتر دانشگاه اصفهان

E-mail:

nemat@eng.ui.ac.ir ، ladani@eng.ui.ac.ir ، m_vtn@yahoo.com *

چکیده

یکی از مهم‌ترین مسائلی که طراحان نرم‌افزار با آن روبرو هستند، تست نرم‌افزار است. تست نرم‌افزار با استفاده از ابزارهای پیش از کد نظیر مدل و مشخصات به توسعه‌دهندگان نرم‌افزار این امکان را می‌دهد که دنباله‌های تست را پیش از تولید کد و به موازات آن تولید کنند. به این روش تست، تست مبتنی بر مدل می‌گویند. روش‌های زیادی برای این منظور ارائه شده‌اند که از مدل‌هایی نظیر ماشین های حالت متناهی یا مدل‌های فرمال برای تولید خودکار موارد تست استفاده می‌کنند. بدنبال کاستی هایی نظیر تولید موارد تست طولانی و تولید موارد تست زائد، بر آن شدیم تا روش جدیدی برای این منظور ارائه دهیم. در این روش از مدل **Viewchart** استفاده کرده‌ایم. با استفاده از این روش به مجموعه‌ای از موارد تست کوتاه، اما جامع دست می‌یابیم که به ما این امکان را می‌دهند تا تست‌ها را به سرعت اجرا کنیم و خطاها را به سرعت ردیابی کنیم.

واژه‌های کلیدی: تست مبتنی بر مدل، Statechart ، Viewchart، تابع تست ، موارد تست

۱- مقدمه

هدف از تست نرم‌افزار نمایان کردن خطاهایی است که در طراحی و ساخت آن به طور ناخواسته به‌وجود آمده‌اند. یک استراتژی برای تست نرم‌افزار، متدهای طراحی موارد تست نرم‌افزار را به صورت دنباله ای از گام‌های خوش تعریف در می‌آورد که به تولید موفق نرم افزار می انجامد. مسئله مهم در انتخاب یک مورد تست، چگونگی تعیین یک دنباله از موارد تست است، به گونه‌ای که به طور موثر عملکرد مورد انتظار سیستم را نشان دهد. در ضمن، یک مورد تست باید کوتاه باشد و تا حد امکان بتواند تمام خطاهایی که ممکن است در سیستم رخ دهند را نشان دهد. روش‌های موجود انتخاب تست در برخورد با این دو مورد و میزان فرمال بودن سیستم، به گونه‌های متفاوت عمل می کنند و هدف اصلی را یکی از این موارد، انتخاب می‌کنند. در این مقاله روش جدیدی برای تست نرم‌افزار معرفی می شود کرد که کاستی‌های روش‌های قبلی نظیر طولانی بودن موارد تست را برطرف کند.

۲- مروری بر روش‌های تست نرم‌افزار

در کل استراتژی‌های تولید موارد تست را می‌توانیم به دو گروه تقسیم کنیم:

۱- براساس کد برنامه

۲- براساس ابزارهای پیش از کد نظیر نیازمندیها، مشخصات و مدل‌های طراحی

انتخاب موارد تست از روی کد برنامه، علاوه بر پیچیدگی ذاتی خود، مستلزم انتظار تا پایان کار است. و در کـل، دوره سـاخت و تحویل نرم‌افزار را افزایش می دهد. اما استفاده از ابزارهای پیش از کد این انتظار را حذف می‌کنند و می‌توان همگام با تولید کد، موارد تست را از نیازمندی‌ها استخراج کرد. اما در بین ابزارهای پیش از کد دو مورد اول یعنی نیازمندی‌ها و مشخصات، جزئی و غیر فرمال هستند و بـنابراین اسـتخراج مـوارد تسـت از آن‌ها مشـکل است و موارد تست استخراجی نیز طبعاً ناکامل هستند. از طرفی در طول فرآیند مهندسی نرم افزار این نیازمندی‌ها و مشخصات، مدل طراحی را شکل می دهند که در تولید کد نرم‌افزار استفاده می‌شوند. به این ترتیب مدل طراحی کاملترین ابزار پیش از کد است. انتخاب موارد تست از روی مدل سیستم، نه تنها به عمل تست ساختار می‌دهد، بلکه دیگر نیازی نیست تا منتظر تکمیل کد سیستم باشیم. ابتدا باید روشی را برای نمایش مدل انتخاب کنیم و سپس متدی را برای استخراج موارد تست از این مدل ایجاد کنیم.

با یک مدل فرمال و قابل تفسیر توسط ماشین، موارد تست را می‌توان به صورت خودکار استخراج کرد. مدل‌های سیستم را می توان از یک توصیف فرمال یا از ابزارهای دیاگرامی تولید کرد.

۲-۱ تکنیک‌های توصیف مدل رفتاری سیستم/کاربر:

تکنـیـک هـای توصـیف مـدل رفـتاری سیستم/کاربر را می توان به موارد زیر تقسیم بندی کرد:

۱- جدول‌های تصمیم: مجموعه‌ای از زوج‌های شرایط/ عمل هستند.

۲- ماشین های حالت متناهی

۳- گرامرها

۴- زنجیره‌های مارکوف: یک فرآیند گسسته و تصادفی که بودن فرآیند در یک زمان خاص در یک حالت خاص مستقیماً به حالت قبلی آن وابسته است.

۵- Statechart ها

جدول‌های تصمیم به خاطر موردی و غیرفرمال بودن ابزار مناسبی برای تست نیستند. گرامرها نحو برنامه نویسی را توصیف می‌کنند و بنابراین استخراج موارد تست از آنها پیچیدگی استخراج از کد برنامه را دارد و زنجیره‌های مارکوف هم با ساختار ریاضی، پیچیدگی خود را دارند. Statechart ها نمونه کاملتری از FSM ها هستند و بیشتر روش‌های تولید خودکار موارد تست از Statechart ها بهره می‌برند.

در این مقاله تمرکز بر روی تست مبتنی بر مدل است. تست مبتنی بر مدل به معنای استخراج موارد تست از مدلی است که رفتار سیستم را نمایش می‌دهد. مدل‌ها برای فهم، تعریف و توسعه سیستم‌ها به کار می‌روند.

تعدادی از روش‌های مبتنی بر مدل عبارتند از:

- W-Method
- WP-Method
- DS(Distinguishing Sequence)
- UIO(Unit Input Output)
- TT(Transition Tour)

۲-۲ مشکلات پیش رو در روش‌های موجود

روش‌های موجود از UML، FSM و Statechart یا ابزارهای مشابه با آن‌ها استفاده می‌کنند که همگی دو مشکل عمده دارند:

۱- با افـزایش و رشد خـطی سیستم، تعداد حالات به صورت نمایی رشد می‌کند. این رشد باعث افزایشی شدید در تعداد حالات در سیستم های بزرگ می شود و در نتیجه موارد تست پیچیده تر می‌شوند. با بهره گیری از Viewchart می‌توانیم این پیچیدگی را کمتر کنیم.

۲- فضای نام عمومی در Statechartها هیچگونه کنترلی در ارتباط با مکانیسم کنترل حوزه با اصطلاحاتی مانند اعلان، حوزه و تقید سیستم ندارد. مثلاً یک تغییر در سیستم ممکن است حالت سیستم را از یک دیدگاه تغییر دهد ولی از یک دیدگاه دیگر حالت سیستم همان حالت قبلی باقی

بماند. به این ترتیب می‌توان با استخراج موارد تست مرتبط با Viewها موارد تست را کوچکتر و گویاتر انتخاب کرد.

مشکلات Statechart، مشکلات تست مبتنی بر مدل نیز هستند.

پیچیدگی و افزونگی که با رشد سیستم ایجاد می شود، عمل استخراج موارد تست را با مشکل مواجه می کند، از طرفی به علت عدم وجود یک مکانیسم کنترل حوزه، موارد تست ایجاد شده حالت عمومی دارند. در حالی که با استفاده از Viewchart می‌توان با محدود کردن یک مورد تست به یک دیدگاه، از بزرگ شدن موارد تست جلوگیری کرد.

به این ترتیب به جای داشتن موارد تست بزرگ و پیچیده، مجموعه ای از موارد تست کوچک داریم که علاوه بر اجرای راحت تر آنها، ردیابی مشکلات کشف شده نیز سریعتر می شود. از دیگر مزایای این روش جزئی بودن مشخصات به کار رفته در Viewchart ها می‌باشد.

۳- مروری بر Viewchartها

بدنبال مشکلات موجود در Statechart، Viewchart (عیسی‌زاده-۱۹۹۶) شکل گرفت. Viewchart بر اساس Statechart و با افزودن مفهوم View به آن شکل گرفته است. مفهوم دیدگاه رفتاری از یک سیستم نرم افزاری به معنای رفتار سیستم از یک دید خاص می‌باشد. مثلاً از دید یک مشتری در یک شبکه برای یک نرم‌افزار شبکه.

در مرحله تعیین نیازمندی‌های سیستم، مهندس تعیین نیازمندی های نرم‌افزار با کاربران بالقوه سیستم مصاحبه می‌کند و انتظار هر کدام از سیستم را ثبت می‌کند و یا به عبارت دیگر هر کاربر مرتبط، دیدگاه خود از سیستم را بیان می کند. بنابراین، نتیجه این مصاحبه، دیدگاه رفتاری کاربر است. سپس این نتایج در قالب Viewchartها به نمایش در می آیند.

مفهوم مرکزی در Viewchart، View است. یک دیدگاه رفتاری در یک سیستم نرم‌افزاری، رفتار سیستم از یک نقطه خاص از دیدگاه است.

هر View از سه جزء تشکیل شده است:

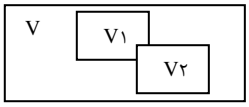
۱- Event در یک یا چند دیدگاه رخ می دهد. مانند Statechartها رویدادها ممکن است داخلی(متعلق به خود View) و یا خارجی باشند.

۲- Action : توسط یک دیدگاه تولید می شود.

۳- متغیرها: در یک دیدگاه تعریف می شوند و در سایر دیدگاه‌ها طبق قوانین حوزه تعریف اعتبار دارند.

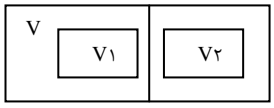
چهار ترکیب از دیدگاه ها در Viewchartها وجود دارد:

۱- Separate: تمام دیدگاه ها فعالند و هیچ انتقالی بین آنها صورت نمی گیرد. اجزاء یک دیدگاه از دید دیگری مخفی است. (شکل۱)



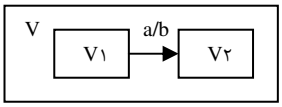
شکل ۱- دو دیدگاه مجزا V=SPT(V۱,V۲)

۲- And: مانند Separate است با این تفاوت که اجزاء یک دیدگاه از دید دیگری مخفی نیستند. (شکل۲)



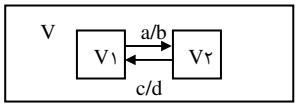
شکل ۲- دو دیدگاه And، V=AND(V۱,V۲)

۳- OR: مانند حالت And با این تفاوت که امکان انتقال بین آنها وجود دارد. (شکل۳)



شکل۳- $V=OR_{[a/b]}(V۱,V۲)$

۳-۱ OR دو طرفه: در شکل ۴ نشان داده شده است.



شکل٤- $V=OR_{[a/b]}(V۱,OR_{[c/d]}(V۲,V۱))$

۴- Hierarchical: ترکیب وراثتی برای دیدگاه ها از ترکیب دیدگاههای مختلف بوجود می آید. حالت های OR، AND و Seperate نیز وراثتی هستند.

۴- تولید دنباله‌های تست بر مبنای Viewchart

در این روش از تکنیک تقسیم و حل برای استخراج موارد تست استفاده می‌شود. به این ترتیب که ابتدا سیستم را بر اساس انواع کاربران، که در حقیقت دیدگاه های مختلف سیستم هستند در نظر می‌گیریم، پس از بررسی نیازمندیهای سیستم از دید هر کدام از این کاربران، نمودارهای Viewchart را برای هر کدام بدست می‌آوریم و سپس موارد تست را به موازات تولید کد برنامه برای هر کدام از این Viewchart ها استخراج می‌کنیم و درنهایت از کنار هم قرار دادن این موارد تست به مجموعه‌ای از موارد تست دست می‌یابیم که در صورت انجام درست کارها در مرحله تعیین نیازمندی‌ها می‌توانیم مطمئن باشیم که تمامی مسیرها و دیدگاه‌ها تست شده‌اند. مراحل کلی عبارتند از:

۱- شناسایی دیدگاه های موجود در سیستم

۲- توصیف مدل رفتاری سیستم با استفاده از Viewchart

۳- یافتن موارد تست از مدل سیستم

دو گام اول این فرآیند درگیر ایجاد مدل سیستم در قالب Viewchart می‌باشند. (برای اطلاعات بیشتر به [۱] مراجعه کنید) و گام سوم روش تولید موارد تست است.

این روش به روی یک دیدگاه اجرا می‌شود و حاصل آن مجموعه ای از موارد تست برای آن دیدگاه است. این روش از روش تقسیم و حل بدست آمده است. به این صورت که به ترتیبی که در ادامه می‌آید، درخت از راس به سمت برگ‌ها که همان Statechart ها هستند پیمایش می‌شود و سپس موارد تست برای Statechart محاسبه می‌شوند و دوباره به هم ملحق می‌شوند. این روش با استفاده از روشهای برنامه نویسی پویا اصلاح شده است تا سربار بازگشت آن حذف شود و الگوریتم آن عملی تر شود.

۴-۱ تعاریف پیشوندی برای یک Viewchart :

تعاریف پیشوندی بر پایه پیمایش Prefix درخت Viewchartها می‌باشند.

مثال: با استفاده از تعاریف پیشوندی، روابط شکل ۵ را بدست می آوریم:

Viewchart: V Views:V۱..V۸ States:A,B,C

Viewchart V=OR[x/β](V۱,Or[x/γ](V۲,V۱))

Viewchart V۲=SPT(V۳,V٤)

Viewchart V۳=AND(V۷,V۸) V۳ ≈ V٤

Viewchart V۸=SPT(V٥,V٦)

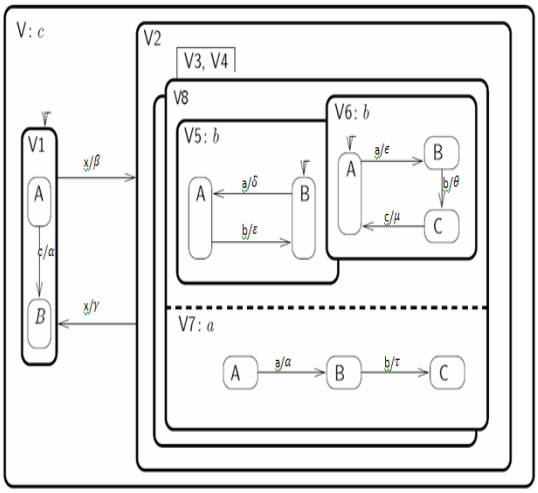
Statechart V۱=OR[c/α](A,B)

Statechart V۷=OR[a/ α](A,OR[b/ϯ](B,C))

Statechart V٥=OR[a/δ](B,OR[b/ε](A,B))

Statechart V٦=OR[a/Є](A,OR[b/Θ](B,

OR[c/μ](C,A)))



شکل۵- مثالی از یک Viewchart

۴-۲ تابع Ach() (تابع دسترسی پذیری)

با استفاده از تابع دسترسی پذیری(Achieve) که به اختصار آنرا با Ach نشان می دهیم می توان از دیدگاه اصلی به دیدگاه مورد نظر رسید. دنباله دسترسی پذیری برای یک دیدگاه V عبارت است از دنباله ای از ورودیهای

مجاز سیستم که با رخ دادن آن‌ها سیستم به دیدگاه V وارد می‌شود. ایـن دنبـاله به صورت Ach(V) نشان داده می شود.

الگوریتم زیر با استفاده از تعاریف پیشوندی Viewchart، Ach(V) را محاسبه می‌کند:

با شروع از تعریف شماره یک تا انتهای تعاریف، هر کدام از تعاریف را بررسی می‌کنیم که آیا دیدگاه V در سمت راست آن‌ها وجود دارد یا خیر. هر بار که دیدگاه V در سمت راست یک تعریف ظاهر می‌شود یکی از۳ حالت زیر را خواهد داشت که متناسب با هر کدام، Ach(V) محـاسبه می شود:

۱- V=And(V۱,V۲)=> Ach(V۱)=Ach(V۲)=Ach(V)

۲- V=SPT(V۱,V۲) => Ach(V۱)=Ach(V۲)=Ach(V)

۳ -V=OR[a/b](V۱,V۲) => Ach(V۱)=Ach(V) ,
Ach(V۲)=Ach(V)[a/b]

برای دیدگاه اصلی که آنرا V_{Head} می نامیم:

$Ach(V_{Head})$=Reset

به این معنا که سیستم در هر حالتی که باشد به حالت آغازین باز می‌گردد. در اینجا به عنوان نمونه Ach(V۷) را برای شکل ۵ محاسبه می‌کنیم:

Step۱: V۳ = And(V۷,V۸) => Ach(V۷) =Ach(V۳)

Step۲ : V۲ = SPT(V۳,V٤) => Ach(V۳) =Ach(V۲)

Step۳: V = OR[x/β](V۱,OR[x/γ](V۲,V۱)) =>

Ach(V۲) = Ach(OR[x/ γ](V۲,V۱)) = Ach(V).[x/ β]

Step٤: Ach(V) = Reset => Ach(V۲) = Reset[x/ β]

Step۱..٤ => Ach(V۷)=Reset.[x/ β]

۴-۳ عملگرهای || و#:

منظور از عمل || اجرای موازی دو مورد تست برای دو دیدگاه And می باشد. در حالت And به دلیل شفاف بودن متغیرهای خصوصی دو دیدگاه برای همدیگر، امکان تداخل در زمان تست وجود دارد که نیاز به دقت عمل زیادی دارد. بروز خطا در این عمل می‌تواند ناشی از سه حالت زیر باشد:

۱- وجود خطا در یکی از دو دیدگاه (با اجرای مجزا و مجرد هر کدام از دیدگاه‌ها می‌توانیم آن را بررسی کنیم)

۲- وجود خطا در اثر تداخل در متغیرهای خصوصی

۳- وجود خطا در اثر عدم رعایت قواعد همروندی در ارتباط با متغیرهای عمومی

تمایز بین موارد ۲ و ۳ در تمام موارد امکان‌پذیر نیست. اما از مقادیر متغیرها می‌توان به عنوان فاکتوری برای تشخیص آن استفاده کرد که در این صورت باید تغییرات در پیکربندی کامل سیستم را هم در هنگام تولید موارد تست، بررسی کنیم.

عملگر # مشابه عمل || است، منتها به دلیل عدم امکان تداخل در متغیرهای خصوصی، اجرایی به مراتب ساده تر دارد. در این عمل، با وجودی که تداخلی در متغیرهای خصوصی وجود ندارد، اما در مورد متغیرهای عمومی یعنی متغیرهایی که در رابطه وراثتی به ارث می‌رسند امکان تداخل وجود دارد. تنها موضوعی که در این رابطه باید رعایت شود، مربوط به امکان همروندی و ناسازگاری حاصل از آن است. در این مرحله از تست می‌توان نقصان فرآیند کنترل همروندی در برنامه را نیز تشخیص داد. پس در دو گام می‌توان به این هدف رسید:

گام اول: هر کدام از دیدگاه‌ها به طور مجزا و در غیاب دیگری تست شود.

گام دوم: دیدگاه‌ها به طور موازی تست شوند.

در نتیجه در صورت اجرای بی‌خطای گام اول، از بروز خطا در گام دوم می‌توانیم به این نتیجه برسیم که خطایی در کنترل همروندی وجود دارد.

۴-۴ تابع Test()

با استفاده از تعاریف پیشوندی Viewchart، موارد تست را برای دیدگاه V به صورت زیر بدست می‌آوریم:

الف- اگر V در تعاریف یک Viewchart باشد:

۱- اگر V به صورت V=AND(V۱,V۲) تعریف شده باشد، آنگاه :

Test (V) =Test (V۱) || Test (V۲)

۲- اگر V به صورت V=SPT(V۱,V۲) تعریف شده باشد، آنگاه :

Test (V) =Test (V۱) # Test (V۲)

اگر V به صورت V=OR[a/b](V۱,V۲) تعریف شده باشد، آنگاه :

Test (V) =Test (V۱).[a/b].Test (V۲)

ب- اگر V یک Statechart باشد:

۱- می‌توان الگوریتم را مانند Viewchart برای Statechart به کاربرد و تا مادامی که V یک State نباشد ادامه داد.

۲- می‌توان از یکی از روش‌های موجود تولید تست خودکار مانند W-Method، برای تولید موارد تست برای V استفاده کرد. به عنوان نمونه Test(V۲) را برای مثال بالا محاسبه می کنیم:

Viewchart V۲ = SPT(V۳,V٤) =>

Test(V۲) = Test(V۳) # Test(V٤)

Viewchart V۳ = AND(V۷,V۸) =>

Test(V۳) = Test(V۷) || Test(V۸)

Viewchart V۸ = SPT (V۵,V٦) =>

Test(V۸) = Test(V۵) # Test(V٦)

Statechart V۷=OR[a/ α](A,OR[b/ד](B,C)) =>

Test(V۷) = Test(A).[a/α].Test(OR[b/ד] (B,C)) =>

Test(V۷) = [a/α].Test(B).[b/ד].Test(C) =>

Test(V۷) = [a/α].[b/ד]

Statechart V۵=OR[a/δ] (B,OR[b/ε](A,B)) =>

Test(V۵) = Test(B). [a/δ].Test(OR[b/ε](A,B)) =>

Test(V۵) = [a/δ].Test(A).[b/ε].Test(B)=>

Test(V۵) = [a/δ]. [b/ε]

Statechart V٦=OR[a/Є](A,OR[b/Θ](B,

OR[c/μ](C,A))) =>

Test(V٦) = [a/Є]. [b/Θ]. [c/μ]

Test(V۸) = [a/δ]. [b/ε] # [a/Є]. [b/Θ]. [c/μ]

Test(V۳) = [a/α].[b/ד] || ([a/δ]. [b/ε] # [a/Є].

[b/Θ]. [c/μ]) , Test(V٤) = Test(V۳)

Test(V۲) = ([a/α].[b/ד] || ([a/δ]. [b/ε] # [a/Є].

[b/Θ]. [c/μ])) # ([a/α].[b/ד] || ([a/δ]. [b/ε] # [a/Є].

[b/Θ]. [c/μ]))

۵- الگوریتم نهایی

الگوریتم محاسبه موارد تست برای یک Viewchart:

فرض کنیم تعاریف پیشوندی V Viewchart را با N دیدگاه داریم. در اینجا الگوریتمی برای تولید یک سوئیت تست(مجموعه ای از موارد تست) ارائه می‌دهیم. این الگوریتم در دو گام سوئیت تست را تولید می‌کند. در این الگوریتم از دو تابع Ach() و Test() برای تولید موارد تست استفاده می‌کنیم:

۱- برای تمام Vi ها (i=۱ ... N)، Ach(Vi) را بدست می آوریم.

۲- برای تمام Vi ها (i=۱ ... N)، Test(Vi) را محاسبه می کنیم.

۳- برای تمام Viها (i=۱ ... N)، موارد تست را که با TC (Test Case) نشان می دهیم از فرمول زیر بدست می آوریم.

TC(Vi) = Ach(Vi).Test(Vi) را محاسبه می‌کنیم.

مجموعه تمام موردهای تست که در این الگوریتم بدست می‌آیند یک سوئیت تست جامع و کامل برای Viewchart می باشد.

به عنوان مثال TC(V۲) را محاسبه می‌کنیم:

TC(V۲)=Ach(V۲).Test(V۲) =>

TC(V۲)=Reset.[x/β].(([a/α].[b/ד] || ([a/δ]. [b/ε] # [a/Є]. [b/Θ]. [c/μ])) # ([a/α].[b/ד] || ([a/δ]. [b/ε] # [a/Є]. [b/Θ]. [c/μ]))) =

((Reset.[x/ β].[a/ α].[b/ד] || (Reset.[x/β].[a/δ].[b/ε] # Reset.[x/β].[a/Є]. [b/Θ].[c/μ])) # (Reset.[x/β]. [a/α].[b/ד] || (Reset.[x/β].[a/δ]. [b/ε] # Reset.[x/β].[a/Є].[b/Θ].[c/μ])))

به این ترتیب با محاسبه زیر می‌توان یک سوئیت تست برای این Viewchart تهیه کنیم:

Step۱: Compute {Ach(V۱)…Ach(V۸)} As "Achievability Set"

Step ۲: Compute {Test(V۱)..Test(V۸)} As "TestCases Set"

Step۳: Compute {TC(V۱).. TC(V۸)} ,$TC(V_i)=Ach(V_i).Test(V_i)$ As "Final Test Suit"

می‌توان نشان داد که این سوئیت تست کاملا جامع است و تمام مسیرها را تست می‌کند. همانطور که پیش از این نیز به آن اشاره شد، این روش از پیمایش پیشوندی درخت سیستم به دست می‌آید و بنابراین تمام گره‌های درخت در این فرآیند در نظر گرفته می‌شوند.

گره های درخت، مجموعه تمام حالات، دیدگاه‌ها و روابط(هر ۴ نوع ممکن) هستند و تست کامل گره‌ها یعنی تست کامل سیستم و در نهایت تست تمام حالات، تمام دیدگاه‌ها و تمام مسیرهای مدل سیستم.

۶- مزایای تولید تست از Viewchart

به این ترتیب با تولید موارد تست بر اساس Viewchart، به دنباله‌های تستی دست می‌یابیم که دارای خواص زیر هستند:

۱- تولید موارد تست جامع و کامل:

همانطور که در روش بالا دیدید موارد تست تولیدی تمام دیدگاه‌های موجود را پوشش می‌دهند، و از طرفی در مقیاسی بالاتر، تمام ترکیبات آن‌ها را نیز پوشش می‌دهند.

۲- کم کردن پیچیدگی های تولید موارد تست:

با این روش به جای آن که با ماشین حالت بزرگی روبرو باشیم، با ماشین‌های حالت کوچکتر روبرو هستیم، که پیچیدگی تولید موارد تست از آن‌ها، بسیار کمتر از ماشین حالت بزرگ است.

۳- یافتن مجموعه ای از موارد تست کوچک به جای تعدادی موارد تست بزرگ:

با بهره‌گیری از این روش، ما مجموعه موارد تست کوچکی داریم که اجرای هر کدام زمان زیادی نمی‌گیرد و از طرفی به دلیل مستقل بودن آن‌ها می‌توان آن‌ها را به طور مجزا و همزمان اجرا کرد.

۴- کاهش دوره ساخت و تحویل نرم افزار:

با اجرای موازی سوئیت‌های تست، مرحله اجرای تست نرم افزار کوتاه‌تر می‌شود. از طرفی به دلیل قابلیت خودکار شدن تولید دنباله‌های تست از Viewchart، تولید آن‌ها نیز زمان کمتری می‌برد.

۵- تسهیل امر خطایابی و تعمیر نرم‌افزار:

به دنبال تولید دنباله های تست از دیدگاه‌ها و اجرای آن‌ها، در صورت بروز خطا در اجرا، به صورت ساده‌تر و سریع‌تری می‌توانیم خطاهای موجود را برطرف کنیم. به دو دلیل: یکی کوچکتر بودن دنباله‌های تست و دوم محدود بودن آن‌ها به یک دیدگاه.

۶- یافتن خطاهای بیشتر و هدفمند:

همانطور که نشان داده شد با اجراهای مختلف موارد تست تولیدی می‌توانیم خطاها را به صورت هدفمند پیدا کنیم، مانند آنچه در مورد "کنترل همروندی" دیدیم که با دو اجرای مجزا و یک اجرای موازی می‌توان ۳ نوع خطا را کشف کرد.

۷- نتیجه‌گیری:

در پروژه‌های تجاری نمی‌توانیم تصور کنیم که همواره مدل کاملی از سیستم وجود دارد. تنها فرض قابل قبول داشتن نیازمندی‌های نیمه فرمال است مانند آنچه در UML است. از طرفی باید موارد تست را هر چه زودتر و حتی زمانی که سیستم به صورت جزئی مدل شده است، تهیه کنیم. برای برآورده کردن این نیازها روش‌هایی پیشنهاد شده‌اند مانند UIT (Use Interaction Test) که از اجماع بین روش‌های موجود به جواب می‌رسد که بسیار وقت‌گیر است و هزینه زیادی دارد. اما روشی که با استفاده از Viewchartها ارائه می‌شود، روش واحدی است که هزینه و زمان کمتری می‌برد.

با این روش پس از مرحله تحلیل نیازمندی‌ها، سیستم به صورت دیاگرام‌های Viewchart مدل می‌شود و مدل ایجاد شده به دو گروه تولیدکنندگان کد و تولیدکنندگان موارد تست تحویل داده می‌شود. این دو گروه به صورت موازی مشغول به کار می‌شوند. با پیشرفت تولید کد، موارد تست متناسب تولید می‌شوند و این بخش ها تست می‌شوند. این ویژگی از جزئی(جزء به جزء) بودن نمودارهای Viewchart سرچشمه می‌گیرد. به این ترتیب سیستم به صورت پی در پی توسط موارد تست کوچک تست می‌شود.

با استفاده از این روش تولید موارد تست، ما به دنباله‌ای از موارد تست دست می‌یابیم که کوتاه و جامع هستند. به این ترتیب اجرای دنباله‌های تست زمان زیادی نمی‌برد و از طرفی به دلیل کوتاهی آن‌ها و وابستگی آن‌ها به بخش خاصی از برنامه، به راحتی منبع خطا یافت می‌شود.

مراجع

[۱] Ayaz Isazadeh, Behavioral Views for Software Requirements Engineering, PHD thesis, Queen's University , September ۱۹۹٦.

[۲] Ayaz Isazadeh,David A_ Lamb Glenn, H_ MacEwen, The Semantics of Viewcharts, External Technical Report, ISSN-۰۸۳٦-۰۲۲۷, ۹۵-۳۹۵ ,۱۹۹۵.

[۳] Larry Apfelbaum, John Doyle, Model Based Testing, software quality week Conference, New Hampshire, USA, May, ۱۹۹۷

[٤] Brian Berger, Majdi Abuelbassal, and Mohammad Hossain, Model Driven Testing, DNA Enterprises Inc,Teradyne software and system test, http://www.teradyne.com/prods/sst/product_center/sstprod.html, March ۱۹۹۷.

[٥] MODEL-BASED TESTING, DACS Gold Practice Document Series, GP-۳٤ V ۱,۱, ۲۰۰۳.

[٦] M.Prasanna, S.N. Sivanandam, R.Venkatesan and R.Sundarrajan, A SURVEY ON AUTOMATIC TEST CASE GENERATION, Academic Open Internet Journal , Volume ۱۵, ۲۰۰۵.

[۷] Axel Belinfante, Lars Frantzen , and Christian Schallhart, tools for test case generation, dagstuhl research seminar, January, ۲۰۰٤.

[۸] Johannes Ryser, Martin Glinz, A Scenario-Based Approach to Validating and Testing Software Systems Using statecharts, ۱۲th International Conference on Software and Systems Engineering and their Applications ICSSEA'۹۹. Proceedings: CNAM, Paris, France, ۱۹۹۹.

[۹] Alexander Pretschner, Classical search strategies for test case generation,

[۱۰] Stefania Gnesi, Diego Latella and Mieke Massink, Formal Test-case Generation for UML Statecharts, international conference on engineering of complex computer systems, ۲۰۰٤.

[۱۱] A. Bertolino and E. Marchetti, Introducing a Reasonably Complete and Coherent Approach for Model-based Testing, Electronic Notes in Theoretical Computer Science, www.elsevier.nl/locate/entcs, ۲۰۰۳.

[۱۲] Henry Muccini, Software Architecture for Testing, Coordination and Views Model Checking, phd thesis, roma university, ۲۰۰۲.