****

**تولید مورد تست جهش رانده شده با استفاده از جهش همزمان کوتاه مدت**

**چکیده**

در زمینه تست جعبه سیاه، موارد تست[[1]](#footnote-1) تولید از طریق مدل کردن جهش[[2]](#footnote-2)، برای تولید مجموعه تست قدرتمند شناخته شده است، اما معمولا مشکل گران بودن را دارد. این مقاله یک نسخه جدید از ابزار MoMuT::UML را ارائه می­دهد، که یک نسخه مقیاس پذیر از تولید مورد تست جهش رانده شده[[3]](#footnote-3) (MDTCG) را پیاده سازی می­کند. آن قادر به کنترل مدل­های UML با اندازه صنعتی شامل شبکه­هایی از قبیل 2800 تعامل دستگاه­های دولتی است. برای رسیدن به مقیاس­پذیری مورد نیاز، الگوریتم پیاده­سازی شده همزمانی را در MDTCG بکار می­برد و آن را با یک استراتژی تولید مبتنی بر جستجو ترکیب می­کند. برای ارزیابی، ما هفت مورد از حوزه های کاربردی مختلف با افزایش سطح دشواری را استفاده می­کنیم، توقف در یک مدل از یک ایستگاه راه­آهن در شبکه راه­آهن اتریش.

**1. مقدمه**

برای سیستم­های انفعالی، به ویژه در حوزه حیاتی ایمنی و در مواردی که بروزرسانی­ها گران قیمت هستند، تست مناسب الزامی است. در حوزه پیشین، استانداردهای ایمنی در واقع نیازمند انواع خاصی از تست برای انجام شدن بود و بعضی استانداردها بسیار استفاده از روش­های رسمی و تست مبتنی بر مدل را توصیه می­کند. تولید مورد تست جهش رانده شده (MDTCG)، که یک شکل مبتنی بر نقص[[4]](#footnote-4) است [13]، می­تواند در رضایت بخشی نیازمندی­های مطرح شده توسط استانداردها کمک کند – اگر آن به اندازه کافی با مدل­های با اندازه صنعتی کار می­کرد.

در این کار، ما توقف مرزهای آنچه را می­تواند با MDTCG انجام شود با پرداختن به مسائل مربوط به عملکرد ناشی از ماهیت خاص از مدل مبتنی بر مسئله تست جهش را مورد خطاب قرار می­دهیم. برای این کار، ما از محل جهش برای اکتشاف خود بهره می­بریم و از اکتشاف مبتنی بر جهش برای هدایت یک روش مبتنی بر جستجو در سرتاسر طرح­های جهش استفاده می­کنیم. به عنوان یک معیار، ما از هفت مدل مختلف UML با اندازه صنعتی استفاده می­کنیم. توجه داشته باشید که ما نمی­توانیم بیشتر از این با نسخه­های قبلی مان از MoMuT::UML بکار ببریم [1]. مدل هم توسط مهندسان صنعتی در شرکت­های همکار و یا در همکاری نزدیک با آنها ساخته شده است. کوچکترین مدل، رفتار یک سیستم دزدگیر ماشین معمولی را توصیف می­کند، در حالی که بزرگترین مدل، مدل یک ایستگاه راه­آهن با اندازه متوسط در شبکه راه­آهن ملی اتریش است. مدل دارای ویژگی­های مختلف است: برخی روش­های نمادین مطلوب، برخی شمارشی­های مطلوب، برخی از آنهایی که بسیار همزمان هستند، برخی به صورت سریال­شده هستند، برخی از زمان گسسته استفاده می­کنند، در حالی که دیگران استفاده نمی­کنند.

جدول 1: سه تولید از MoMuT::UML

|  |  |  |
| --- | --- | --- |
| **تولید کننده مورد تست** | **جهش دهنده** |  |
| اجرا | اکتشاف | تشخیص | جستجو | فشرده­سازی | سطح | جهش | موتور |
| مفسر | شمارشی | S | BF | - | UML | F | Gen1 |
| SMT | نمادین | S,W | Re | - | UML | F | Gen2 |
| گردآوردن | EPOR | S,W | GR | طرح | OOAS | F,H | Gen3 |

F: اولین سفارش جهش

H: سفارش جهش برتر

OOAS: سیستم عمل شیء گرا

GR: هدایت تصادفی

BF: جستجو اول عرض

Re: بررسی پالایش نمادین + قابل دسترسی

S: جهش قوی + IOCO

W: جهش ضعیف

EPOR: کاهش سفارش جزئی شمارشی

SMT: نظریه ارضا پیمانه

جدول 1 یک مرور کلی از سه تولید MoMuT::UML و تفاوت آنها را نشان می­دهد. تولید اول حدود سال 2010 ظاهر شد و بر اساس یک چک کننده متابعت ورودی خروجی شمارشی مبتنی بر پرولوگ (ioco) [17]، به نام ulisses بود، که با یک موتور جهش UML جداگانه جفت می­شود. تولید یک اثبات کرد به اندازه کافی قدرتمند است برای کنترل مدل­های ساده به پیچیده [3]. تولید دوم MoMuT::UML روش اکتشاف را از شمارش تفسیر به اکتشاف مبتنی بر حل کننده SMT تغییر داد. اگرچه پایان اثبات مبتنی بر SMT باعث کارآمدتر شدن نسبت به موتور تولید اول می­شود و یک تاثیر عملکرد بزرگ با یک نوع خاص از مدل ایجاد می­کند، نبود لیست پشتیبانی آن و متغیر شیء به این معنا که آن نمی­تواند مدل­هایی را که ما در نهایت برای هدف قرار دادیم بکار برد. ما آن را به طور موفقیت آمیز برای یک مورد بکار بردیم که دوباره در این مقاله ارائه می­شود. مقاله [2] خلاصه زیر از یافته­های ما هنگام استفاده از تولید دوم MoMuT::UML را ارائه می­دهد:

ارقام در زمان محاسبه ثابت می­کند که ما بیش از 60% از کل زمان TCG چک کردن معادل مدل جهش. ... داده­ها همچنین نشان می­دهند که موتور جهش ما نیاز به بهبود یافتن برای تولید جهش معنی­دارتر است ... 80% از جهش مدل توسط موارد تست تولید شده برای 2% از آنها پوشیده شده است.

در آن زمان، مجموع مدت زمان TCG در بیش از 44 ساعت برای موارد تست با یک عمق 25 تداخل تولید شده از یک مدل UML شامل یک نمونه، و 64 ویژگی بود. هدف ما این بود که مدل­ها را در بیش از 2000 نمونه، 3000 ویژگی رسیدگی کنیم و موارد تست بسیار عمیق­تر در زمان کمتر تولید کنیم. بنابراین، برای MoMuT::UML 3.0، ما از چک کردن همزمانی رسمی به یک استراتژی اکتشاف مبتنی بر جستجو با جهش کوتاه مدت تغییر یافتیم که در [3] معرفی شده است:



شکل 1: معماری اصلی MoMuT::UML 3.x

... نویسندگان برای ترکیب اکتشاف تصادفی جهت­دار و تولید مورد تست مبتنی بر جهش بر روی پرواز تلاش می­کنند. ایده این است که در طول اکتشاف از سیستم، جهش­هایی که به وضعیت فعلی نزدیک هستند، به صورت پویا قرار داده شوند و همزمانی بررسی شود. ...

سهم این مقاله سه مرحله است. اول، ما تکنیک­های استفاده شده در آخرین نسخه از MoMuT::UML را شرح می­دهیم که به ما اجازه بالا بردن مقیاس MDTCG برای اندازه مدل صنعتی مربوطه را می­دهد. دوم ما چهار مورد گرفته شده از صنعت اولین و سومین بار را ارائه می­دهیم. ما رویکردمان را بر طبق سؤالات پژوهشی زیر ارزیابی می­کنیم. سؤال پژوهشی 1: آیا ما می­توانیم MDTCG برای مدل­های صنعتی شده در هزینه (زمان و منابع) که کاربر مایل به قبول کردن باشد را استفاده کنیم؟ سؤال پژوهشی 2: آیا ما پوشش نقص کافی برای ابزار برای مفید شدن را حفظ می­کنیم؟ سؤال پژوهشی 3: با توجه به دو روش هدایت اکتشافی مختلف، آیا هر دو روش به یک اندازه مناسب است؟

مقاله به صورت زیر مرتب شده است. بخش 2 مسئله تولید مورد تست مبتنی بر جهش را معرفی می­کند، اشاره می­کند به مشکلات اصلی مرتبط با آن، یک لیست از عملگرهای جهش مورد استفاده برای آزمایش­های معروف را فراهم می­کند، و یک مقدمه کوتاه از زبان مدل­سازی سیستم­های عمل شیء گرا را ارائه می­دهد. بخش 3 معماری اصلی سیستم MoMuT::UML را ارائه می­دهد، موتور جهش را با جزئیات مورد بحث قرار می­دهد، و الگوریتم جهش کوتاه مدت را معرفی می­کند. بخش 4 هفت مورد مطالعه را ارائه می­دهد و توسط یک ارزیابی در بخش 5 دنبال می­شود. مقاله با بحث پیرامون کارهای مرتبط در بخش 6 و نتیجه­گیری در بخش 7 پایان می­یابد.

**2. تولید تست­ها از مدل جهش**

تولید مورد تست جهش رانده شده برای تولید به طور خودکار موارد تست تلاش می­کند که قادر به تشخیص نقص ("جهش یافته") نسخه هایی از یک مشخصه داده شده ("مدل") هستند. تحقیقات نشان می­دهند که این روش قدرتمند است و می­تواند دیگر معیارهای پوشش را شامل شود، مانند پوشش شرایط [15]، با توجه به مجموعه صحیح از عملگرهای جهش. علاوه بر نیاز به یک مدل، اشکال اصلی MDTCG پیچیدگی محاسباتی در یافته­های داده تست است که قادر به آشکار ساختن یک مدل نقص ("قابل جهش") است. این چالش به راحتی به عنوان یک مسئله مدل چک کردن معین می­شود. با این حال، با استفاده از مدل چک کننده برای تست کردن تولید (نسل)، مسائل خاص خود را دارد [6]، مانند، مدل­های بزرگ و غیر قطعی.

علاوه بر پیچیدگی کلی محاسباتی مسئله، موضوع دیگری که باعث افزایش بیشتر هزینه محاسباتی می­شود، جهش معادل است. جهش معادل جهشی است که هیچ تفاوتی در رفتار را نشان نمی­دهد. به عبارت دیگر، آنها معادل مدل اصلی هستند. این جهش­ها، که نمی­توانند به یک مورد تست منجر شوند، حداکثر جریمه را برای روش­های TCG مبتنی بر اکتشاف جامع تحمیل می­کنند. برای مثال، در نسخه قبلی MoMuT::UML، ما 60% از زمان کلی محاسبات در چک کردن جهش معادل صرف کردیم.

جدول 2: عملگرهای جهش OOAS

|  |  |
| --- | --- |
| **عملگر جهش** | **تعداد جهش، کران بالا** |
| غیر فعال کردن فرمان محفوظ O(g) | g. . . # guarded commands |
| جایگزینی عملگر دودویی O(b) | b… # of ≥,>,≤,<,=,≠,+,-,\*,div,mod,and,or, |
| جایگزینی عملگر یگانی O(u) | u. . . # of -,+,abs,not, |
| معکوس کردن بولین O(x) | x. . . # of true/false in rhs of assignments |
| جایگزینی عدد صحیح O(x) | x. . . # of literals in rhs of assignments |

به عنوان یک ابزار TCG جعبه سیاه، MoMuT::UML نیاز به یک محیط تست جهش قوی دارد، که بدان معنی است که آن تنها می­تواند شامل یک جهش شناسایی ("خاتمه­یافته[[5]](#footnote-5)") باشد، اگر جهش یک تفاوت در رفتار قابل مشاهده نسبت به اصلی نشان دهد. رابطه انطباق که ما برای تست استفاده می­کنیم، انطباق ورودی-خروجی Tretman است [17]، که در سیستم­های انتقال برچسب شده تعریف می­شود و گفته می­شود که هر گونه مشاهده که بتواند از سیستم تحت تست (SUT) بعد از یک اثر خاص ایجاد شود، باید توسط مدل پیش­بینی شود. توجه داشته باشید که اگر SUT فقط یک زیرمجموعه از مشاهدات ممکن تولید کند، آن هنوز خوب است. برای تعریف رسمی ioco و LTS، ما به Tretman [17] مراجعه می­کنیم.

شکل 1 یک مرور کلی از معماری MoMuT را نشان می­دهد. ابزار به عنوان فایل .jar توزیع شده و یک پایان جلو[[6]](#footnote-6)، نوشته شده در جاوا، با یک پایان عقب[[7]](#footnote-7) را ترکیب می­کند که در C++11 نوشته شده است. در حالی که پایان جلو برای چک کردن ورودی معتبر است، پایان عقب کار واقعی را انجام می­دهد. توجه داشته باشید که پایان جلو و عقب توسط بافرهای پروتکل گوگل از هم جدا می­شوند، بنابراین آن ممکن است از پایان عقب به طور مستقل استفاده کند. به عنوان ورودی، MoMuT::UML ممکن است از نمودارهای حالت UML یا سیستم­های عمل شیء گرا (OOAS) استفاده کند. اگر یک مدل UML – شامل نمودار حالت، نمودار کلاس، و نمودار نمونه – استفاده شود، MoMuT ابتدا آن را به کد OOAS ترجمه می­کند [10]. مرحله بعد اضافه کردن جهش­ها به مدل است. همانطور که در شکل مشاهده می­کنید، این در سطح OOAS انجام می­شود. استفاده از جهش­ها به طور مستقیم برای OOAS، مزیت قابل اجرا بودن برای تمام زبان­های پایان جلو است (نه تنها UML). چیزی که ممکن است دیده شود احتمال معرفی "جهش­های UML معنایی" به عنوان یک نقطه ضعف است، اما از آنجایی که نگاشت UML به OOAS، ساختار مدل UML را حفظ می­کند، اکثریت جهش­ها دارای یک نگاشت یک به یک به UML هستند. برای مثال، عملگر جهش غیر فعال کردن یک نقشه فرمان محفوظ به غیر فعال کردن انتقال در مدل UML.

جدول 2 یک مرور کلی از عملگرهای جهش استفاده شده در آزمایش­های شرح داده شده در این مقاله ارائه می­کند. بعد از اضافه کردن جهش، OOAS برای کد ماشین اجرا می­شود و سرانجام مرحله تولید مورد تست آغاز می­شود. مانند تمام ابزارهای تست مدل، MoMuT موارد تست در سطح انتزاع از مدل تحویل می­دهد – به عبارت دیگر موارد تست انتزاعی [19]. از این رو قبل از اجرای هر تست در SUT، آنها ممکن است نیاز به واقعی شدن داشته باشند. به عنوان یک فرمت خروجی مورد تست، MoMuT فرمت دبران[[8]](#footnote-8) را استفاده می­کند، اما یک نقطه نمودار[[9]](#footnote-9) را نیز می­نویسد.

**2.1 زبان ورودی: سیستم­های عمل شیء گرا**

OOAS یک زبان مدل­سازی مناسب هستند که دارای معناشناسی رسمی هستند، نسبتاً ساده هستند، و به خوبی برای بیان سیستم انتقال حالت گسسته مناسب هستند. یک مثال از OOAS می­تواند در سایت [www.momut.org](http://www.momut.org) یافت شود. این زبان بر اساس یک تعمیمی از زبان فرمان محفوظ دیکسترا[[10]](#footnote-10) [14] و فرمالیسم سیستم عمل برگشت[[11]](#footnote-11) [5] است. این شبیه به رویداد-B در برخی مفهوم است، اگرچه از نظر دستورات و اقدامات محفوظ کمتر محدود است.



شکل 2: مثال OOAS

شکل 2 یک مدل نمونه در MoMuT::UML در حال تجزیه را نشان می­دهد. هر سیستم عمل ممکن است ویژگی­ها، روش­ها، اقدامات برچسب­شده، و یک بلوک do-od را اعلام کند، که اجرا را تحریک می­کند. تا زمانی که یک عمل در بلوک do-od می­تواند اجرا شود، بلوک تکرار کردن را ادامه می­دهد. MoMuT:UML چندین تعمیم برای سیستم عمل اصلی را استفاده می­کند، یکی از اقدامات برچسب شده است. عمل می­تواند با "obs" (قابل مشاهده)، "ctr" (قابل کنترل) و یا نه، علامت­دار شود، که به این معنی است که آن یک عمل درونی است. توجه کنید که ما بازگشت را در OOAS نمی­پذیریم و همچنین حلقه­ها را ارائه نمی­دهیم (به استثنای بلوک do-od). در عوض، زبان یک عملگر برابر فراهم می­کند. انواع داده­ها، از جمله فهرست­ها و اعداد صحیح، نیاز به اعلام شدن با کران­ها دارند. دسترسی مستقیم به ویژگی­های اشیاء دیگر ممنوع است. در عوض، روش گیرنده/گذارنده[[12]](#footnote-12) استفاده می­شود.

توضیحات می­توانند با ترتیبی (";")، غیر قطعی ("[]")، یا اولویت­بندی ("//") عملگرهای ترکیب، ترکیب شوند. نمونه شیء فقط روی ویژگی­های مقداردهی اولیه به عنوان MoMuT::UML مجاز است که مجموعه­ای از اشیاء مؤثر در زمان اجرا را محاسبه می­کند. فراخوانی روش روی شیء (late-binding) پشتیبانی نمی­شود. ویژگی­های زبان یک راه برای بیان یک سیاست برنامه­ریزی ساده: بر اساس سیستم، اشیاء می­توانند با توجه به اولویت ("//") روی اشیاء سیستم­های دیگر باشند. ویژگی نهایی زبان، پروژه نامیده می­شود. با توجه به جمله var input: t\_UserAction: c\_userAction(input) که در آن t\_UserAction یک شمارنده با مقدار SetPause…StopMeasurment است، سیستم برای اجرای c\_userAction با تمام مقادیر ممکن t\_UserAction تلاش خواهد کرد. به طور متفاوت، آن یک مخفف است برای گفتن c\_userAction(SetPause) [] … [] c\_userAction(StopMeasurement).

تفاوت اصلی بین OOAS و زبان­های متعارف، راه ترکیب متوالی کارها همراه با دستورات محافظت شده ("نیاز") است. اگر ما یک جمله نیاز (A): skip end، نیاز(B): skip end داشته باشیم، سیستم ابتدا بررسی می­کند که آیا A و B برآورده می­شوند در زمان مورد نیاز قبل از اجرای جمله ترکیب شده. این سیستم، به عبارت دیگر، نیاز به محاسبه قابل فعال بودن اعمال دارد. برای مدل انیمیشن این بدان معنی است که نوعی از عقبگرد الزامی است. همچنین، چنانکه همزمانی از طریق انتخاب غیر قطعی بیان می­شود، یک موتور جستجو کارآمد مورد نیاز است، در هنگام محاسبه تمام آثار ممکن از طریق OOAS.



شکل 3: گراف تصمیم یک OOAS تولید شده از UML

با توجه به تمام راه­های مختلف ترکیب اعمال در راه­های غیرمتوالی، آن جالب توجه است که چگونه بسیاری از سطوح عمقی این ساختارها معمولاً تو در تو است. شکل 3 یک برش از گراف تصمیم گرفته شده از مدل ایستگاه راه­آهن بزرگ را نشان می­دهد. گره قرمز بلوک اولیه است، که بخشی از زمانبندی است. این جایی است که اجرا شروع می­شود. تمام دیگر گره­ها، رنگی شده، گره­ها نشان دهنده نقاط عقب­گرد/تصمیم هستند. گره خاکستری برای ترکیب اولیه، گره نارنجی برای ترکیب غیرقطعی، و گره بنفش برای تصویر[[13]](#footnote-13) (خود حلقه نشان دهنده مقدار ارزش برای امتحان کردن است) استفاده می­شود. گره­های سفید نشان دهنده یک پایان[[14]](#footnote-14)، بلوک متوالی است. همانطور که دیده می­شود، نگاشت UML معمولاً ساختارهای عقب­گرد بسیار عمیق تولید نمی­کند. با این حال، این یک نگرش ایستا است و عملگرهایی مانند fold، foral و با جزئیات کامل را به حساب نمی­آورد.

**3. یک معماری برای جهش کوتاه مدت**

در مقایسه با تولید­های قبلی MoMuT::UML، تغییرات مهم معماری (a) کامپایل به موقع محلی[[15]](#footnote-15)، (b) فشرده­سازی جهش از طریق طرح، (c) اتخاظ روش­های کاهش سفارش جزئی، و (d) نمونه پویا از جهش کوتاه مدت، هستند. به منظور تسهیل در این تغییرات، پایان عقب به طور کامل با تمرکز بر عملکرد بازنویسی می­شود.

گام اول نسبت به این هدف، اجرای محلی مدل OOAS است. این جایی است که MoMuT متکی بر چارچوب کامپایلر LLVM [11] برای کامپایل به موقع مدل باشد. از آنجا که OOAS ذاتاً غیر قطعی هستند و نیاز به عقبگرد دارند، اولین نمونه اصلی، کد را برای جستجو عقبگرد در نمایش واسط LLVM (LLVM-IR) از مدل در خط[[16]](#footnote-16) می­کند. با این حال، اثبات بسیار ضعیف (منطق حفظ پشته، ناتوانی برای مقایسه حالت­های میانی، الگوریتم جستجوی ثابت)، بنابراین ما به رویکرد اجرای رو به جلو روی می­آوریم. اجرای رو به جلو به این معنی است که هیچ عقبگردی در داخل مدل وجود ندارد. در عوض، در حین اجرا، مدل کامپایل شده درخواست اجزاء زمانبندی خارجی می­کند، که بخشی از زمان اجرا ارائه شده توسط MoMuT است، که جایگزینی برای گرفتن است. این برای یک ردیابی در بلوک do-od در یک زمان انجام می­شود. در مورد زمانبندی کردن هدایت موتور اجرا برای برخی از دستور محافظت که غیرفعال است، اجرا لغو شده و زمانبند در مورد نتیجه مطلع می­شود.



شکل 4: معماری MoMuT::UML 3.x TCG

در نمودار کلاس ساده شده شکل 4، کلاس­های FWTraceExecutor و FWDfsScheduler چارچوب اجرای اساسی را می­سازند. توجه داشته باشید که اجرای رو به جلو در مورد ما، به عنوان اعمال در داخل یک تکرار بلوک do-od از یک سیستم که توابع بسیار پیچیده ریاضی را محاسبه نمی­کند ، کار می­کند. بنابراین زمان صرف شده برای محاسبه دوباره نتایج، ناچیز است.

با استفاده از این چارچوب اجرای ساده مبتنی بر ردیابی به عنوان پایه، MoMuT::UML موفق در اضافه کردن یک جستجوگر عمومی (FWExplorerGeneric) شده و یکی که سفارشات جزئی (FWPartialOrderExplorer) را در بالا به حساب می­آورد. در واقع برای تولید موارد تست، یک نمونه از کلاس FETcgGraphExplorer می­تواند استفاده شود. این کلاس FWPartialOrderExplorer را می­گیرد و منطق برای ایجاد یک نمودار تست را می­افزاید. منطقی که استراتژی جهش کوتاه مدت را ایجاد می­کند، در FWOtfTaskmaster پیاده­سازی می­شود. این کلاس دارای مجموعه­ای از کارگران (FWOtfWorker) است و وظایف اکتشاف را مدیریت می­کند (FWOtfTask). هر کارگر در رشته خود اجرا می­شود و یک نمونه از FWOtfExplorer را دارا است، یک زیر کلاس از FWTcgGraphExplorer.

در مجموع، معنی که شامل موتور عقب­گرد است، MoMuT::UML شامل حدود 74600 LoC جاوا است، و 59300 LoC کدهای C++. بخش جاوا از کامپایلر OOAS اخیراً تحت BSD منتشر شده و در [www.momut.org](http://www.momut.org) موجود است. ما در حال حاضر به دنبال گزینه­های ساخت منبع C++ در دسترس تحت برخی مجوز تحقیقات دانشگاهی هستیم.

**3.1 طرح جهش**

برای پروژه جهش کوتاه مدتمان، ما به یک مدل کامپایل شده نیاز داریم که، در هر نقطه، شامل اطلاعات در مورد جهش­های ممکن باشد. همچنین، ما می­خواهیم فراخوانی­هایمان به کامپایلر LLVM JIT به حداقل محدود کنیم و به نمونه جهش کم ارزش در هر حالت سیستم داده شده نیاز داریم. این منجر به طراحی زیر می­شود. اول، موتور جهش روی درخت نحو انتزاعی[[17]](#footnote-17)(AST) از OOAS قبل از آنکه به LLVM-IR تبدیل شود، می­گردد. با توجه به معیارهای انتخاب که ممکن است توسط کاربر تعیین شود، موتور جهش گره­ها را در AST انتخاب می­کند و یک تفسیر جهش را اضافه می­کند که در اصل یک جایگزین AST با جهش مندرج است. دوم، در طول تولید LLVM-IR، یک تولید کننده کدهای تخصصی در تمام تفسیرهای جهش به نظر می­رسد و کدهای اضافی و فراخوانی­های اضافی به زمان اجرا MoMuT::UML تولید می­کند که تععین می­کند که آیا اجرا، مدل اصلی یا هر یک از جهش­ها را دنبال می­کند. به طور فنی، این با تطبیق شناسه جهش انجام می­شود. مدل اصلی دارای شناسه جهش 0، و هر جهش واقعی یک مقدار بیشتر را دارد. هنگام معرفی FWTraceExecuter، موقعیت پیکربندی توصیف می­کند که شناسه جهش باید در حین اجرا فعال شود. این طراحی این امکان را برای فعال کردن یک مجموعه از جهش برای یکبار می­دهد، به طور مؤثر MoMuT::UML را برای مقابله با جهش­های سفارش بالاتر قادر می­سازد. در این کار، با این حال، ما همیشه فقط یک شناسه جهش را انتخاب می­کنیم. بنابراین در پایان فعال یا غیرفعال کردن، یک جهش یک موقعیت مقدار صحیح در پیکربندی موتور اجرا می­شود که تسهیل نمونه جهش­یافته است. همچنین، چونکه مدل کامپایل شده زمان اجرا که جهش یافته­اند را برای انتخاب کردن می­خواهد، زمان اجرا در مورد تمام جهش­های ممکن در مسیر جاری را می­داند و می­تواند تصمیم بگیرد که آیا یک جهش تولید کند یا نه. توجه داشته باشید که ما نیاز به ساختار حالت برای بدون تغییر باقی ماندن بین اصلی و جهش­یافته داریم، که مجموعه­ای از عملگرهای جهش ممکن را محدود می­کند.

الگوریتم 1. TCG با جهش­های کوتاه مدت



**3.2 تولید مورد تست با جهش کوتاه مدت**

الگوریتم 1 حلقه کنترل اصلی MoMuT::UML را طرح ریزی می­کند که برای جهش­های شروع/توقف استفاده می­شود و راه خود را از طریق فضای حالت پیدا می­کند. الگوی اصلی این الگوریتم کشف مدل اصلی از حالت داده شده است، به عنوان مثال، حالت اولیه، تا نقطه انتخاب بعدی که در آن تست کننده برای انتخاب یک عمل قابل مشاهده مورد نیاز است. به عبارت دیگر، تمام اعمال داخلی و تمام اعمال قابل مشاهده که یک انتخاب در بخشی از تست کننده را تضمین نمی­کند، به طور خودکار گسترش یافته است. در طی اکتشاف مدل اصلی، تعدادی از جهش­یافته­های منتخب ردیابی می­شوند.

پس از رسیدن به نقطه انتخاب بعدی، اکتشاف مدل اصلی متوقف می­شود و جهش­یافته­های جدید معرفی می­شوند. نمونه جهش­یافته نه تنها بر اساس اطلاعات بدست آمده به هنگام کاوش اصلی است، بلکه تعدادی از جهش­های در حال اجرا و یک محدودیت تعریف شده کاربر از حداکثر تعداد از جهش­های فعال را به حساب می­آورد. بعد از بروز رسانی لیست جهش­های فعال، سیستم دوباره نقش ورودی را بازی می­کند، مدل اصلی مواجه می­شود، در تمام جهش­ها. در مرحله بعد، جهش­ها فیلتر می­شوند. تمام جهش­یافته­هایی که قابل مشاهده به نظر می­رسند، توسط اصلی پیش­بینی نمی­شوند، خاتمه یافته در نظر گرفته شده و حذف می­شوند. تمام جهش­یافته­هایی که در طول اجرا ازبین می­روند نیز حذف خواهند شد. همچنین تمام جهش­یافته­هایی که به حالت مساوی رسیدند به عنوان اصلی حذف می­شوند و سرانجام جهش­یافته­هایی که برای مراحل بسیار زیاد زنده بوده­اند (20)، حذف می­شوند و به عنوان "تسلیم[[18]](#footnote-18)" علامت گذاری می­شوند. در نهایت، MoMuT خواستار روش اکتشافی برای انتخاب یک ورودی جدید می­شود و مدل اصلی را دوباره بررسی می­کند، خاتمه دادن به حلقه.

در حال حاضر دو روش اکتشافی مختلف برای انتخاب ورودی بعدی اجرا می­شود. روش اکتشافی اول فقط یک انتخاب تصادفی وزن­دار در سرتاسر مجموعه اعمال موجود را استفاده می­کند. روش اکتشافی، اکتشاف را در یک مفهوم راهنمایی می­کند که آن اعمال صورت گرفته را جریمه می­کند به طوری که اعمالی که پیش از این هرگز انتخاب نشده­اند را به آنهایی که قبلاً گرفته شده­اند ترجیح می­دهد. روش اکتشافی دوم یک زیرمجموعه محدود از اعمال موجود انتخاب می­کند و به بررسی همه آنها می­پردازد. پس از انجام، عمل منجر به انتخاب شدن جدیدترین نامزد جهش می­شود. در موردی که هیچ کدام در دسترس نیست، روش اکتشافی به انتخاب تصادفی هدایت شده که قبلاً شرح داده شد پس رفت می­کند.

**4. مطالعات موردی**

جدول 3: ویژگی­های مدل­های تست



Time: آیا مدل UML از محرک­های زمان استفاده می­کند. مانند زمان گسسته.

Objs: تعداد نمونه نمونه در مدل UML. حالت ماشین­های در حال اجرا موازی.

Traces: حداکثر نظری برای یک بلوک اجرای do-od. محدود شدن لیست یک عنصر.

Stmnts: تعداد جملات OOAS.

Attrs: تعداد ویژگی­ها. مانند متغیرهای غیرمحلی/"سطح کلاس"

State Size: در کیلو بایت. حافظه مورد نیاز برای نگه­داشتن تمام ویژگی­های تمام نمونه­ها. M7: کران پایین.

Ctrs: تعداد رویدادهای قابل کنترل در مدل.

Init-τ: تعداد ردهای[[19]](#footnote-19) لازم برای گسترش تمام τها از حالت اولیه، بعد از POR.

Init-S: تعداد مراحل گسترش لازم برای گسترش تمام τها از حالت اولیه، بعد از POR.

جدول 3 یک مقایسه برخی از خواص کلیدی مدل­های تست در نظر گرفته شده در این مقاله را فراهم می­کند. مدل­های M1 و M3 که در انتشار قبلی [1و2و3] شرح داده شد و بدون تغییر باقی ماند، در حالی که مدل M2 یک مدل جدید برای یک مورد استفاده شرح داده شده در [3] است. جدول به سه ستون اصلی جدا می­شود، خواندن UML، OOAS و TCG. ویژگی­های لیست شده در زیر ستون UML به مدل UML مربوط است: آیا مدل از محرک­های زمان استفاده می­کند، و چه تعداد اشیاء معرفی می­شود. برای اکثر مدل­ها، تعداد اشیاء مربوط به تعداد ماشین­های حالت در حال اجرای موازی است، به استثنای مدل M5 که کلاس غیرفعال معرفی می­کند. لطفاً توجه داشته باشید که ما فقط از نمودارهای حالت UML، نمودارهای کلاس و نمودارهای نمونه در این کار استفاده می­کنیم.

ستون OOAS در جدول 3 لیست خواص کلیدی مدل­های UML مبدل است. به طور خاص، آن تعداد ردیابی­های اجرای مختلف که ممکن هستند را نشان می­دهد، با توجه به اعمال متشکل غیرقطعی از بلوک­های do-od. توجه داشته باشید که از لیست­های پشتیبانی MoMuT::UML، مدل­های OOAS نیاز به یک تکرار روی آنها را دارد. از آنجاکه ما بازگشت و حلقه را در OOAS نمی­پذیریم، این قابلیت­ها از طریق عملگر fold موجود در زبان فراهم می­شود. تعداد ردیابی­های گزارش شده، که به طور کلی وابسته به تعداد دفعاتی که عملگر fold در لیست اعمال شده است می­باشد، فرض می­شود که همه لیست­ها دارای طول یک عنصر است. ویژگی دوم نشان داده شده، تعداد جملات است، که توسط کامپایلر OOAS پیدا شده است. در نهایت، تعداد ویژگی­های در سطح سیستم داده می­شود.

ستون TCG ویژگی­های پویا مدل­ها را توصیف می­کند. اول، اندازه حالت در کیلو بایت داده شده است. برای تمام مدل­ها، بجز M7، این تعداد به عنوان محتوی حداکثر طول تمام لیست­های استفاده شده به عنوان ویژگی­ها محاسبه می­شود و، از این رو نشان دهنده یک کران بالا است. برای M7 این تعداد بی معنی می­شود (>8 MiB، بیش از 9000 لیست)، از این رو ما کران پایین اندازه حالت را می­گیریم، با این فرض که تمام لیست­ها خالی است. دوم، تعداد رویدادهای قابل کنترل گزارش می­شود. قابل کنترل­ها ورودی­هایی هستند که تست­کننده می­تواند یک سیستم تحت تست (SUT) بدهد. برخی از مدل­ها استفاده از قابل­کنترل­های پارامترشده را سنگین می­کند، از این رو تعداد فقط نشانه­ای از پیچیدگی­های درگیر است. به طور خاص مدل M2 دارای قابل کنترل­های پارامترشده است که منجر به بیش از 1600 ورودی ممکن برای SUT می­شود. سوم، تعداد آثار موتور تولید مورد تست نیاز به دیدن در هنگام گسترش تمام اعمال داخلی از حالت اولیه دارد، گزارش شده است. در نهایت، ستون آخر تعداد تکرارهای بلوک do-od را نشان می­دهد که برای گسترش تمام اعمال داخلی با شروع از حالت اولیه لازم هستند. لطفا توجه داشته باشید که ارقام دو ستون آخر پس از کاهش سفارش جزئی (POR) می­باشد. در ادامه این بخش ما موارد مورد استفاده را به صورت جداگانه معرفی می­کنیم.



شکل 5: نمودار حالت M1

M1- سیستم هشدار دهنده. M1 یک مدل ساده از سیستم هشداردهنده (دزدگیر) ماشین است. نتایج قبلی با نسل­های قبلی از MoMuT و مدل در [1و3] توصیف شده است. تنها پیچیدگی عرضه شده توسط مدل، استفاده از محرک­های زمان است که بر عملگرهای fold در سرتاسر لیست اشاره دارد. آن همچنین از مناطق مستقل استفاده می­کند همانطور که در شکل 5 مشخص است.

M2- بسته اجرای بارکننده[[20]](#footnote-20). مدل M2 حلقه کنترل (از جمله رفع خطا) یک کنترل کننده اجرای بسته بارکننده است. کنترل کننده مقدار انحراف دسته[[21]](#footnote-21) را به عنوان ورودی دریافت می­کند و مقدار خروجی را دریافت می­کند که دریچه­ها را به عقب نشانده در کنترل کردن حرکات بسته­ها. اگرچه M2 یک مدل نسبتاً کوچک است، آن بسیار پیچیده است، که می­تواند در جدول 3 دیده شود: توجه به اعمال پارامترشده سنگین، آن نیاز به بیشترین تعداد از ردیابی­ها برای یک تکرار از بلوک do-od دارد. به جای اجرای نمادین در MoMuT، ما ورودی­ها را برای تعریف دستی کلاس­های هم­ارزی محدود می­کنیم. یافته­های اولیه با ورژن قبلی از MoMuT و مورد استفاده[[22]](#footnote-22)، در [3] یافت می­شود. لطفا توجه کنید که مدل مورد استفاده در کار جاری، از کار استفاده شده در [3] متفاوت است: M2 یک مدل کامل از سیستم است. ما به مدل­های جزئی که قبلاً داشتیم متوسل نمی­شویم.

M3- دستگاه اندازه­گیری. M3 یک مدل از یک پروتکل کنترل از راه دور دستگاه اندازه­گیری دود اگزوز گرفته شده از صنعت است. یافته­های اولیه ما از نسل مورد تست برای M3 در زمینه صنعتی، قبلا در [2] منتشر شده است. در اینجا، ما از همان مدل با موتور تولید مورد تست جدیدمان استفاده می­کنیم. از نظر پیچیدگی مطرح شده، مدل کمی پیچیده­تر از M1 است.

M4 – دستگاه شوک خروجی خودکار. M4 منطق تشخیصی دستگاه شوک خروجی خودکار را مدل می­کند. ویژگی­ها می­تواند در جدول 3 دیده شود.: آن ویژگی­ها بیشتر از M3 است و کمتر پیچیده است.

M5 – رابط سیستم­های بحرانی ایمنی. M5 یک مدل از زیر سیستم یک رابط سیستم­های بحرانی ایمنی است که در تولید است. در تفاوت با دیگر مطالعات موردی، مدل باعث استفاده گسترده از محرک پاسخ UML [[23]](#footnote-23) می­شود و نزدیک به اجرای واقعی است. اگرچه آن محرک زمان ندارد، آن از زمان گسسته داخلی استفاده می­کند: با کمک اعمال پارامتری شده، تست­کننده در کنترل پیشرفت زمان است. M5 تعداد نسبتا زیادی از اشیاء را معرفی می­کند اما نسبتا به طور همزمان با توجه به استفاده گسترده از محرک پاسخ رفتار می­کند. سربار محاسباتی با توجه به سطح پایین از انتزاع، قابل توجه است.



شکل 6: طرح ایستگاه M7



شکل 7: طرح ایستگاه M6

M6 و M7 – سیستم بهم پیوسته راه­آهن. M6 و M7 نمونه­هایی از سیستم بهم پیوسته راه­آهن هستند. آنها شامل دو مدل هستند: یکی مدل کلی به اشتراک گذاشته شده که تمام کلاس­ها و ساختارهای داده را تعریف می­کند، و یکی که اشیاء مورد نیاز برای ایستگاه را معرفی می­کند. در حالی که M6 یک حداقل ایستگاه که اجازه می­دهد قطارها با یکدیگر گذر کنند را نشان می­دهد، M7 یک مدل از یک ایستگاه راه­آهن واقع در اتریش است. طرح آن در شکل 6 نشان داده شده و شامل 37 بخش مسیر، 56 رله مسیر، 34 سوئیچ، 22 سیگنال اصلی، و 145 مسیر قطار که اپراتور می­تواند انتخاب کند است. M6 در مقابل، تنها شامل 10 بخش مسیر، 4 رله مسیر، 2 سوئیچ، 6 سیگنال اصلی، و 10 مسیر قطار است. هر دو مدل بسیار غیر قطعی با توجه به شبکه­های 2847 (M7) و 125 (M6) ماشین حالت اجرای موازی هستند. ماشین­های حالت برای هر دو مدل استفاده می­شوند، فیزیکی، همچنین وجود منطقی، مانند مسیرهای قطار. نه M6 نه M7 شامل زمان و یا اعمال پارامتری شده نیستند. با این حال، هر دو مدل استفاده گسترده­ای از لیست­ها می­کنند و برای همه/موجود کمیت­سنج­ها. برای مثال، M7 شامل بیش از 9000 لیست در حالت است، دارای بیش از 50 کمیت­سنج موجود و بیش از 100 برای تمام کمیت­سنج­ها است که بیشترین عمق تودرتویی از 5 را دارد. علاوه بر این، هر دو مدل دارای یک توالی اولیه طولانی از اعمال قابل مشاهده نسبتا مرتب قبل از اینکه قابل کنترل مشاهده شوند است. M7، برای مثال، نیاز به MoMuT دارد برای نگاه کردن به رد­های 83 و 389 و 266، فقط برای محاسبه اولین – و گران­ترین- مرحله در این دنباله طولانی. در مجموع، توالی اولیه شامل 264 مرحله است. تنها بعد از این توالی اولیه کامل شد، MoMuT می­تواند شروع به ایجاد ورودی­هایی کند که منطق به هم پیوسته را بررسی می­کند. توجه کنید که M7 به طور مستقیم از داده­های ایستگاه اصلی استفاده شده برای پیکربندی سیستم بهم پیوسته کامپیوتری در ایستگاه مشتق می­شود. به این ترتیب M7 پیچیده­ترین مدل ارائه شده دراین کار است.

**5. ارزیابی آزمایشی**

ما معیارهای ارزیابی را در دو سرور مختلف اجرا کردیم. دستگاه اول با پردازنده 6 هسته ای Xeon X5690 با فرکانس 3.47 گیگا هرتز مجهز می­شود. این دستگاه 24 هسته منطقی ارائه می­دهد. ویژگی­های سرور دوم دارای دو پردازنده Xeon E-2680 v2 با فرکانس 2.80 گیگا هرتز است و 40 هسته منطقی ارائه می­دهد. هر دو سرور توسط حافظه 192 گیگایابت حمایت می­شوند. ما فقط حدود 100 گیگابایت از حافظه را استفاده می­کنیم، حتی موقع اجرای 40 کار به طور موازی. با توجه به بیش از حد بودن ورود به سیستم، MoMuT::UML ثابت می­شود به حد ورودی/خروجی برای مدل­های کوچک بر روی سرور کوچک.

جدول 4: نتایج ارزیابی TCG



ردیف­ها نتایج اجراهای TCG را نشان می­دهند که 3 مورد تست با 150 انتخاب تولید می­شود. استراتژی S1 روش تصادفی هدایت شده، و استراتژی S2 هنوز مواجه نشده مبتنی بر جهش[[24]](#footnote-24) را نشان می­دهد. آمار و ارقام در پرانتز نشان می­دهند که چه تعداد خطاهای اجرا به دلیل وقفه است. Skipped به این معنی است که جهش­ها پیدا شده­اند اما هرگز برنامه­ریزی نشده­اند. آمار و ارقام کج، مورد تست تولیدشده تنها را نشان می­دهد.

جدول 4 نتایج اصلی ارزیابی ما را نشان می­دهد. هر ردیف برای یک اجرای TCG است که سه مورد تست با عمق 150 انتخاب تولید می­شود. ما این مقادیر را عمدتا برای مقایسه با انتشار قبلیمان انتخاب می­کنیم. متاسفانه، فرایند اجرای طولانی TCG برای M2، یک اشکال در MoMuT بر روی سرور 40 هسته­ای ایجاد می­کند که باعث می­شود ابزار برای تولید کردن یک مورد تست باشد. بنابراین این ارقام کج تنظیم شده است. توجه کنید که برای مدل­های بزرگتر، نسل فقط سه مورد تست برای نتیجه در کمتر از کل پوشش جهش مورد نظر انتظار می­رود، بطوریکه مقدار رفتار ممکن به شدت افزایش می­یابد. نکته دیگر این اجازه را به ما می­دهد که بگوییم، ما می­دانیم برای داشتن جهش­ها در قسمت بیهوده کد[[25]](#footnote-25) در M6 و M7. این بخاطر این واقعیت است که این موارد یک اشتراک را می­سازند، مدل کلی راه­آهن و UML ما برای موتور ترجمه OOAS به اندازه کافی برای خارج کردن کدهای بیهوده باهوش نیست. این همچنین با این واقعیت است که مدل بزرگتر M7، که عناصر بیشتر مدل استفاده می­کند، دارای پوشش جهش کل بهتر نسبت به M6 است. همانطور که جدول نشان می­دهد، روش پیشنهادی ما به طور عجیبی خوب کار می­کند: اگر MoMuT قادر به یافتن جهش­یافته باشد، به احتمال زیاد خاتمه می­یابد.

جدول 5: زمان TCG و تعداد مراحل برای خاتمه[[26]](#footnote-26)



داده­ها برای اجرا با 40 کارگر و یک برش زمان اکتشاف جهش­یافته 2\*time(orig)+3 min داده می­شود، بجز برای M2 که داده­ها از یک اجرای نامحدود با 24 کارگر است.

جدول 5 خلاصه ­ای از اینکه چه تعداد مراحل برای خاتمه نیاز است را نشان می­دهد. داده­ها تمرکز ما روی جهش­های مدت کوتاه را تایید می­کند، به طوریکه اکثر خاتمه­ها زود اتفاق بیفتد. برای مدل­های کوچکتر، ما همچنین می­توانیم پوشش جهش کل بسیار سخت، غالبا فراتر از 85% را گزارش کنیم. این نتایج همچنین در زمینه زمان TCG کل دیده می­شود: آن MoMuT 140 ثانیه برای تولید مورد تست (S2) برای M1 بر روی دستگاه بزرگ با 40 کارگر اجرا شد. ابزار با M3 در 320 ثانیه در همان دستگاه انجام می­شود، دستیابی به پوشش جهش کل حدود 88%. سه تست برای M7 در 22.6 ساعت کامل شد، که آن MoMuT حدود 7.5 ساعت در هر تست اجرا شد. بیشتر هزینه در واقع در محاسبه ایستگاه­دهی اولیه نهفته است: به طوریکه هر عنصر به موقعیت شروع خود حرکت می­کند، MoMuT با مقدار زیادی از همزمانی مواجه می­شود. بدیهی است که این می­تواند به راحتی با استفاده از حالت بعد از مقدار دهی اولیه به عنوان حالت اولیه بهینه سازی شود. از این رو ارقام ارائه شده در اینجا بدترین حالت هستند. در مقایسه با TCG تصادفی خالص، که یک تست تصادفی با 150 انتخاب در حدود 2.5 ساعت تولید می­کند، استراتژی S2 حدود سه برابر گران­تر است. این بخاطر جهش­هایی است برای محاسبه بیشتر طول کشیده، اما همچنین به خاطر کلاک پایین­تر پردازنده[[27]](#footnote-27) هنگام رسیدن حد قدرت بسته آن است. توجه کنید که ما اجازه می­دهیم به جهش­ها که تنها مقدار دو برابر (به علاوه سه دقیقه) از زمان بگیرند، مدل اصلی مورد نیاز برای کامل کردن گام. این عمدتا برای بهبود یافتن از جهش در داخلی است، به عنوان مثال حلقه­ها.



شکل 8: تعداد دفعات حالت­هایی که ملاقات شده است

در طول ارزیابی ما حالت­های MoMuT که در حال کاوش بوده­اند را دنبال می­کنیم. شکل 8 نتیجه نمودار تعداد بازدید در مقابل تعداد حالت­ها برای M1 و M7 و هر دو استراتژی را نشان می­دهد. همانطور که در شکل دیده می­شود، استراتژی S2 تعداد حالت­های ملاقات شده در مقایسه با S1 را افزایش می­دهد. در مجموع، MoMuT کاوش می­کند 185 حالت از مدل اصلی M1 را با توجه به استراتژی S1. این تعداد به 217 حالت افزایش می­یابد، با تغییر به استراتژی S2. برای M7، اشکال شبیه به نظر می­رسند: استراتژی S1، MoMuT 39 539 حالت منحصر به فرد را پیدا می­کند، هنگام کاوش مدل جهش نشده. تغییر به استراتژی S2، این تعداد را به 7 113 946 حالت افزایش می­دهد. شکل دوم عجیب به نظر می­رسد و نیاز به بررسی بیشتر دارد، با این حال، آزمایش ساخت این داده­ها بر روی سرور بزرگ اجرا می­شود. از این رو استراتژی S2 40 اعمال در حالت موازی کاوش می­شود در هنگام تلاش برای پیدا کردن بهترین عمل منتخب بعدی. بجز برای استراتژی S2 در M1، نمودار در شکل 8 آنگونه که ما می­خواهیم به نظر می­آیند: اکثر حالت­ها تنها چندبار بازدید شده است.

سؤال پژوهشی 1: آیا ما می­توانیم MDTCG را برای مدل­ها با اندازه صنعتی در هزینه (زمان منابع) اعمال کنیم، کاربر مایل به قبول کردن خواهد بود؟ بر اساس نتایج، ما تصور می­کنیم می­توانیم استدلال کنیم که، در واقع، ما قادر به رسیدگی به مدل­ها با اندازه صنعتی در هزینه که قابل قبول است هستیم. حتی بزرگترین مدل­ها با هزاران دستگاه­های حالت در حال اجرای موازی توسط یک کامپیوتر در یک زمان و حافظه بودجه کافی به اندازه مدل به کار گرفته می­شود. هیچ یک از هفت مورد مورد نیاز بیش از یک روز از زمان محاسبه، با اکثریت موارد در واقع کمتر از 20 دقیقه است. البته همیشه جا برای بهینه­سازی­های بیشتر وجود دارد، با این حال، MoMuT::UML 3.0 یک خط پایه قوی را ثابت می­کند.

سؤال پژوهشی 2: آیا ما پوشش نقص کافی برای ابزار برای مفید شدن حفظ می­کنیم؟ در مورد مدل­های کوچکتر و انتخاب تنظیمات نسل مورد تست، ما پوشش نقص کافی حفظ می­کنیم. برای مدل­های بزرگتر، ما مقدار پوشش نقص که تصور می­کردیم لازم است را بدست نیاوردیم. به احتمال زیاد، این به دلیل تنها سه مورد تست عمق محدود در حال تولید باشد. تجزیه و تحلیل کامل­تر مورد نیاز است برای تایید این فرضیه و همچنین از جهش­ها در کد بیهوده تحریف نتیجه جلوگیری می­کند. ما نمایش نتیجه روشمان برای مؤثر شدن را استدلال می­کنیم: به طور ویژه مفهوم "همزمان مدت کوتاه" جهش­یافته­ها ارزش خود را به عنوان هر جهش­یافته که معرفی شده با احتمال بالا خاتمه یافته ثابت می­کند. بنابراین، بله، در اصل ما پوشش نقص کافی برای مفید شدن ابزار را حفظ می­کنیم، البته ما به گسترش الگوریتم تولید مورد تست نیاز داریم، بنابراین آن شامل تعداد جهش­های هنوز دیده نشده به عنوان یک معیار توقف می­شود.

سؤال پژوهشی 3: با توجه به دو روش اکتشافی ما، آیا هر دو به طور برابر مناسب هستند؟ از ارقام، به نظر می­رسد هر دو روش اکتشافی رفتار تقریبا یکسان دارند، S2 با داشتن تنها یک مزیت کوچک. این نتیجه غیرمنتظره است و نیاز به بررسی بیشتر دارد. نظر به اینکه S2 کمی گرانتز برای اجرا است، آن به پوشش جهش بهتر منجر شود. چرا این مورد نامشخص باقی مانده نبود. از این رو، تحقیق بیشتر لازم است.

نشر به اعتبار[[28]](#footnote-28). (A) نتایج ما به وضوح به عملگرهای جهش انتخاب شده بستگی دارد. برای آزمایش­های نشان داده شده، ما یک مجموعه استاندارد انتخاب می­کنیم که باید نماینده باشد. ما می­دانیم که جهش­های پیچیده­تر ممکن است ایجاد شود. (B) نتایج همچنین به تعداد جهش­ها در راه­های مختلف بستگی دارد: تعداد زیادی از جهش­ها احتمال برخی یافته را افزایش می­دهد اما آن همچنین مقدار جهش­ها در کد بیهوده را افزایش می­دهد. (C) آزمایش­های ما بر اساس روش­های اکتشافی هدایت شده هستند که ممکن است به طور مختلف مربوط به مولد عدد تصادفی انجام شود. (D) دانستن عمق لازم برای رسیدن به یک ضریب خاتمه بالا، می­تواند تعادل را تغییر دهد. ما به یافتن بعضی زمینه میانی با استفاده از یک عمق تلاش کرده­ایم که باید مدل­های ساده را به خوبی پوشش دهد، اما برای مدل­های بزرگتر کافی نیست. (E) اگرچه ما هفت مطالعات موردی متفاوت ارائه می­دهیم، این هنوز یک نمونه است و ممکن است نماینده (کافی) نباشد. (F) اشکالات نرم­افزار. برای کاهش هر موضوع بالقوه مربوط به اشکالات، ما با دقت نتایج را بررسی می­کنیم، داده­های پرت[[29]](#footnote-29) تحت نظارت نزدیک و با تنظیمات مختلف دویاره اجرا می­کنیم، و فایل­های ورود برای خروجی که ما نمی­توانیم توضیح دهیم اسکن شده و تحقیقات مورد نیاز.

**6. کارهای مرتبط**

کار اخیری که نزدیک به کار ما است، مقاله­ای از just و همکارانش است [9]. در کار آنها، نویسندگان مسئله تست جهش برنامه­های جاوا را بهینه­سازی می­کنند. یکی از سهم آنها، یک سیستم حمل­ونقل هوشمند[[30]](#footnote-30) پویا است که ترکیب جهش تست می­تواند بطور امن ترک شود از اجرای تست واقعی. برای این کار، وسیله نویسندگان برنامه جاوا اصلی، اجرای تست­ها و استفاده از کتابخانه زمان اجرا برای پیگیری است، به عنوان مثال یک عبارت در برنامه یک مقدار مختلف با توجه به ورودی­های تست ارائه شده و عملگرهای جهش را محاسبه می­کند. بر اساس این اطلاعات، نویسندگان می­توانند برای اجرا نشدن ترکیب تست جهش خاص تصمیم بگیرند. نویسندگان همچنین از طرح جهش برای کاهش زمان کامپایل جهش­ها استفاده می­کنند. حالت­های آلوده، چنانکه Just و همکارانش حالت­های اصلاح شده محلی از جهش را فرامی­خوانند، برای MoMuT::UML بررسی می­شوند. هنگامی که MoMuT::UML یک جهش معرفی کرده است، آن در نقطه انتخاب بعدی بررسی خواهد شد که آیا حالت با اصلی متفاوت است یا نه. بر اساس این نتیجه، MoMuT جهش را حفظ خواهد کرد یا آن را از مجموعه جهش­های زنده حذف خواهد کرد.

ابزار MuTMuT [7] نیز برای بهینه­سازی مسئله تست جهش تلاش می­کند. با توجه به مجموعه تست و یک برنامه چند رشته­ای، آن تلاش می­کند یاد بگیرد که جهش­های تست­ها قادر به حصول هستند. مشابه MoMuT، آن از جهش­هایی استفاده می­کند که از یک حالت داده شده از برنامه اصلی شروع می­شود.

در سروی مربوط به McMinn و همکارانش [12]، در حوزه کلی از تست بر اساس جستجو در سال­های اخیر کار انجام شده است. برای مثال، مربوط به تست جهش، Papadakis و همکارانش در [16] نشان می­دهند که یک الگوریتم تپه نوردی عمومی منجر به نتایج خوبی برای تولید مورد تست مبتنی بر جهش از برنامه­های C می­شود. ما همچنین می­خواهیم به خواننده برای یک سروی[[31]](#footnote-31) عالی تست جهش در [8] اشاره کنیم.

**7. نتیجه­گیری و چشم­انداز**

ما نسل سوم از MoMuT::UML را ارائه دادیم که بر روی بالا بردن مقیاس MDTCG برای موارد استفاده با اندازه صنعنی تمرکز دارد. ما کاربرد را با کمک هفت مدل UML نشان دادیم، که بزرگترین آنها یک مدل از یک ایستگاه راه­آهن و شامل بیش از 2800 دستگاه­های موازی در حال اجرا است. در پاسخ به پرسش تحقیق، ما نشان داده­ایم که ما می­توانیم MDTCG را در هزینه قابل قبول (زمان و منابع) بر اساس مطالاعات موردی اعمال کنیم. ما همچنین نشان دادیم که MoMuT::UML قادر به دستیابی به پوشش نقص خوب است، در صورتیکه تعداد و عمق موارد تست با پیچیدگی مدل مطابقت داشته باشد. ما همچنین نیاز به بهبود روش هدایت اکتشافی­مان، مخصوصاً برای مدل­های پیچیده را شناسایی کردیم. از دو روش هدایت اکتشافی پیشنهاد شده، S2 یک یک مزیت کم نسبت به S1 در مدل­های پیچیده­تر دارد، اما این یک مزیت قطعی نیست.

در مقایسه با نسل­هی قبلی MoMuT::UML، نسخه ارائه شده یک جهش بزرگ است. آن به پوشش جهش قابل مقایسه در مدل­های نسل­های قبلی دست­یافته که قادر به رسیدگی در زمان کمتر است و نیز در کنترل مدل­های صد برابر پیچیده­تر موفق شد. ابزار بهبود آینده بر روی محتوی گران­تر، به عنوان مثال رسمی، متمرکز خواهد شد، روش­ها برای بالا بردن ضریب تشخیص جهش کل، روش اکتشافی ارائه شده بی­اثر می­شود. آن برای ادغام تکنیک­های اکتشاف نمادین و برای اضافه کردن استراتژی­های جستجو بیشتر و توابع برازش مانند معیارهای فاصله برای ابزار برنامه­ریزی شده است. در طول ارزیابی، ما همچنین بهبود برنامه­ریزی وظیفه ممکن در الگوریتم ارائه شده را تشخیص دادیم.

**تشکر و قدر دانی**

نویسندگان می­خواهند از شرکای صنعتی و به ساختن مدل­ها کمک کردند تشکر کنند. به خصوص Werner Schutz، Peter Tummeltshammeer و همکاران هستند. این مقاله ممکن نمیشد بدون حمایت­های آنها. نویسندگان همچنین می­خواهند از گروه Bernard Aichering در دانشگاه Graz برای همکاری مستمرشان تشکر کنند. تحقیق منجر به این نتایج، کمک­های مالی برنامه هفتم اتحادیه اروپا را دریافت کرده است، برای CRYSTAL Critical System Engineering Acceleration Joint Undertaking تحت موافقت­نامه کمک بلاعوض شماره 332830 و از آژانس تحقیقات اتریش (FFG) به نماینگی از وزارت فدرال حمل­ونقل و نوآوری و فناوری اتریش.

**References**

1. B. Aichernig, H. Brandl, E. Jobstl, W. Krenn, R. Schlick, and S. Tiran. Mo- Mut::UML model-based mutation testing for UML. In Software Testing, Veri\_- cation and Validation (ICST), 2015 IEEE 8th International Conference on, pages 1-8, April 2015.

2. B. K. Aichernig, J. Auer, E. Jobstl, R. Koro\_sec, W. Krenn, R. Schlick, and B. Schmidt. Model-based mutation testing of an industrial measurement device. In M. Seidl and N. Tillmann, editors, Tests and Proofs, volume 8570 of Lecture Notes in Computer Science, pages 1-19. Springer International Publishing, 2014.

3. B. K. Aichernig, H. Brandl, E. Jobstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. Software Testing, Veri\_cation and Reliability, Feb. 2014.

4. P. Ammann, M. E. Delamaro, and J. O\_utt. Establishing theoretical minimal sets of mutants. In Proceedings of the 2014 IEEE International Conference on Software Testing, Veri\_cation, and Validation, ICST '14, pages 21-30, Washington, DC, USA, 2014. IEEE Computer Society.

5. R.-J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In PODC, pages 131-142. ACM, 1983.

6. G. Fraser, F. Wotawa, and P. Ammann. Issues in using model checkers for test case generation. Journal of Systems and Software, 82(9):1403 - 1418, 2009.

7. M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: E\_cient exploration for mutation testing of multithreaded code. In Software Testing, Veri\_cation and Validation (ICST), 2010 Third International Conference on, pages 55-64. IEEE, 2010.

8. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on, 37(5):649-678, 2011.

9. R. Just, M. D. Ernst, and G. Fraser. E\_cient mutation analysis by propagating and partitioning infected execution states. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, pages 315-326. ACM, 2014.

10. W. Krenn, R. Schlick, and B. K. Aichernig. Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems. In F. S. d. Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers, volume 6286 of LNCS, pages 186-207. Springer, 2009.

11. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75-88, San Jose, CA, USA, Mar 2004.

12. P. McMinn. Search-Based Software Testing: Past, Present and Future. In 2011 IEEE Fourth International Conference on Software Testing, Veri\_cation and Val- idation Workshops (ICSTW), pages 153-163, Mar. 2011.

13. L. Morell. Theoretical insights into fault-based testing. In , Proceedings of the Second Workshop on Software Testing, Veri\_cation, and Analysis, 1988, pages 45062-, 1988.

14. G. Nelson. A generalization of Dijkstra's calculus. ACM Trans. Program. Lang. Syst., 11(4):517-561, 1989.

15. A. J. O\_utt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, Dept. of Information and Software Systems Eng., George Mason Univ., 1996.

16. M. Papadakis and N. Malevris. Killing mutants e\_ectively a search based approach. In M. Virvou and S. Matsuura, editors, Knowledge-Based Software Engineering - Proceedings of the Tenth Conference on Knowledge-Based Software Engineering, JCKBSE 2012, Rodos, Greece, August 23-26, 2012, volume 240 of Frontiers in Arti\_cial Intelligence and Applications, pages 217-226. IOS Press, 2012.

17. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. Soft- ware - Concepts and Tools, 17(3):103-120, 1996.

18. R. H. Untch, A. J. O\_utt, and M. J. Harrold. Mutation analysis using mutant schemata. In Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '93, pages 139-148, New York, NY, USA, 1993. ACM.

19. M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

1. Test case [↑](#footnote-ref-1)
2. Mutation [↑](#footnote-ref-2)
3. Mutation-driven test case generation [↑](#footnote-ref-3)
4. Fault-based [↑](#footnote-ref-4)
5. Killed [↑](#footnote-ref-5)
6. Front end [↑](#footnote-ref-6)
7. Back end [↑](#footnote-ref-7)
8. Aldebaran [↑](#footnote-ref-8)
9. Dot-graph [↑](#footnote-ref-9)
10. dijkstra [↑](#footnote-ref-10)
11. Back’s action system formalism [↑](#footnote-ref-11)
12. Getter/setter [↑](#footnote-ref-12)
13. projection [↑](#footnote-ref-13)
14. Terminal [↑](#footnote-ref-14)
15. Native just-in-time compilation [↑](#footnote-ref-15)
16. Inline [↑](#footnote-ref-16)
17. Abstract syntax tree [↑](#footnote-ref-17)
18. Give up on [↑](#footnote-ref-18)
19. Traces [↑](#footnote-ref-19)
20. Loader [↑](#footnote-ref-20)
21. Joystick [↑](#footnote-ref-21)
22. Use case [↑](#footnote-ref-22)
23. UML-call-trigger [↑](#footnote-ref-23)
24. Not-yet-encountered mutants based [↑](#footnote-ref-24)
25. Dead code [↑](#footnote-ref-25)
26. Kill [↑](#footnote-ref-26)
27. CPU clocking lower [↑](#footnote-ref-27)
28. Threats to validity [↑](#footnote-ref-28)
29. Outliers [↑](#footnote-ref-29)
30. Pre-pass [↑](#footnote-ref-30)
31. Survey [↑](#footnote-ref-31)