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

چکیده

در زمینه تست جعبه سیاه، موارد تست[1] تولید از طریق مدل کردن جهش[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] (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

[1] Test case
[2] Mutation
[3] Mutation-driven test case generation

جهش رانده شده (MDTCG)، که یک شکل مبتنی بر نقص[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

[4] Fault-based

Re: بررسی پالایش نمادین + قابل دسترسی

S: جهش قوی + IOCO

W: جهش ضعیف

EPOR: کاهش سفارش جزئی شمارشی

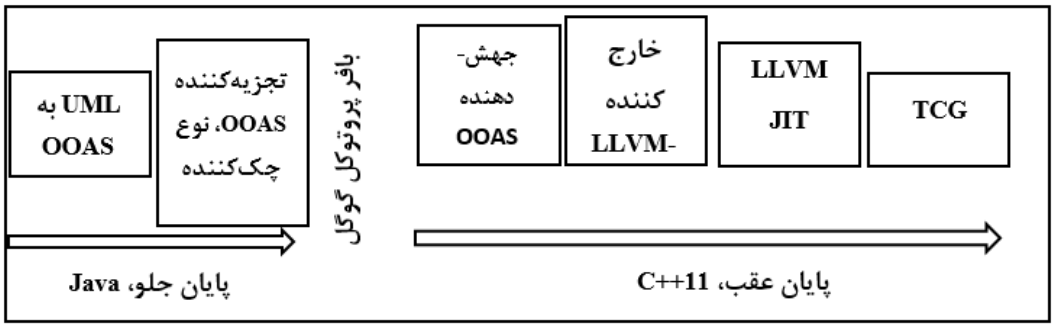
SMT: نظریه ارضا پیمانه

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 می-شود. 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
O(g) غیر فعال کردن فرمان محفوظ	g. . . # guarded commands
O(b) جایگزینی عملگر دودویی	b… # of ≥,>,≤,<,=,≠,+,-,*,div,mod,and,or,
O(u) جایگزینی عملگر یگانی	u. . . # of -,+,abs,not,
O(x) معکوس کردن بولین	x. . . # of true/false in rhs of assignments
O(x) جایگزینی عدد صحیح	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]") 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]، نوشته شده در جاوا، با یک پایان عقب[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.

[5] Killed
[6] Front end
[7] Back end

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] را استفاده می‌کند، اما یک نقطه نمودار[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 یافت شود. این زبان بر اساس یک تعمیمی از زبان فرمان محفوظ دیکسترا[10] [14] و فرمالیسم سیستم عمل برگشت[11] [5] است. این شبیه به رویداد-B در برخی مفهوم است، اگرچه از نظر دستورات و اقدامات محفوظ کمتر محدود است.

[8] Aldebaran
[9] Dot-graph
[10] dijkstra
[11] Back’s action system formalism

```
types
    t_Status = {Ready, Busy};
    t_UserAction = {SetPause, SetStandby, StartMeasurement, StopMeasurement};
    Example = autocons system
    ||[
        var
            status                    : t_Status = Ready;
            runDeviceStateChangeEvents: bool     = false
        methods
            setStatus(newStatus: t_Status) =
                requires newStatus <> status: o_statusChanged(newStatus); end // skip
            end;
            delayedActions: bool = result := runDeviceStateChangeEvents end
        actions
            obs o_statusChanged(newStatus: t_Status) = requires true: skip end;
            ctr c_userAction(action: t_UserAction) =
                requires not delayedActions():
                    requires action = t_UserAction.SetStandby:
                    /* ... */
                end;
            i_delayedDeviceStateActions =
                requires runDeviceStateChangeEvents:
                    runDeviceStateChangeEvents := false
                end
        do
            i_delayedDeviceStateActions() []
            var input: t_UserAction: c_userAction(input)
        od
    ]||
system
    Example
```

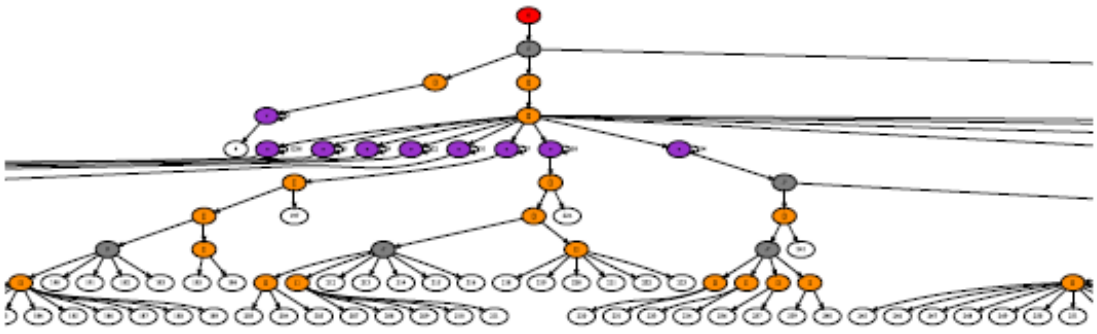
شکل 2: مثال OOAS

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] استفاده می‌شود.

[12] Getter/setter

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 راه-آهن 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[۱۳] (خود حلقه نشان دهنده مقدار ارزش برای امتحان کردن است) استفاده می‌شود. گره‌های سفید نشان دهنده یک پایان[۱۴]، 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.

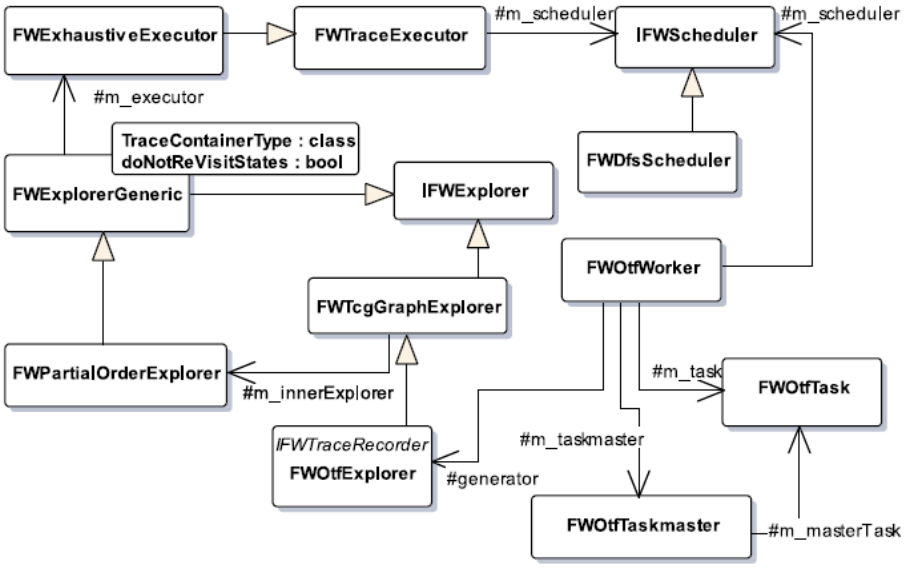
3. یک معماری برای جهش کوتاه مدت

در مقایسه با تولیدهای قبلی MoMuT::UML، تغییرات مهم معماری (a) کامپایل به موقع محلی[۱۵]، (b) فشرده-سازی 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[۱۶] 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

[13] projection
[14] Terminal
[15] Native just-in-time compilation
[16] Inline

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 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](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 تعیین 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

[17] Abstract syntax tree

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

```
procedure GENERATE(numTests, depthTest)
    error ← 0
    test ← 0
    while test < numTests do
        master ← submitInitialTask()                    ▷ Explore original until first choice
        error ← waitForTaskCompletion()
        if error ≠ 0 then
            break
        end if
        choiceString ← ""
        choiceStates ← master.getStartState()
        choiceCount ← 0
        while choiceCount < depthTest do
            queueLiveMutants(choiceString)              ▷ Schedule running mutants
            queueNewMutants(choiceStates)               ▷ Add new mutants
            error ← waitForTaskCompletion()
            if error ≠ 0 then
                break
            end if
            filterMutants()                             ▷ Killcheck!
            selectNextInput(master, out choiceString, out choiceStates)
            choiceCount ← choiceCount + 1
        end while
        writeTest(master, test)
        test ← test + 1
    end while
    return error
end procedure
```

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 جهش-یافته‌های 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[۱۸]" 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.

[18] Give up on

4. مطالعات موردی

جدول 3: ویژگی‌های مدل‌های تست

	UML		OOAS			TCG			
Model	Time	Objs	Traces	Stmnts	Attrs	State Size	Ctrs	Init-τ	Init-S
M1	Y	1	$55 \cdot 10^6$	362	34	0.3	$8\,2 \cdot 10^3$	4	
M2	Y	2	$108 \cdot 10^9$	748	98	0.8	$8\,3 \cdot 10^3$	2	
M3	Y	1	$41 \cdot 10^3$	877	64	0.5	$26\,1 \cdot 10^2$	2	
M4	Y	2	$3 \cdot 10^3$	1215	67	0.4	$34\,3 \cdot 10^1$	2	
M5	N	151	$3 \cdot 10^3$	3831	479	9.7	$14\,4 \cdot 10^4$	46	
M6	N	125	$2 \cdot 10^3$	4798	404	22.3	$172\,1 \cdot 10^5$	82	
M7	N	2847	$51 \cdot 10^3$	26385	3127	*184.9*	$1652\,8 \cdot 10^7$	2572	

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] 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. مدل-های 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

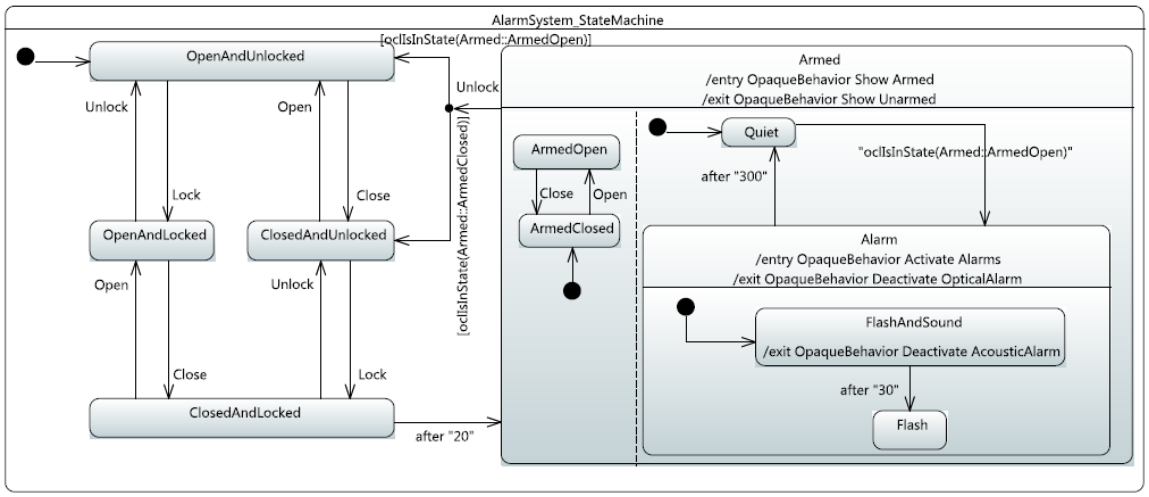
[19] Traces

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]. مدل M2 حلقه کنترل (از جمله رفع خطا) یک کنترل کننده اجرای بسته بارکننده است. کنترل کننده مقدار انحراف دسته[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]، در [3]

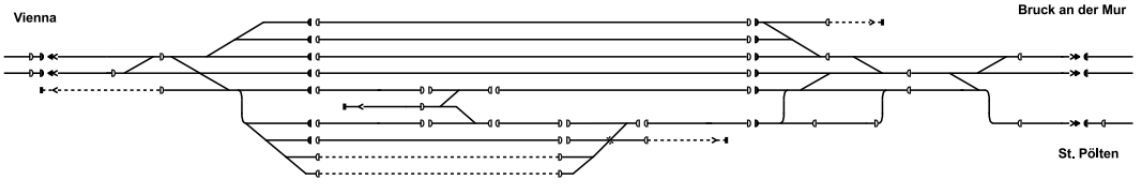
[20] Loader
[21] Joystick
[22] Use case

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.

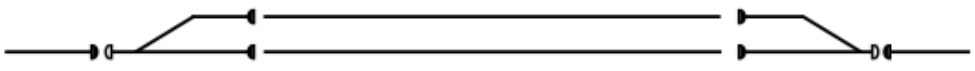
M3- دستگاه اندازه‌گیری. M3 یک مدل از یک پروتکل کنترل از راه دور دستگاه اندازه‌گیری دود اگزوز گرفته شده از صنعت است. یافته‌های اولیه ما از نسل مورد تست برای M3 در زمینه صنعتی، قبلا در [2] منتشر شده است. در اینجا، ما از همان مدل با موتور تولید مورد تست جدیدمان استفاده می‌کنیم. از نظر پیچیدگی مطرح شده، 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 [۲۳] 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

[23] UML-call-trigger

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 گیگابایت 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

			# Mutants						% Kills	
Model	Strat.	Workers	Total	Found	Exec-Err	Killed	Skipped	Given Up	Total	Found
	S1	24		303	15	272	0	16	88.3	94.4
M1	S1	40	323	303	15	267	0	21	86.7	92.7
	S2	40		303	15	261	0	27	84.7	90.6
	S1	24		1089	77	850	111	51	63.4	90.0
M2	S1	40	1417	*856*	*37* (5)	*310*	*484*	*25*		
	S2	40		*996*	*27* (3)	*319*	*625*	*25*		
	S1	24		1145	25	1103	8	9	86.7	98.5
M3	S1	40	1297	1145	25	1098	0	22	86.3	98.0
	S2	40		1145	25	1116	14	19	87.7	99.6
	S1	24		810	21	781	0	8	69.7	99.0
M4	S1	40	1142	810	21	775	0	14	69.1	98.2
	S2	40		835	21	801	0	13	71.5	98.4
	S1	24		635	55 (9)	389	0	191	26.8	67.1
M5	S1	40	1505	724	108 (39)	426	0	190	30.5	69.2
	S2	40		725	90 (22)	474	0	161	33.5	74.6
	S1	24		802	23	661	76	42	32.7	84.9
M6	S1	40	2044	802	23 (2)	654	41	84	32.4	84.0
	S2	40		832	23 (2)	680	40	89	33.7	84.1
	S1	24		1601	30	1496	47	28	42.8	95.2
M7	S1	40	3524	1601	37 (11)	1495	2	67	42.9	95.6
	S2	40		1751	37 (11)	1637	12	65	46.9	95.5

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[۲۴] 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.

[24] Not-yet-encountered mutants based

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] 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]

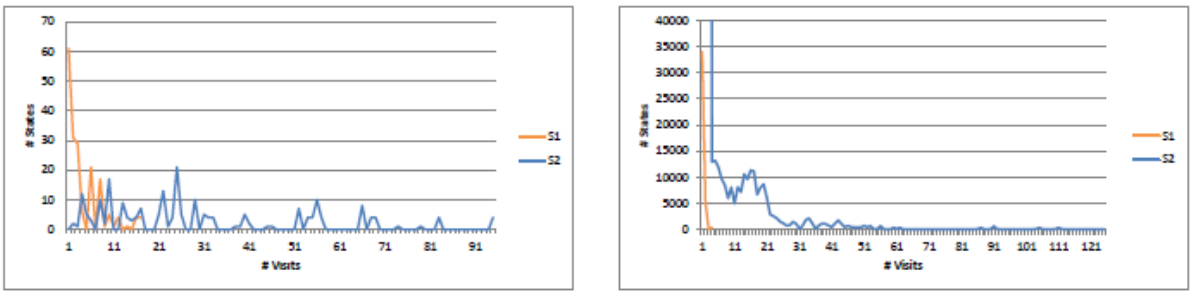
	Time (h:m:s)		S1: Killed in Step#					S2: Killed in Step#				
Model	S1	S2	0-4	5-9	10-14	15-19	20	0-4	5-9	10-14	15-19	20
M1	00:01:21	00:01:44	241	3	23	0	0	255	6	0	0	0
M2	*19:30:22*	-	*850*	*0*	*0*	*0*	*0*	-	-	-	-	-
M3	00:02:32	00:05:20	1094	4	0	0	0	1109	0	7	0	0
M4	00:02:18	00:03:16	767	0	0	8	0	788	8	0	5	0
M5	00:32:49	00:30:18	400	16	8	0	2	447	3	24	0	0
M6	00:13:24	00:22:34	647	1	4	2	0	675	2	2	0	1
M7	17:09:24	22:37:33	1493	2	0	0	0	1633	2	2	0	0

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 می‌

[25] Dead code
[26] Kill

توانیم 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[۲۷] 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.



(a) M1 (b) M7

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

[27] CPU clocking lower

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 MoMuT،S1 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 داده-ها 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[۲۸]. (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[۲۹] تحت نظارت نزدیک و با تنظیمات مختلف دوباره اجرا می‌کنیم، و فایل‌های ورود برای خروجی که ما نمی‌توانیم توضیح دهیم اسکن شده و تحقیقات مورد نیاز.

[28] Threats to validity
[29] Outliers

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] 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] عالی تست جهش در [8] اشاره کنیم.

[30] Pre-pass
[31] Survey

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 صنعتی 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 مطالعات 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، روش-ها 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 نمایندگی 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.