دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

گروه نرم‌افزار

آزمون مبتنی بر خاصیت توصیف‌های تابعی

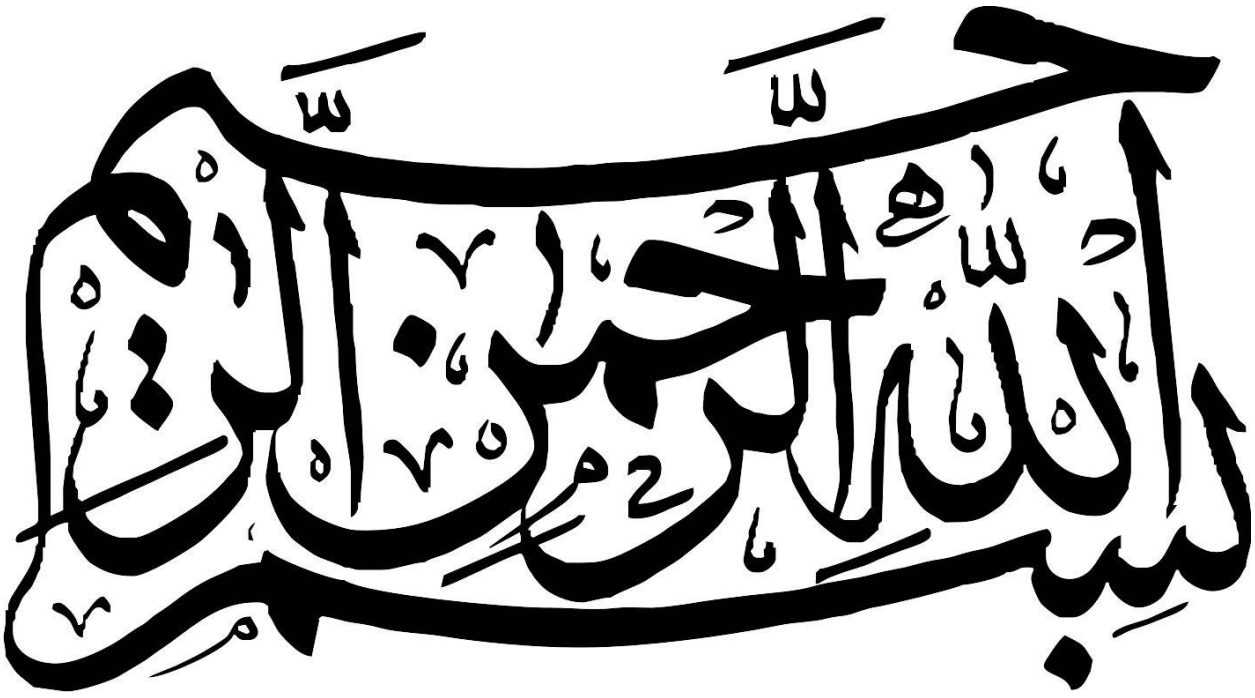
پروژه کارشناسی مهندسی کامپیوتر تمرکز سیستم‌های نرم‌افزاری

مهدی جهانی

استاد راهنما

دکتر رامتین خسروی

تیر ۱۴۰۱


بسم الله الرحمن الرحيم

تعهدنامه اصالت اثر

این‌جانب مهدی جهانی تأیید می‌کنم که مطالب مندرج در این پروژه حاصل کار پژوهشی خودم بوده و به دستاوردهای پژوهشی دیگران که در این نوشته از آن‌ها استفاده شده است مطابق مقررات ارجاع گردیده است. هم‌چنین این پروژه قبلاً برای احراز هیچ مدرک هم‌سطح یا بالاتری ارائه نشده است.

قدردانی

سپاس خداوندگار حکیم را که با لطف بی‌کران خود، آدمی را به زیور عقل آراست.

در آغاز وظیفه خود می‌دانم از زحمات و حمایت‌های بی‌دریغ استاد راهنمای خود، جناب آقای دکتر خسروی، صمیمانه تشکر و قدردانی کنم که در طول انجام این پروژه با صبوری و همدلی راهنما و یاریگر من بودند. بی شک و اغراق بدون حمایت همدلانه و راهنمایی‌های ارزنده ایشان، این مجموعه به انجام نمی‌رسید.

از آقایان آروین ذاکریان و هادی صفری که دوستانه و مشفقانه در قسمت‌های مختلف زحمت مشاوره و کمک در این پروژه را تقبل فرمودند کمال امتنان را دارم.

سپاس‌گزارم از دوستان گران‌مایه‌ام که حامی و پشتیبان من بودند و با هم‌فکری مرا صمیمانه و مشفقانه یاری داده‌اند؛ علی‌الخصوص الهام و هیثم عزیز.

بوسه می‌زنم بر دستان خداوندگار مهر و مهربانی، مادر عزیزم که همواره دل‌سوزانه همراه من بوده و بدون چشم‌داشت و منت از هیچ تلاشی برای حمایت از من فروگذار نکرده است. خدا را شکر می‌کنم که همواره حمایت او را در زندگی داشته‌ام.

و در پایان خدا را سپاس می‌گویم و ستایش می‌کنم که تمام این نعمت‌ها را به من ارزانی داشته و این فرصت را در اختیار من داد تا بیاموزم.

مهدی جهانی

تیر ۱۴۰۱

فهرست مطالب

فهرست تصاویر

چکیده

در روش آزمون مبتنی بر توصیف‌های تابعی، منطق کارکرد سیستم تحت آزمون[۱] در قالب زبان‌های برنامه‌نویسی تابعی مدل می‌شود و پس از آن به کمک روش آزمون جست‌وجو محور آزمایه‌هایی به طور خودکار تولید می‌شوند. مانند تمام روش‌های آزمون مبتنی بر توصیف[۲]، کیفیت این روش در گرو صحت و اعتبار توصیف انجام شده است. برای اعتبارسنجی توصیف انجام شده می‌توان از روش‌های اثبات صوری استفاده کرد که این روش‌ها نیازمند تخصص بالایی در حوزه اثبات درستی سیستم‌ها دارد. روش جایگزینی که در این پروژه مورد بررسی قرار گرفته است، استفاده از آزمون مبتنی بر خاصیت[۳] است. در این روش همانند روش‌های اثبات، خاصیت‌هایی که صحت سیستم را تعریف می‌کنند در قالب فرمول‌های منطقی توصیف می‌شوند اما به جای اثبات آن‌ها، یک آزمونگر خودکار به صورت تطبیقی موارد آزمونی را طراحی و روی توصیف اجرا می‌کند تا به طور نسبی از برقراری خاصیت‌های ارائه‌شده اطمینان حاصل کنیم. این پروژه بر توصیف خاصیت‌های یاد شده در قالب فرمول‌های منطقی و حصول اطمینان نسبی از برقراری آن‌ها با کمک آزمونگر خودکار تمرکز دارد. توصیف تابعی مورد مطالعه در این پروژه توصیف هسته معاملات بورس است که در زبان تابعی هسکل[۴] انجام شده و برقراری سه ویژگی اساسی در آن نشان داده می‌شود.

کلمات کلیدی برنامه‌نویسی تابعی، آزمون نرم‌افزار، کوییک‌چک، توصیف تابعی، آزمون مبتنی بر توصیف

---

[۱] System Under Test (SUT)

[۲] Specification-based Testing

[۳] Property-based Testing

[۴] Haskell

فصل ۱: مقدمه

۱.۱ انگیزش

با توجه به همه‌گیری استفاده و گستردگی مقیاس خدمات نرم‌افزاری در دنیای امروز، یک سامانه نرم‌افزاری (اپلیکیشن موبایل یا دسکتاپ، وب‌سایت و انواع سرویس‌ها) باید ویژگی‌های متعددی را با کیفیت بالا داشته باشد. به عنوان مثال یک کاربر اپلیکیشنی مثل تپسی علاوه بر این که انتظار دارد بتواند درخواستی برای دریافت سرویس حمل‌ونقل دریافت کند، انتظار دارد که اپلیکیشن به صورت روان اجرا شده، هر زمان که احتیاج دارد در دسترس بوده، منابع گوشی را بیش از اندازه استفاده نکرده و در یک کلام علاوه بر این که از لحاظ منطق کارکرد خطایی ندارد، از لحاظ چگونگی اجرای خواسته‌های کاربر هم کم‌خطا باشد.

یا مثلاً یک سامانه تطبیق معاملات سهام را در نظر بگیرید که نه تنها باید درخواست‌ها را در زمان پاسخ بسیار کوتاه ارائه کند، بلکه در زمان‌های خاص باید حجم زیادی از درخواست‌ها قبل از ددلاین معینی مدیریت کرده و در عین حال همواره در دسترس باشد و تراکنش‌های کاربران را به‌صورت مطمئن، امن و قابل اعتماد انجام دهد.

در چنین سرویس‌هایی که ما آن‌ها را «سرویس نرم‌افزاری با توانمندی ویژه[۵]» یا به اختصار «سِنتو» می‌نامیم، فرآیند پردازش معمولاً به تعداد زیادی درخواست[۶] یا تراکنش با طول عمر کوتاه افراز می‌شود[۱]. یک سنتو همواره باید ویژگی‌هایی که مرتبط با معماری نرم‌افزار هستند را داشته باشد؛ مثل عملکرد بهینه، مقیاس پذیری و در دسترس بودن. علاوه بر چنین انتظاراتی سایر الزامات غیرکارکردی[۷] از جمله امنیت را هم باید برآورده کند. اگر همان سامانه تطبیق معاملات سهام را در نظر بگیریم می‌بینیم در چنین سیستمی اطمینان از صحت عملکرد به اندازه برآوردن نیازهای غیرکارکردی مهم است.

---

[۵] این عبارت معادل High Capability Software Service بوده و از منبع [۱] وام گرفته شده است.

[۶] Request

[۷] Non-functional

برقرار کردن چنین ویژگی‌هایی معمولاً به منطق کارکرد سنتو ارتباطی نداشته و نیاز به بهره‌گیری از ابزارهایی دارد که با قسمت غیرکارکردی دامنه سر و کار دارد. با این حال برآورده ویژگی‌های مورد انتظار باعث حجیم‌تر شدن قسمت غیرکارکردی سنتو شده که نهایتاً روی ویژگی‌های کارکردی و عملکرد منطقی آن نیز تاثیر می‌گذارد. در چنین حالتی حتی اگر قبلاً با آزمون واحد[۸] از صحت عملکرد منطق دامنه مطمئن شده باشیم، باز هم باید آزمون‌های سراسری روی سنتو اجرا کرده تا از صحت عملکرد آن در وضعیت فعلی - که المان‌های غیرمرتبط با منطق دامنه به آن اضافه شده‌اند – اطمینان حاصل کنیم.

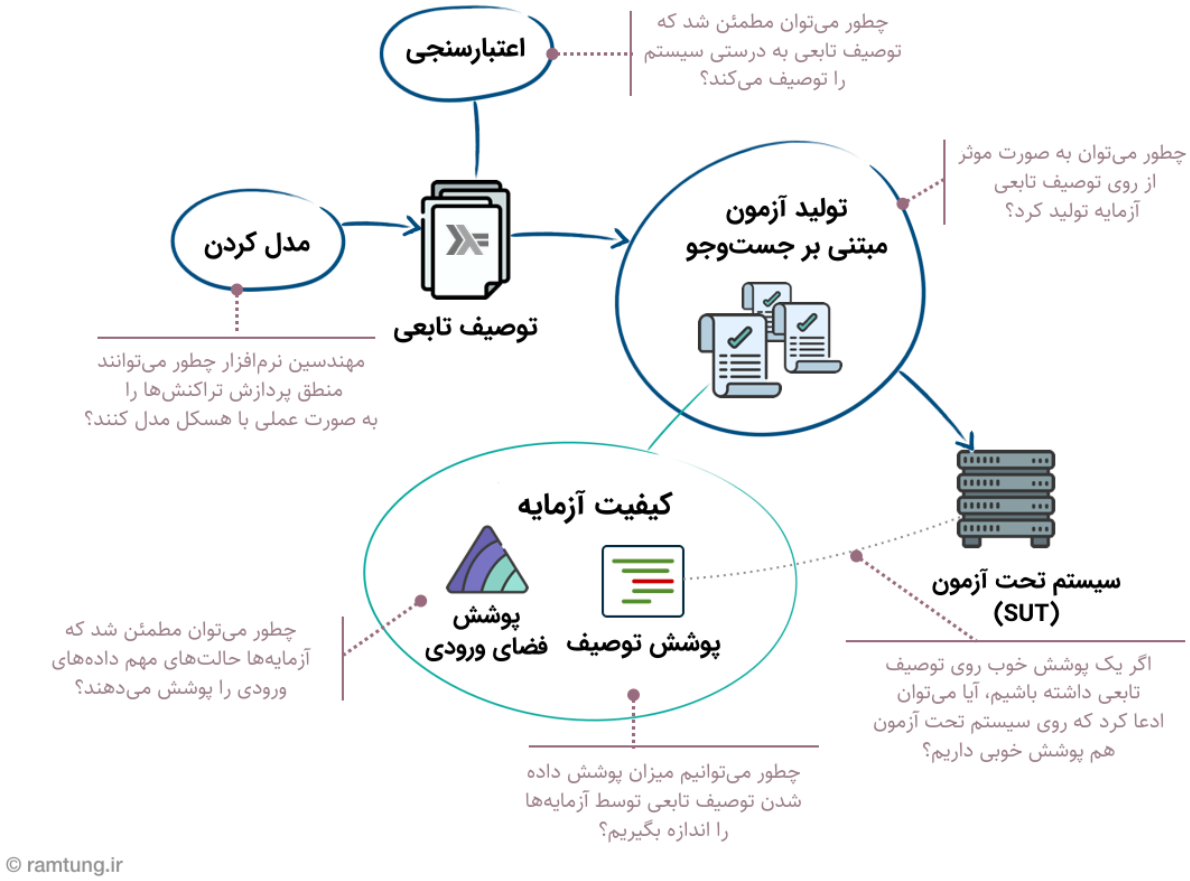
برای تست کردن یک سنتو از روش آزمون مبتنی بر توصیف استفاده می‌شود. این روش بدون احتیاج به داشتن دانش خاصی از جزئیات پیاده‌سازی سنتو عمل کرده و به همین جهت برای طیف وسیعی از سنتوها قابل استفاده است[۲]. چنین رویکردی به دو جهت مورد استفاده قرار می‌گیرد. یکی این که هدف اطمینان از عملکرد صحیح سنتو در وضعیتی است که پیش از این توصیف کردیم. بنابراین، ممکن است به موارد آزمایشی نسبتاً بزرگی نیاز داشته باشیم تا سناریوهای مختلف ممکن را بررسی کنیم. از این رو، تولید اوراکل آزمون به صورت دستی و پوشش دادن حداکثری حالت‌های مختلف امکان پذیر نیست. دوم این که نیاز داریم تا حد خوبی از مورد اعتماد بودن عملکرد سیستم مطمئن باشیم و بدانیم که معیارها و ویژگی‌های مورد نظر ما از سیستم پوشش داده شده‌اند. متأسفانه از آن جایی که کد منبع سنتو نسبتاً بزرگ است و منطق دامنه بسیار با منطق غیر دامنه مخلوط شده است، تعریف یک متریک پوشش مناسب بر روی کد پیاده‌سازی‌شده عملاً غیر ممکن است. بنابراین، توصیف تابعی به عنوان مبنایی منظم برای تولید آزمایه[۹] نیز عمل می‌کند[۲].

حال که قرار است از توصیف تابعی برای آزمون سراسری یک سنتو استفاده کنیم، یک چالش مهم اطمینان حاصل کردن از صحت توصیف است. اگر قرار باشد توصیف تابعی رفتار سامانه اصلی را مدل کند، باید مطمئن باشیم که خود این مدل ویژگی‌های مد نظر را دارد؛ چرا که خروجی‌های توصیف تابعی قرار است به ما کمک کند که تعداد زیادی آزمایه تولید کرده و به کمک آن یک اوراکل برای سنتو داشته باشیم.

---

[۸] Unit Test

[۹] Test Case



تصویر ۱ نمای کلی از اجزای تست مبتنی بر توصیف یک سیستم

۲.۱ رویکرد پیشنهادی

همان‌طورکه در قسمت قبل بررسی شد، مانند تمام روش‌های آزمون مبتنی بر توصیف، کیفیت روش مورد اشاره در گرو صحت و اعتبار توصیف انجام شده است. یک راه برای اعتبارسنجی و اثبات درستی توصیف انجام شده بهره‌گیری از روش‌های اثبات صوری است[۳]. در چنین روشی با بهره‌گیری از خاصیت‌های زبان‌های تابعی و بهره‌گیری از منطق ریاضی، به صورت عینی ثابت می‌کنیم که یک ویژگی برقرار است یا خیر. استفاده از چنین روشی نیازمند تخصص بالایی در حوزه اثبات درستی سیستم‌ها دارد.

به این ترتیب روش جایگزینی که در این پروژه مورد بررسی قرار گرفته است، استفاده از آزمون مبتنی بر خاصیت است. در این روش همانند روش‌های اثبات، خاصیت‌هایی که صحت سیستم را

تعریف می‌کنند در قالب فرمول‌های منطقی توصیف کرده‌ایم اما به جای این که به صورت ریاضی آن‌ها را اثبات کنیم، بررسی می‌کنیم که در تعداد بسیار بسیار زیادی از حالت‌هایی ممکن برقرار باشند. به این صورت که یک آزمونگر خودکار به صورت تطبیقی موارد آزمونی (حالت‌های ممکن) را طراحی و روی توصیف اجرا می‌کند و به این ترتیب می‌توانیم به طور نسبی از برقراری خاصیت‌های ارائه‌شده اطمینان حاصل کنیم.

فصل ۲: مروری بر پژوهش‌های انجام شده

همان‌طور که پیش‌تر اشاره کرده و در ادامه بیشتر توضیح خواهیم در این پروژه بر روی توصیف تابعی هسته معاملات بورس که با زبان تابعی هسکل پیاده‌سازی شده است کار می‌کنیم. برای تولید آزمایه‌های خودکار و بررسی حالت‌های ممکن، از ابزار کوییک‌چک[۱۰] خواهیم کرد[۴]. به این ترتیب در این بخش نگاهی به یکی از پژوهش‌های مرتبط این حوزه که الهام‌بخش این پروژه بوده‌اند خواهیم داشت.

۱.۲ آزمون سامانه آتوسار[۱۱] با استفاده از کوییک‌چک

آتوسار یک استاندارد در حال تکامل برای نرم‌افزارهای نهفته[۱۲] در وسایل نقلیه است که توسط صنعت خودرو تعریف شده و توسط بسیاری از سازندگان مختلف پیاده سازی شده است. یک خودرو مدرن ممکن است در بخش‌های مختلف دارای بیش از ۱۰۰ پردازنده باشد که هر یک نسخه‌ای به‌خصوص از نرم‌افزار آتوسار را از تامین کنندگان مختلف اجرا می‌کند. تمامی این بخش‌ها باید به صورت هماهنگ با هم ارتباط برقرار کرده و کار کنند تا در نهایت خودرو بتواند به درستی کار کند.

در پژوهش انجام شده توسط توماس آرتس[۱۳]، جان هیوز[۱۴] و همکاران بیش از سه هزار صفحه از مشخصات متنی به مدل‌های کوییک‌چک ترجمه شده و بسیاری از پیاده‌سازی‌های مختلف با استفاده از حجم زیادی از آزمایه‌های تصادفی تولید شده از این مدل‌ها آزمایش شدند. این پژوهش در نهایت منجر به پیدا کردن بیش از دویست مشکل شد. مقایسه رویکرد این پژوهش با رویکردهای دستی قدیمی نشان می‌دهد که چنین رویکردی کارآمدتر، مؤثرتر و صحیح‌تر است [۶].

روش توسعه‌داده شده در این پژوهش بر اساس ایجاد یک پیکربندی بزرگ بوده که تمام ویژگی‌هایی را که سازنده اصلی نرم‌افزار – به عنوان فرد خبره‌ی دامنه - به آن‌ها علاقه دارد پوشش

---

[۱۰] QuickCheck (برای اطلاعات بیشتر به منبع [۴] و [۵] مراجعه کنید)

[۱۱] AUTOSAR (AUTomotive Open System ARchitecture)

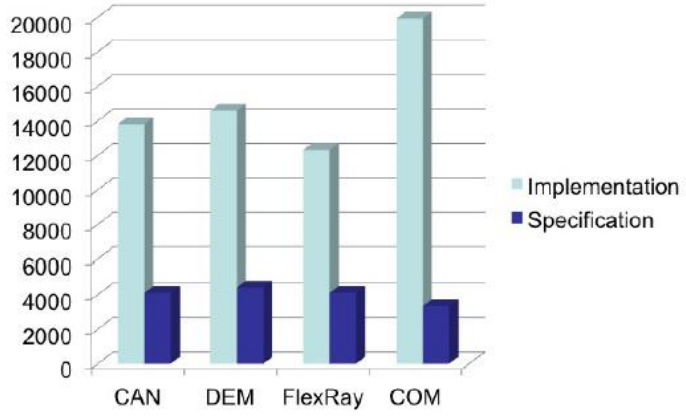
[۱۲] Embedded

[۱۳] Thomas Arts

[۱۴] John Hughes

می‌دهد. این بدان معناست که هر توسعه‌دهنده که نسخه‌ای از آتوسار را ساخته است کافی است فقط یک بار برای آزمایش نرم‌افزار خود را پیکربندی کند. سپس نرم افزار پیکربندی و کامپایل شده به عنوان یک کتابخانه C برای آزمون ارسال می‌شود. سپس با تطابق نزدیک بین کد مدل و ویژگی‌های توصیف شده از سیستم در اسناد، مدل‌های کوییک‌چک برای هر یک از ماژول‌های موجود در کُد منبع توسعه داده شدند. این مدل‌های کوییک‌چک را می‌توان به عنوان مشخصات رسمی قابل اجرا مشاهده کرد. هر مدل می‌تواند برای تولید آزمایه‌های تصادفی برای ماژول مشخص شده استفاده شود[۶].

استفاده از این رویکرد و تست کردن هر یک از ماژول‌ها با بهره‌گیری از آزمایه‌های خودکاری که توسط کوییک‌چک تولید شده بودند، چنین نتایجی را به دنبال داشت: چیزی در حدود سه هزار صفحه اسناد که استانداردها و ویژگی‌های آتوسار را توضیح می‌دادند، در مجموعاً بیست‌هزار خط کُد کوییک‌چک توصیف شدند. توصیف کوییک‌چک ۶ نسخه مختلف از آتوسار که توسط ۶ سازنده مختلف تولید شده بود و مجموعاً شامل یک میلیون خط کد سی[۱۵] می‌شد را پوشش داد. آزمون‌های انجام شده چیزی در حدود دویست مشکل را پیدا و ریشه‌یابی کرد که حدود نیمی از آن‌ها نه به کارکرد سیستم که به تعاریفی که در استاندارد آمده بود برمی‌گشت. بررسی‌های دست‌اندرکاران این پروژه هم‌چنین نشان داد که چنین نتایجی نسبت به رویکردهای متداول آزمون نرم‌افزار نه برابر وقت کمتری مصرف کرد [۷].



تصویر ۲ مقایسه تعداد خط کد سی برای پیاده‌سازی هر ماژول و تعداد خط کد کوییک‌چک برای توصیف ویژگی‌های آن [۶]

[۱۵] C

Module	TTCN3 lines of code	TTCN3 nr of tests	QuickCheck lines of code	QuickCheck time to generate 100 tests
CanIf	16930	65	1978	24 sec
CanSM	6751	17	1255	10 sec
CanNm	12318	58	1716	47 sec
CanTp	21984	105	2062	20 sec
cluster	57983	245	7011	33 sec

تصویر ۳ مقایسه حجم کد مورد نیاز برای تست در رویکرد TTCN3 (استاندارد خود آتوسار) و کوییک‌چک

فصل ۳: روش تحقیق

۱.۳ تعریف مسئله

همان‌طور که پیش از این اشاره کردیم، مسئله این پروژه نشان دادن برقراری سه ویژگی خاص برای توصیف تابعی هسته معاملات بورس است. در ادامه به تعریف اجزای مختلف مسئله می‌پردازیم.

۱.۱.۳ توصیف تابعی هسته معاملات بورس

سیستمی که آزمون مبتنی بر خاصیت روی آن پیاده‌سازی می‌شود، توصیف تابعی هسته معاملات بورس است که با زبان هسکل نوشته شده است[۸]. این سیستم با بهره‌گیری از الگوی آذینگر[۱۶] توسعه داده شده و کُد منبع آن مطابق تصویر ۴ دارای سه بخش اصلی است؛ دامنه، زیرساخت[۱۷] (ابزار) و آذینگر[۱۸]ها. پیکربندی پروژه با استفاده از کابال[۱۹] انجام شده و چهار فایل دیگری که در کنار کُد منبع قرار دارند، حالت‌های اجرای مختلف را توصیف می‌کنند.

قسمت دامنه، کُدهای مربوط به منطق کارکرد و تعاریف دامنه را در بر می‌گیرد. به طور خاص ME.hs تمامی تعاریف مربوط به ساختارهای داده و کارکردها و تابع‌های پایه‌ی سیستم را شامل می‌شود. ما از این فایل به عنوان مرجعی برای تعریف آزمایه‌های خودکار و هم‌چنین آزمون عملکرد سیستم استفاده می‌کنیم.

قسمت آذینگرها کارکردهای مختلف سیستم را در بر می‌گیرد. از بین همه‌ی کارکردهایی که برای سیستم تعریف شده‌اند، ما با OrderHandler.hs سروکار داریم که وظیفه مدیریت کردن سفارش جدیدی که به سیستم وارد شده است را ایفا می‌کند. در این پروژه آزمون را روی عملکرد این آذینگر اجرا کرده‌ایم.

---

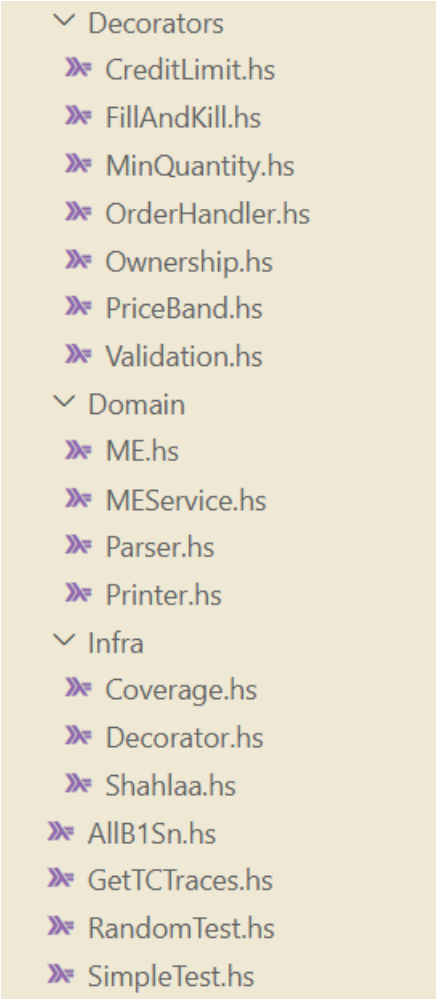
[۱۶] Decorator Pattern

[۱۷] Infra

[۱۸] Decorator

[۱۹] Cabal (برای اطلاعات بیشتر رجوع کنید به منبع [۹])

قسمت زیرساخت امکاناتی برای کار با سیستم ارائه می‌دهد که به منطق عملکرد آن ارتباطی نداشته و در این پروژه از آن‌ها هیچ استفاده‌ای نشده است.



تصویر ۴ ساختار پروژه توصیف تابعی هسته معاملات بورس

کارکردی که در این پروژه آن را آزموده‌ایم، مدیریت و تطابق[۲۰] یک سفارش محدود[۲۱] که به سیستم وارد شده است، با دفترچه سفارش[۲۲]هایی است که از مرحله قبل در سیستم موجود بوده است. برای درک بهتر این موضوع یک مثال از این فرآیند می‌زنیم.

---

[۲۰] Match

[۲۱] Limit Order

[۲۲] Order Book

۲.۱.۳ مدیریت و تطابق سفارش محدود

«سفارش محدود» یکی از چندین نوع سفارش است که معمولاً توسط سیستم‌های مشابه پشتیبانی می‌شود. سفارش محدود سفارشی است برای خرید یا فروش سهام با قیمتی خاص یا بهتر. یک سفارش خرید را فقط می‌توان با یک سفارش فروش که قیمتی کمتر یا مساوی قیمت حدی[۲۳] سفارش خرید دارد معامله کرد. به همین ترتیب یک سفارش فروش را فقط با یک سفارش خرید که قیمتی بیشتر یا مساوری قیمت حدی سفارش فروش دارد می‌توان تطبیق داد. در ساده‌ترین شکل، یک سفارش محدود شامل شناسه سفارش، نماد سهام، قیمت حد، مقدار و هم‌چنین شناسه‌های سهام‌دار و یا کارگزار صادرکننده سفارش است. یک دستور محدود ممکن است فوراً اجرا نشود، زیرا ممکن است دستوری از جنس مخالف با قیمت منطبق پیدا نشود. در چنین حالتی سفارش وارد دفترچه سفارش‌ها می‌شود[۲].

دفترچه سفارش شامل دو صف اولویت‌دار[۲۴] جداگانه از سفارشات در سیستم است، یکی برای سفارش‌های خرید و دیگری برای سفارش‌های فروش. سفارشات خرید با بالاترین قیمت و سفارشات فروش با کمترین قیمت بالاترین اولویت را در صف‌های مربوط به خود دارند. سفارشات با رتبه یکسان با توجه به زمان ورود اولویت بندی می‌شوند. نمونه تطبیق یک سفارش خرید را می‌توانید در تصویر ۵ مشاهده کنید[۲].

۳.۱.۳ ویژگی‌های مورد بررسی

با توجه به اطلاعات جمع‌آوری شده از خبرگان دامنه، سه ویژگی برای کارکرد تطابق سفارش محدود انتخاب شد. این ویژگی‌ها را می‌توان به برخی کارکردهای دیگر سیستم نیز تأمیم داد.

---

[۲۳] Limit Price

[۲۴] Priority Queue

دفترچه خرید قبل از تطبیق

خرید			فروش		
ID	Price	Qty	ID	Price	Qty
1	50	500	4	55	500
2	40	1000	5	60	300
3	40	800	6	70	1000

سفارش خرید ورودی

ID	Price	Qty
7	60	1200

دفترچه خرید بعد از تطبیق

خرید			فروش		
ID	Price	Qty	ID	Price	Qty
7	60	400	6	70	1000
1	50	500			
2	40	1000			
3	40	800			

معاملات انجام شده

BID	SID	Price	Qty
7	4	55	500
7	5	60	300

تصویر ۵ نمونه‌ای از یک سفارش محدود. یک سفارش با شناسه ۷ وارد سیستم شده و قسمتی از آن با دو سفارشی که در سر صف فروش قرار دارند، تطبیق و معامله می‌شود. به این ترتیب دو معامله صورت گرفته و باقی‌مانده این سفارش نیز به صف خرید اضافه می‌شود.

- **ویژگی بقای مقدار[۲۵]**

طبق این ویژگی حاصل جمع مقدار[۲۶] سفارش ورودی و مجموع مقدارهای سفارش‌های موجود در دفترچه سفارشات قبل از تطبیق سفارش ورودی برابر است با مجموع مقدارهای سفارش‌های موجود در دفترچه بعد از تطبیق سفارش ورودی به اضافه دو برابر مجموع مقدارهای معاملات انجام شده. برای درک بهتر توصیف ریاضی زیر را در نظر بگیرید.

فرض کنید که صف خرید و فروش قبل از ورود سفارش جدید buyQueue و sellQueue باشند. هم‌چنین سفارش ورودی را ord_{in} در نظر می‌گیریم.

$$\text{buyQueue} = \{b_1, b_2, b_3, \dots, b_m\}$$
$$\text{sellQueue} = \{s_1, s_2, s_3, \dots, s_n\}$$
$$\text{inputOrder} = \text{ord}_{\text{in}}$$

[۲۵] Conservation of Quantity

[۲۶] Quantity

حال فرض کنید که سفارش ورودی تطبیق پیدا کرده، صف‌های خرید و فروش جدید به ترتیب buyQueue’ و sellQueue’ بوده و tradesList لیست معاملات انجام شده هم باشد. اگر فرض کنیم qty(x) برابر با مقدار اوردر x باشد، به این ترتیب باید داشته باشیم:

$$\text{buyQueue}' = \{b'_1, b'_2, b'_3, \dots, b'_k\}$$
$$\text{sellQueue}' = \{s'_1, s'_2, s'_3, \dots, s'_l\}$$
$$\text{tradesList} = \{t_1, t_2, t_3, \dots, t_r\}$$
$$\text{qty(x)} = \text{Quantity of Order x}$$
$$\sum_{i=1}^{m} \text{qty}(b_i) + \sum_{i=1}^{n} \text{qty}(s_i) + \text{qty}(\text{ord}_{in}) = \sum_{i=1}^{k} \text{qty}(b'_i) + \sum_{i=1}^{l} \text{qty}(s'_i) + 2 \times \sum_{i=1}^{r} \text{qty}(t_i)\ (*)$$

برقراری ویژگی فوق را به کمک استقرای قوی روی مجموع تعداد سفارش‌های موجود در صف خرید و فروش قبل از تطبیق سفارش ورودی (یعنی $m+n$) می‌توان نشان داد. پایه استقرا برای زمانی که هر دو صف خالی باشند یا این که هر کدام حداکثر یک سفارش داشته باشند نشان داد. کافی است به این نکته توجه کرد که اگر دو سفارش a, b با یکدیگر تطبیق پیدا کنند و معامله trd بین آن‌ها انجام شده و دو سفارش a', b' از آن‌ها باقی بماند، داریم:

$$\text{qty}(a) + qty(b) = qty(a') + qty(b') + 2 \times qty(trd)(**)$$

برای گام استقرا نیز کافی است آخرین معامله‌ای که انجام می‌شود را در نظر بگیریم. بر اساس فرض استقرا می‌دانیم قانون پایستگی مقدار برای دفترچه سفارش بدون در نظر گرفتن دو سفارشی که در انتها با یکدیگر تطبیق پیدا کرده‌اند برقرار است. از طرفی طبق فرمول (**) این قانون برای آخرین معامله‌ای که انجام می‌شود نیز برقرار است. با کنار هم گذاشتن دو فرمول (**) و (*) می‌توانیم نتیجه بگیریم که این قانون برای حالتی که آن را در نظر گرفته‌ایم برقرار بوده و به این ترتیب همواره برقرار است.

$$\sum_{i=1}^{m} \mathrm{qty}(b_i) + \sum_{i=1}^{n} \mathrm{qty}(s_i) + \mathrm{qty}(\mathrm{ord_{in}})$$
$$= \sum_{i=1}^{k} \mathrm{qty}(b'_i) + \sum_{i=1}^{l} \mathrm{qty}(s'_i) + 2 \times \sum_{i=1}^{r} \mathrm{qty}(t_i)\ (*)$$

$$\mathrm{qty}(a) + qty(b) = qty(a') + qty(b') + 2 \times qty(trd)(**)$$

$$(*), (**) \Rightarrow \left[\sum_{i=1}^{m} \mathrm{qty}(b_i)\right] + qty(b) + \left[\sum_{i=1}^{n} \mathrm{qty}(s_i)\right] + \mathrm{qty}(a) + \mathrm{qty}(\mathrm{ord_{in}})$$
$$= \left[\sum_{i=1}^{k} \mathrm{qty}(b'_i)\right] + qty(b') + \left[\sum_{i=1}^{l} \mathrm{qty}(s'_i)\right] + qty(a')$$
$$+ 2 \times \left[\sum_{i=1}^{r} \mathrm{qty}(t_i)\right] + 2 \times qty(trd)$$

- **عدم تطبیق سر صف خرید و سر صف فروش بعد از تطبیق سفارش ورودی جدید**

طبق این ویژگی هنگامی که یک سفارش جدید وارد سیستم شده و تطبیق آن انجام شده و پایان می‌پذیرد، سفارشی که سر صف خرید است، نباید بتواند با سفارشی که سر صف فروش است تطبیق و معامله شود. این ویژگی از این جهت برقرار است که فرض می‌کنیم هنگام مدیریت یک سفارش ورودی، وقتی پروسه تطبیق پایان می‌پذیرد، دیگر هیچ معامله‌ای برای انجام شدن نباید وجود داشته باشد. چرا که اگر امکان انجام معامله جدید وجود می‌داشت، فرآیند تطبیق نباید خاتمه می‌یافت.

- **درصورتی که سفارش ورودی از نوع خرید باشد، قیمت آخرین معامله انجام شده باید از قیمت سفارشی که در سر صف فروش وجود دارد بیشتر باشد. به همین ترتیب درصورتی که سفارش ورودی از نوع فروش باشد، قیمت آخرین معامله انجام شده باید از قیمت سفارشی که در سر صف خرید وجود دارد کمتر باشد.**

فرض کنید که سفارش ورودی یک سفارش خرید باشد. با توجه به توضیحاتی که در بخش ۲.۱.۳ دادیم می‌دانیم که این سفارش با سفارش‌هایی که در صف فروش هستند، معامله می‌شود. از طرفی می‌دانیم سفارشات فروش با کمترین قیمت بالاترین اولویت را در صف خود دارند. به این ترتیب سفارش‌های فروش از سر صف به ته صف به صورت صعودی بر اساس قیمت مرتب شده‌اند. فرض کنید یک سفارش خرید وارد شده و یکی یکی با سفارش‌هایی که در صف ورود هستند، تطابق پیدا کرده و معامله می‌شود.

طبق تصویر ۵ می‌دانیم قیمت یک معامله برابر با قیمت سفارشی است که در صف قرار دارد. به این ترتیب فرض کنیدسفارش خرید ورودی p باشد. فرض کنید s آخرین سفارشی باشد که قسمتی از p با آن معامله می‌شود. قیمت این معامله را t در نظر گرفته و فرض کنید s' سفارش بلافاصله بعد از سفارش s در صف فروش باشد. می‌دانیم که قیمت s برابر با t است چرا که قیمت سفارش صف در قیمت معامله ثبت می‌شود. از طرفی با توجه به صعودی بودن قیمت‌ها در صف فروش قیمت s' حتماً از قیمت s و بنابراین حتماً از قیمت t بیشتر خواهد بود. استدلالی مشابه را درباره حالتی که سفارش ورودی از نوع فروش باشد می‌توان ارائه کرد.

۲.۳ پیاده‌سازی راه‌حل

پیاده‌سازی راه‌حل و بررسی برقراری ویژگی‌های یاد شده در بخش ۳.۱.۳ در سه گام انجام شد. با توجه به عدم آشنایی قبلی با زبان تابعی هسکل، در گام اول اقدام به نوشتن یک توصیف ابتدایی از تابع «تطبیق یک سفارش محدود ورودی با لیستی از سفارش‌ها» به زبان هسکل و بررسی برقراری ویژگی بقای مقدار و چند ویژگی ساده دیگر کردیم[۱۰]. بعد از آشنایی با ویژگی‌ها و قابلیت‌های زبان هسکل و ابزار کوییک‌چک در گام بعدی برقراری ۳ ویژگی یاد شده را روی نسخه‌ی ساده‌تری از سیستم که بدون بهره‌گیری از آذینگرها نوشته شده و تنها کار «تطبیق یک سفارش ورودی با دفترچه سفارش‌ها» را انجام می‌داد بررسی کردیم[۱۱]. در نهایت پس از اطمینان از برقراری این ویژگی روی نسخه ساده‌تر سیستم، ۳ ویژگی را روی تازه‌ترین نسخه‌ی سیستم که در بخش ۱.۱.۳ معرفی شد بررسی نمودیم[۸].

در هر گام از پیاده‌سازی راه‌حل سه مرحله اصلی صورت گرفت: یک توصیف ویژگی موردنظر با توجه به منطق و ویژگی‌های دامنه، دو تعریف فضای حالت برای تولید آزمایه توسط کوییک‌چک و سه بررسی برقراری ویژگی مد نظر در سیستم با استفاده از کوییک‌چک. در کنار این سه مرحله به فراخورد نیاز توابع یا داده‌ساختارهای جدیدی برای راحت‌تر کردن محاسبات نیز تعریف شدند. گام اول کاملاً به صورت جداگانه پیاده‌سازی شد اما گام دو و سه از لحاظ ترتیب زمانی تقریباً هم‌زمان با هم پیش برده شدند. برای همین پیاده‌سازی را برای تازه‌ترین نسخه‌ی سیستم مفصل‌تر بررسی می‌کنیم. در ادامه به توضیح و بسط جزئیات مربوط به پیاده‌سازی و ملاحظات فنی می‌پردازیم.

۱.۲.۳ تعریف فضای حالت برای تولید آزمایه توسط کوییک‌چک

یکی از مهم‌ترین گام‌های هر بخش تعریف فضای حالت برای تولید آزمایه توسط کوییک‌چک بوده و با توجه به اهمیت آن قبل از پرداخت به جزییات پیاده‌سازی راه‌حل در هر گام به صورت جداگانه نگاهی به این قسمت از پیاده‌سازی خواهیم داشت.

به طور کلی زمانی که قرار است با استفاده از کوییک‌چک برقراری یک ویژگی را برای یک تابع بررسی کنیم، نیاز به تولید حالت‌های مختلف از ورودی تابع داریم. کتاب‌خانه کوییک‌چک به صورت پیش‌فرض قابلیت تولید حالت‌های مختلف برای نوع‌داده‌های اصلی هسکل (مثل Int، Float، String، Bool و...) را داراست. به این معنی که می‌توان انتظار داشت که اگر مثلاً ورودی یک تابع از نوع Int باشد، کوییک‌چک می‌تواند تمامی حالت‌های موردنیاز برای Int را تولید کند[۱۴].

اما زمانی که پای نوع‌داده‌های جدید به میان می‌آید، کوییک‌چک نیاز دارد که تعریف فضای حالت را برای نوع‌داده تعریف شده بداند. برای حل کردن این چالش باید از سازنده[۲۷]ها و نوع‌کلاس اربیتری[۲۸] استفاده کرد. به‌طوری کلی سازنده‌ها به ساختن یک متغیر، یا لیستی از متغیرها کمک کرده و بعد از تعریف آن‌ها باید فضای حالت یک نوع‌داده را با نمونه‌گیری[۲۹] از نوع‌کلاس اربیتری به کوییک‌چک فهماند[۱۴]. در هر گام جزییات استفاده از این مفاهیم را مشاهده خواهیم کرد.

---

[۲۷] Generator (برای اطلاعات بیشتر به منبع [۱۵] مراجعه کنید)

[۲۸] Arbitrary

[۲۹] Instantiation

۲.۲.۳ بررسی ویژگی‌ها روی توصیف ابتدایی تابع[۳۰]

در این گام علاوه بر سه مرحله اصلی، تعریف نوع‌داده[۳۱] و نوشتن تابع ابتدایی که کار تطبیق یک سفارش محدود ورودی با لیستی از سفارش‌ها را انجام دهد نیز انجام شد.

- **نوع‌داده‌های تعریف شده**

قبل از هر چیز برای پیاده‌سازی تابع ابتدایی دو نوع‌داده جدید تعریف کردیم که سفارش‌ها و معاملات را توصیف کنند. نوع‌داده Dorder یک سفارش را توصیف می‌کند و پیاده‌سازی آن را در قطعه کد ۱ می‌بینید. این نوع‌داده یک فیلد شناسه، یک فیلد نوع سفارش، یک فیلد آی‌دی سهام، یک فیلد قیمت و نهایتاً یک فیلد مقدار دارد. نوع‌داده SimpleTrade هم یک معامله را توصیف می‌کند. هر معامله یک شناسه خریدار، یک شناسه فروشنده، یک شناسه سهام، یک قیمت و یک مقدار دارد.

```
data Dorder = Dorder {ordid :: Int
                      ,ordtype :: OrderType
                      ,stock_id :: Int
                      ,price :: Float
                      ,quantity :: Int}
                      deriving (Show)
```

قطعه کد ۱ توصیف نوع‌داده سفارش

[۳۰] برای مشاهده کد منبع به منبع [۱۰] مراجعه کنید.

[۳۱] Data Type (برای کسب اطلاعات بیشتر به منبع [۱۲] مراجعه کنید)

```
data Trade = SimpleTrade {buy_id :: Int
                          ,sell_id :: Int
                          ,tstock_id :: Int
                          ,tprice :: Float
                          ,tquantity :: Int}
                    deriving (Show)
```

قطعه کد ۲توصیف نوع داده معامله

فیلد نوع سفارش خود یک نوع‌داده دیگر به اسم OrderType است که پیاده‌سازی آن را در قطعه کد ۳ می‌بینید. هر OrderType یا از نوع خرید است یا از نوع فروش و یا از نوع خالی (در ادامه دلیل تعریف نوع‌داده خالی را ذکر می‌کنیم). این نوع‌داده از نوع‌کلاس Show و Eq نیز مشتق می‌شود. به این معنی که یک متغیر از نوع OrderType می‌تواند در کنسول نمایش داده شده و بررسی برابری دو متغیر از این نوع امکان‌پذیر است[۳۲].

```
data OrderType = Sell | Buy | Empty
    deriving(Eq, Show)
```

قطعه کد ۳ توصیف نوع‌داده جنس سفارش

- **توصیف تابع تطبیق سفارش محدود با دفترچه سفارش‌ها**

برای پیاده‌سازی تابعی که یک سفارش محدود را با لیستی از سفارش‌ها تطبیق دهد، ابتدا یک تابع به اسم monoOrderMatch تعریف کردیم که دو سفارش با هم مقایسه و در صورت امکان تطبیق و معامله می‌کند. خروجی این تابع دو سفارش جدید و یک معامله است. دلیل این که در نوع‌داده OrderType یک حالت خالی تعریف کردیم، برای این بود که وقتی یک سفارش کاملاً با سفارش دیگری تطبیق داده می‌شود، خروجی آن یک سفارش خالی باشد.

قطعه کد ۴ تابع تطبیق دو سفارش

```
monoOrderMatch :: Dorder -> Dorder -> (Dorder, Dorder, Trade)
```

سپس از این تابع کمکی برای پیاده‌سازی تابع اصلی یعنی processBuyOrder استفاده کردیم که کار تطبیق یک سفارش را با لیستی از سفارش‌ها را به صورت بازگشتی انجام می‌دهد.

[۳۲] برای کسب اطلاعات بیشتر به منبع [۱۳] مراجعه کنید.

قطعه کد ۵ توصیف تابع تطبیق سفارش با صفی از سفارش‌ها

```
processBuyOrder :: Dorder -> [Dorder] -> (Dorder, [Dorder],
[Trade])
processBuyOrder (Dorder _ Empty _ _ _) sords = (nullOrder,
sords, []) where nullOrder = Dorder 0 Empty 0 0 0
processBuyOrder bord [] = (bord, [], [])
processBuyOrder bord [sord] = let (remBuyOrder, remSellOrder,
trade) = monoOrderMatch bord sord in (remBuyOrder,
[remSellOrder], [trade])
processBuyOrder bord (sord:sords)
    | ordtype remainBuyOrder == Empty = if ordtype
remainSellOrder == Empty
                                then (remainBuyOrder, sords,
[trade])
                                else (remainBuyOrder,
remainSellOrder:sords, [trade])
    | otherwise = (remainOrder, if ordtype remainSellOrder ==
Empty
                                    then remainOrderBook
                                    else
remainSellOrder:remainOrderBook, trade:trades)
    where (remainBuyOrder, remainSellOrder, trade) =
monoOrderMatch bord sord
          (remainOrder, remainOrderBook, trades) =
processBuyOrder remainBuyOrder sords
          nullOrder = Dorder 0 Empty 0 0 0
```

توابع پیاده‌سازی شده در این گام بهینه نبوده و تفاوت ساختاری با توصیف اصلی هسته معاملات بورس داشتند. مثلاً در توصیف اصلی فرض بر این است که تمامی سفارش‌ها روی یک نماد مشترک انجام شده‌اند اما در این توصیف ابتدایی دیدیم که هر سفارش شناسه سهام داشت به این معنی که سفارش‌ها لزوماً بر روی یک نماد صورت نمی‌گیرند.

- **توصیف ویژگی‌های موردنظر**

ویژگی‌های بررسی شده در این گام ویژگی‌های بسیار ساده‌ای بوده و تنها برای آشنایی با ابزارها انجام شده‌اند اما از بین آن‌ها می‌توان به حالت خاصی از ویژگی بقای مقدار اشاره کرد که در آن مقدار یک سفارش قبل از انجام تطبیق باید با مجموع مقدار معاملات انجام شده و باقی‌مانده‌ی سفارش بعد از تطبیق مساوی باشد. البته نهایتاً این ویژگی روی تابع processBuyOrder با مثال نقض مواجه شد.

قطعه کد ۶ توصیف حالت خاص ویژگی بقای مقدار

```
prop_quantitySum_check buyOrder orderBook =
    quantity buyOrder == (getTradesQuantitySum trades) +
(quantity remainBuyOrder)
    where (remainBuyOrder, remainOrderBook, trades) =
processBuyOrder buyOrder orderBook
```

- **تعریف فضای حالت برای تولید آزمایه توسط کوییک‌چک**

در این گام تعریف فضای حالت برای تولید آزمایه به ساده‌ترین شکل ممکن انجام شد. همان‌طور که در قطعه کد ۶ می‌توان مشاهده کرد، سازنده OrderType تابعی است که می‌گوید یک متغیر دل‌خواه از نوع OrderType عضوی از لیست سه تایی خرید و فروش و خالی است. سپس نمونه‌گیری از نوع‌کلاس اربیتری با استفاده از همین سازنده انجام می‌شود. به این ترتیب کوییک‌چک می‌داند هرگاه بخواهد متغیری از نوع OrderType را بسازد، چطور باید این کار را انجام دهد.

قطعه کد ۷ تعریف فضای حالت برای جنس سفارش

```
instance Arbitrary OrderType where
    arbitrary = elements [Sell, Buy, Empty]
```

تعریف فضای حالت برای Order کمی پیچیده‌تر انجام شد. سازنده این نوع‌داده از چند پیرایندهٔ[۳۳] استفاده شده است. همان‌طور که می‌توان دید، برای برخی از فیلدهای Dorder شروطی تعیین

[۳۳] Modifier (برای اطلاعات بیشتر به منبع [۱۶] مراجعه کنید)

کرده‌ایم. استفاده از پیراینده Positive این قید را اضافه می‌کند که هر گاه قرار باشد برای نوع‌داده Dorder فیلد شناسه تولید کنیم، اطمینان حاصل می‌کنیم که مثبت باشد.

قطعه کد ۸ تعریف فضای حالت برای نوع‌داده سفارش

```
instance Arbitrary Dorder where
   arbitrary = do
     Positive ordid <- arbitrary
     ordtype <- arbitrary
     Positive stock_id <- arbitrary
     Positive price <- arbitrary
     Positive quantity <- arbitrary
     return $ Dorder ordid ordtype stock_id price quantity
```

۳.۲.۳ بررسی ویژگی‌ها روی جدیدترین نسخه سیستم

- **تعریف فضای حالت برای تولید آزمایه توسط کوییک‌چک**

پرچالش‌ترین بخش پروژه تعریف فضای حالت درست برای تولید آزمایه‌ها توسط کوییک‌چک بود. برای پیاده‌سازی این قسمت ما ۴ رویکرد متفاوت را پیش گرفتیم که در ادامه به چالش‌ها و ویژگی‌های هر یک اشاره خواهیم کرد. پیش از آن مروری روی نوع‌داده‌های این نسخه از سیستم خواهیم داشت.

```
type Quantity = Int
type Price = Int
type TimeStamp = Int
type OrderID = Int
type BrokerID = Int
type ShareholderID = Int
type CreditInfo = Map.Map BrokerID Int
type OwnershipInfo = Map.Map ShareholderID Int
```

قطعه کد ۹ تایپ‌های تعریف شده

قطعه کد ۱۰ نوع‌داده سفارش و نوع‌داده جنس سفارش (ما در این بخش تنها از LimitOrder استفاده کردیم)

```
data Side = Buy | Sell deriving (Show, Eq, Ord)

data Order = LimitOrder
    { oid         :: OrderID
    , brid        :: BrokerID
    , shid        :: ShareholderID
    , price       :: Price
    , quantity    :: Quantity
    , side        :: Side
    , minQty      :: Maybe Quantity
    , fillAndKill :: Bool
    } | IcebergOrder
    { oid          :: OrderID
    , brid         :: BrokerID
    , shid         :: ShareholderID
    , price        :: Price
    , quantity     :: Quantity
    , side         :: Side
    , minQty       :: Maybe Quantity
    , fillAndKill  :: Bool
    , disclosedQty :: Quantity
    , visibleQty   :: Quantity
    } deriving (Show, Eq)
```

قطعه کد ۱۱ نوع‌داده دفترچه سفارش‌ها و صف سفارش

```
type OrderQueue = [Order]

data OrderBook = OrderBook
    { buyQueue  :: OrderQueue
    , sellQueue :: OrderQueue
    } deriving (Show, Eq)
```

- رویکرد ساده، شبیه به بخش ۲.۲.۳

در این رویکرد ما شبیه به بخش ۲.۲.۳ یک نمونه‌گیری از نوع‌کلاس اربیتررِی انجام داده و از هیچ سازنده‌ی دل‌خواهی برای تولید متغیرهایی از نوع‌داده Order و یا OrderBook استفاده نکردیم. تنها از پیاینده استفاده کردیم که مطمئن باشیم مقادیری مثل قیمت سفارش، مقدار سفارش یا شناسه سفارش عدد منفی نباشند.

در این حالت هیچ یک از ویژگی‌های بررسی شده برقرار نبوده و فراتر از آن در بسیاری از موارد فضای حالت تولید شده هیچ یک از خواص مورد نظر سیستم را نداشت. به عنوان مثال طبق تعریف در یک متغیر از نوع Order مقدار minQty باید کمتر یا مساوی quantity باشد. در حالی که در این حالت این ویژگی برقرار نبود.

- تولید تصادفی دفترچه سفارش‌ها

در این مرحله سازنده‌های شخصی‌سازی‌شده برای نوع‌داده Order طراحی شده و توانستیم برخی از ویژگی‌های پیش‌فرض دامنه را رعایت کنیم. از جمله این که مقدار minQty حتماً از quantity کوچک‌تر باشد. برای این کار تابع سازنده genQtyandMinQty را طبق قطعه کد ۱۲ ساختیم.

قطعه کد ۱۲ تعریف تابع سازنده مقدار و مقدار کمینه

```
genQtyandMinQty :: Gen (Quantity, MinimumQuantity)
genQtyandMinQty = elements list
    where list = [(a, Nothing) | a <- [1..1000]] ++ [(a, Just
b) | a <- [1..1000], b <- [1..1000], a >= b]
```

در این حالت ولی کماکان برخی دیگر از ویژگی‌های دامنه برقرار نبودند. به عنوان مثال نحوه ساخت دفترچه سفارش به صورت تصادفی بود. به این معنی که نه تنها سفارش‌ها در صف‌های خرید و فروش اولویت‌بندی شده نبودند، بلکه کوییک‌چک مثال‌هایی تولید می‌کرد که در آن‌ها در صف خرید سفارشی از جنس فروش داشتیم و برعکس در صف فروش سفارش‌هایی از جنس خرید.

- تولید دفترچه سفارش‌ها به صورت مرتب‌شده

با ایجاد تغییرات در حالت قبلی مشکل تولید صف خریدی که در آن سفارش فروش وجود داشت (و برعکس) حل شد. برای حل این مشکل از سازنده‌هایی که در مرحله قبل نوشته بودیم و کار آن‌ها

تولید سفارش خرید یا یک سفارش فروش بود استفاده کرده و صف‌هایی ساختیم که تنها از یک جنس سفارش در آن‌ها قرار داشته باشد.

قطعه کد ۱۳ تابع تولیدکننده سفارش از جنس فروش

```
genOnlySellOrder :: Gen Order
genOnlySellOrder = do --Positive oid <- arbitrary
                      --Positive brid <- arbitrary
                      --Positive shid <- arbitrary
                      (oid, brid, shid) <- genIDs
                      price <- genPrice
                      sellSide <- genOnlySellSide
                      (quantity , minQty) <- genQtyandMinQty
                      fillAndKill <- genFillAndKill
                      return (LimitOrder oid brid shid price
quantity sellSide minQty fillAndKill)
```

در این حالت ویژگی بقای مقدار برقرار بوده ولی برای دو ویژگی دیگر مثال نقض تولید می‌شد. برای حل این مشکل تصمیم گرفتیم که تابع سازنده صف خرید و فروش را طوری تغییر دهیم که صف مرتب تولید کند.

قطعه کد ۱۴ تابع سازنده صف خرید و فروش مرتب‌شده

```
isSellQueueSorted :: OrderQueue -> Bool
isSellQueueSorted sellQ = sellQ == sortOrderQueue sellQ

genSellQueue :: Gen OrderQueue
genSellQueue = listOf genOnlySellOrder `suchThat`
isSellQueueSorted

genOrderBook :: Gen OrderBook
genOrderBook = do buyQ <- genBuyQueue
                  sellQ <- genSellQueue
                  --let buyQ2 = sortBuyQueue buyQ
                  return (OrderBook buyQ sellQ)
```

این حالت هر سه ویژگی مورد نظر را برقرار می‌کرد اما تست کردن ویژگی مدت زمان بسیار زیادی طول می‌کشید (جزییات نتایج را در قسمت ۴ بررسی می‌کنیم).

- تولید دفترچه سفارش‌ها به صورت ارگانیک

تمام تلاش‌های مراحل پیشین ما را به جایی رساند که دیگر دفترچه سفارش‌ها را با تعریف فضای حالت تولید نکرده و آن را به صورت ارگانیک به دست بیاوریم. در این رویکرد به جای این که یک سازنده برای OrderBook تعریف کرده و یک نمونه‌گیری از اربیتری برای آن انجام دهیم، یک لیست از سفارش‌ها را به صورت تصادفی تولید کرده و از روی آن یک دفترچه سفارش می‌ساختیم. در این حالت دیگر نیازی به نمونه‌گیری از کلاس اربیتری برای نوع‌داده OrderBook نخواهیم داشت.

```
genRandomOrder :: Gen Order
genRandomOrder = do --Positive oid <- arbitrary
                    --Positive brid <- arbitrary
                    --Positive shid <- arbitrary
                    (oid, brid, shid) <- genIDs
                    price <- genPrice
                    side <- genBothSides
                    (quantity , minQty) <- genQtyandMinQty
                    fillAndKill <- genFillAndKill
                    return (LimitOrder oid brid shid price
quantity side minQty fillAndKill)

genOrderList :: Gen OrderList
genOrderList = listOf genRandomOrder
```

قطعه کد ۱۵ تابع سازنده لیستی از سفارش‌های تصادفی

این لیست سفارش‌ها به تابع matchListOfOrders داده می‌شود که این تابع از یک دفترچه سفارش خالی شروع کرده و به صورت بازگشتی سفارش‌های موجود در لیست تصادفی را با دفترچه سفارش تطبیق و معامله‌ها را انجام داده و ذخیره می‌کند. خروجی این تابع دفترچه سفارشی است که منطقاً تمامی ویژگی‌های پیش‌فرض دامنه را باید داشته باشد. از جمله مرتب بودن صف‌های خرید و فروش براساس اولویت‌هایی که پیش از این تعریف کردیم. نهایتاً با استفاده از این روش در تولید دفترچه سفارش هر ۳ ویژگی برقرار بوده و تست کردن آن‌ها هم در مقایسه با حالت قبلی بسیار سریع‌تر انجام شد.

```
matchListOfOrders :: OrderList -> OrderBook -> [Trade] ->
(OrderBook, [Trade])
matchListOfOrders [] ob trds = (ob, trds)
matchListOfOrders [ord] ob trds = let (newOb, newTrds) =
matchNewOrder' ord ob in (newOb, trds ++ newTrds)
matchListOfOrders (ord:remOrdsList) ob trds =
let (newOb, newTrds) = matchNewOrder' ord ob
updatedTrds = trds ++ newTrds
in matchListOfOrders remOrdsList newOb updatedTrds
```

قطعه کد ۱۶ توصیف تابع matchListOfOrders

- **توصیف ویژگی‌های موردنظر**

در این گام بررسی ویژگی‌ها بر روی تابع matchNewOrder صورت گرفت که در OrderHandler.hs قرار داشته و کار تطبیق سفارش جدید با دفترچه سفارش‌ها را انجام می‌دهد. این تابع یک سفارش جدید و یک دفترچه سفارش‌ها به عنوان ورودی گرفته و یه دفترچه سفارش‌های جدید به همراه لیست معاملات انجام شده را باز می‌گرداند. با توجه به نوع خروجی نسخه اصلی این تابع که با توجه به نیازمندی‌های سیستم از نوع Coverage (OrderBook, [Trade]) تعریف شده بود، با توجه به نیازمندی این پروژه تغییر کرد.

قطعه کد ۱۷ نسخه اصلی و نسخه تغییریافته تابع matchNewOrder

```haskell
matchNewOrder :: Order -> OrderBook -> Coverage (OrderBook,
[Trade])
matchNewOrder o ob = do
    let oq = oppositeSideQueue o ob
    (remo, oq', ts) <- match o oq
    let ob' = updateOppositeQueueInBook o oq' ob
    let ob'' = enqueue remo ob'
    (ob'', ts) `covers` "MNO"

matchNewOrder' :: Order -> OrderBook -> (OrderBook, [Trade])
matchNewOrder' o ob = do
    let oq = oppositeSideQueue o ob
    let (remo, oq', ts) = match' o oq
    let ob' = updateOppositeQueueInBook o oq' ob
    let ob'' = enqueue remo ob'
    (ob'', ts)
```

با توجه به این تغییرات، تعریف یکی دو تابع دیگر نیز تنها در حد اصلاح نوع خروجی تابع تغییر داده شد. با توجه به این تغییرات، سه ویژگی یاد شده در بخش ۳.۱.۳ به صورتی که در ادامه می‌آید توصیف شدند. توجه کنید که با توجه به مواردی که در قسمت تعریف فضای حالت اشاره کردیم و توضیح دادیم که سازنده‌های در طی زمان چه تغییراتی داشتند، در این‌جا آخرین توصیفی که برای هر یک از این ویژگی‌ها نوشتیم را ارائه می‌کنیم. توصیف‌های دیگر این ویژگی از منطق مشابه پیروی می‌کند با این تفاوت که نوع دست‌یابی به دفترچه سفارش‌ها در آن متفاوت است[۳۴].

در پیاده‌سازی آزمون برای هر یک از ویژگی‌ها یک تابع برقرار بودن ویژگی مد نظر را چک کرده و یک تابع هم برای تست کردن ویژگی با استفاده از کوییک‌چک به کار می‌رود. تابع دوم با کلمه prop شروع می‌شود. دقت کنید که عبارت collect (length trades) در تابع‌هایی که با prop شروع می‌شوند، این کاربرد را دارد که در هر حالت اندازه لیست معاملات انجام شده را جمع‌آوری کرده و نهایتاً با نتیجه آزمون ارائه می‌کند.

[۳۴] برای اطلاعات بیشتر به منبع [۸] مراجعه کنید.

- ویژگی بقای مقدار

قطعه کد ۱۸ توصیف ویژگی بقای مقدار در حالتی که دفترچه سفارش را به صورت ارگانیک تولید کنیم

```
newQuantitySumEquityCheck :: Order -> OrderList -> Bool
newQuantitySumEquityCheck newOrder newOrderList = quantity
newOrder + getOrderBookQuantitySum organicOrderBook ==
                                                    getOrderBookQ
uantitySum finalOrderBook + 2 * getTradesQuantitySum trades
    where (organicOrderBook, organicTrades) = matchListOfOrders
newOrderList primeOb primeTrds
          primeOb = OrderBook [] []
          primeTrds = []
          (finalOrderBook, trades) = matchNewOrder' newOrder
organicOrderBook

prop_newQuantitySumEqual_Classified:: Order -> OrderList ->
Property
prop_newQuantitySumEqual_Classified newOrder newOrderList =
collect (length trades) $ newQuantitySumEquityCheck newOrder
newOrderList
    where (organicOrderBook, organicTrades) = matchListOfOrders
newOrderList primeOb primeTrds
          primeOb = OrderBook [] []
          primeTrds = []
          (finalOrderBook, trades) = matchNewOrder' newOrder
organicOrderBook
```

همان‌طور که دیده می‌شود دفترچه سفارش که برای آزمون تابع matchNewOrder به کار می‌رود، توسط کوییک‌چک تولید نشده و از خروجی تابع matchListOfOrders به دست می‌آید. همان‌طور که توضیح داده شد تابع matchListOfOrders کارش را با لیستی از سفارش‌ها و دفترچه سفارش و لیست معامله خالی آغاز کرده و نهایتاً یک دفترچه سفارش پرشده و لیستی از معاملات را برمی‌گرداند.

قطعه کد ۱۹ توصیف ویژگی بقای مقدار در حالتی که دفترچه سفارش به صورت مرتب توسط خود کوییک‌چک تولید شود

```
quantitySumEquityCheck :: Order -> OrderBook -> Bool
quantitySumEquityCheck newOrder orderBook = quantity newOrder +
getOrderBookQuantitySum orderBook == getOrderBookQuantitySum
remainOrderBook + 2 * getTradesQuantitySum trades
    where (remainOrderBook, trades) = matchNewOrder' newOrder
orderBook

prop_quantitySumEqual_Classified:: Order -> OrderBook ->
Property
prop_quantitySumEqual_Classified newOrder orderBook =
orderBookNotNull orderBook ==> collect (length trades) $
quantitySumEquityCheck newOrder orderBook
    where (remainOrderBook, trades) = matchNewOrder' newOrder
orderBook
```

- عدم تطبیق سر صف خرید و سر صف فروش

قطعه کد ۲۰ توصیف ویژگی عدم تطابق سر صف خرید و فروش

```
canHeadsMatchGeneral :: OrderBook -> Bool
canHeadsMatchGeneral orderBook
    | null (buyQueue orderBook) = True
    | null (sellQueue orderBook) = True
    | otherwise = ordersCantBeMatched buyHead sellHead
    where buyHead = head $ buyQueue orderBook
          sellHead = head $ sellQueue orderBook

prop_newCanHeadsMatch :: Order -> OrderList -> Property
prop_newCanHeadsMatch newOrder newOrderList = collect (length
trades) $
                                              canHeadsMatchGene
ral finalOrderBook
    where (organicOrderBook, organicTrades) = matchListOfOrders
newOrderList primeOb primeTrds
          primeOb = OrderBook [] []
          primeTrds = []
          (finalOrderBook, trades) = matchNewOrder' newOrder
organicOrderBook
```

- مقایسه قیمت آخرین معامله با سر صف خرید یا فروش

قطعه کد ۲۱ توصیف ویژگی وضعیت قیمت آخرین معامله نسبت به سر صف خرید و فروش

```
tradePriceMoreThanBuyLessThanSell' :: Trade -> OrderBook ->
Side -> Bool
tradePriceMoreThanBuyLessThanSell' trd ob side
    | null (sellQueue ob) && side == Buy = True
    | null (buyQueue ob) && side == Sell = True
    | otherwise = if side == Buy
                  then tradePrice <= sellHeadPrice
                  else tradePrice >= buyHeadPrice
    where buyHeadPrice = price (head $ buyQueue ob)
          sellHeadPrice = price (head $ sellQueue ob)
          tradePrice = priceTraded trd

prop_newTradePriceCompareWithHeads_Classified :: Order ->
OrderList -> Property
prop_newTradePriceCompareWithHeads_Classified newOrder
newOrderList = collect (length trades) $

       null trades ||

       tradePriceMoreThanBuyLessThanSell' lastTrade
finalOrderBook lastSide
    where (organicOrderBook, organicTrades) = matchListOfOrders
newOrderList primeOb primeTrds
          primeOb = OrderBook [] []
          primeTrds = []
          (finalOrderBook, trades) = matchNewOrder' newOrder
organicOrderBook
          lastTrade = last trades
          lastSide = side newOrder
```

- **دیگر نکات پیاده‌سازی**

همان‌طور که پیش‌تر اشاره شد، در نسخه نهایی توصیف هسته معاملات بورس از الگوی آذینگر استفاده شده و پروژه دارای قسمت‌های مختلف بود. با توجه به ساختار پروژه، تمامی موارد مربوط به آزمون در یک ماژول اجرایی با نام AllB1Sn قرار گرفتند. این ماژول تمامی تعاریف موردنیاز از ماژول‌های دیگر را با دستور import دریافت می‌کند. جدا قرار گرفتن کدهای آزمون علاوه بر بالا بردن خوانایی و نگهداری کد باعث جلوگیری از تداخل با منطق دامنه می‌شود.

تصویر ۶ یادآوری ساختار پروژه و تعاریف import شده در فایل آزمون

۴.۲.۳ بررسی ویژگی‌ها روی نسخه ساده‌تر سیستم

پیاده‌سازی این بخش بسیار شبیه به بخش ۳.۲.۳ بوده و تفاوت آن با قسمت ۳.۲.۳ در این است که این قسمت پایه‌ی منطق عملکردی نسخه پیشرفته‌تر است با این تفاوت که در آن خبری از آذینگرها و قابلیت‌های غیرمرتبط با دامنه نیست. ما از این توصیف برای بررسی خالص منطق عملکرد سیستم استفاده کردیم. به عنوان مثال در برخی از آزمون‌هایی که با کوییک‌چک روی نسخه

پیشرفته‌تر انجام می‌شد، ما به برخی استثناها[۳۵] در مسیر برنامه برخورد می‌کردیم. بررسی‌ها روی مثالی که کوییک‌چک تولید کرده و به ایجاد استثنا منجر شده بود نشان می‌داد که آن حالت خاص با منطق عملکرد سیستم هماهنگ است. راه ما برای این که بفهمیم آن حالت خاص با منطق عملکرد سیستم هماهنگی دارد این بود که همان مثال را روی حالت ساده‌تر سیستم امتحان کنیم. این کار باعث شد که تغییراتی که در OrderHandler.hs انجام دادیم را بازنگری کرده و برخی از حالت‌هایی که در آن‌ها استثنا ایجاد می‌شد را شناسایی و برطرف کنیم.

[۳۵] Exception

فصل ۴: نتایج

۱.۴ نتایج حاصل از بررسی ویژگی‌ها روی توصیف ابتدایی تابع

۱.۱.۴ نتایج حاصل از تست

در این حالت از پیاده‌سازی نتیجه‌ی ویژه‌ای از تست نگرفتیم. خاصیت‌های مورد بررسی در این حالت یا بسیار بدیهی بوده که برقراری آن‌ها به صورت واضحی قابل اثبات بود و یا مثال نقض برای آن‌ها وجود داشت. با این حال این مرحله نتایج ارزشمندی از لحاظ پیاده‌سازی ارائه کرد.

۲.۱.۴ نتایج حاصل از پیاده‌سازی

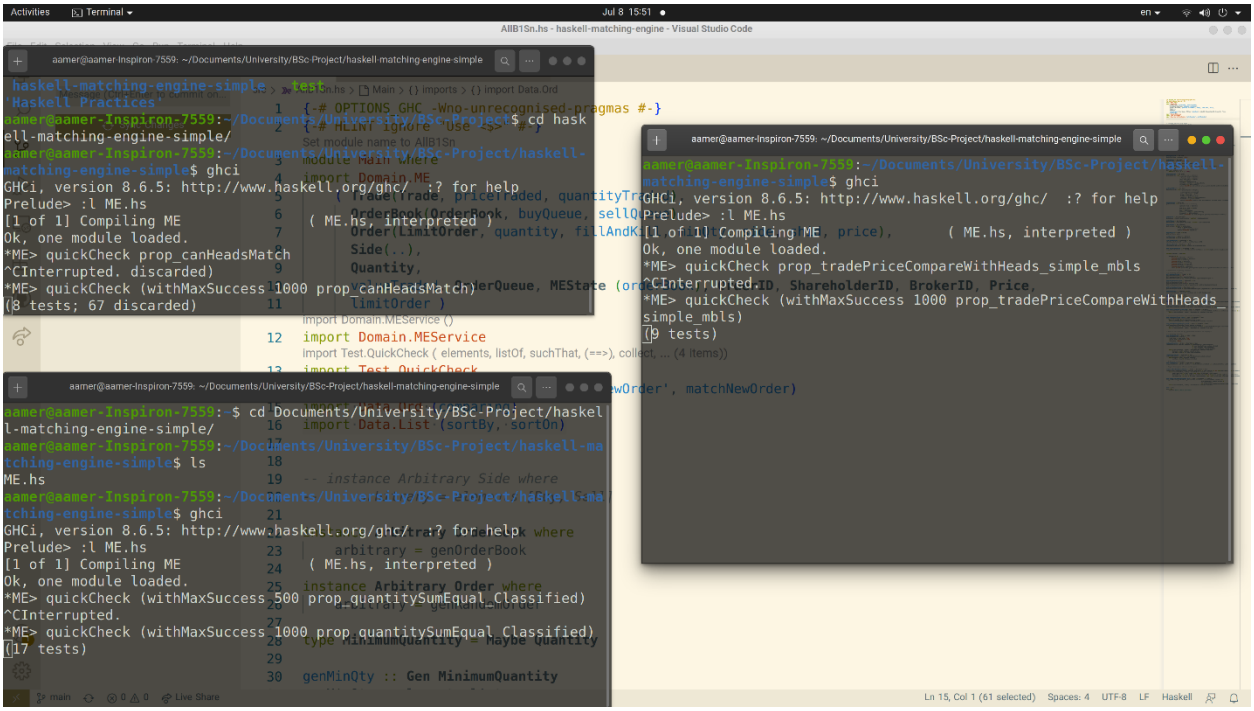
با توجه به عدم آشنایی قبلی با زبان تابعی هسکل و ابزار کوییک‌چک این مرحله از پیاده‌سازی راه‌حل سرنخ‌های ارزشمندی برای آزمون سیستم در مراحل بعد ارائه کرد. از جمله این سرنخ‌ها می‌توان به آشنایی با ویژگی‌ها و کاربردهای کلاس اربیتری، توابع سازنده و تولید فضای حالت برای کوییک‌چک اشاره کرد.

۲.۴ نتایج حاصل از بررسی ویژگی‌ها روی نسخه ساده‌تر سیستم

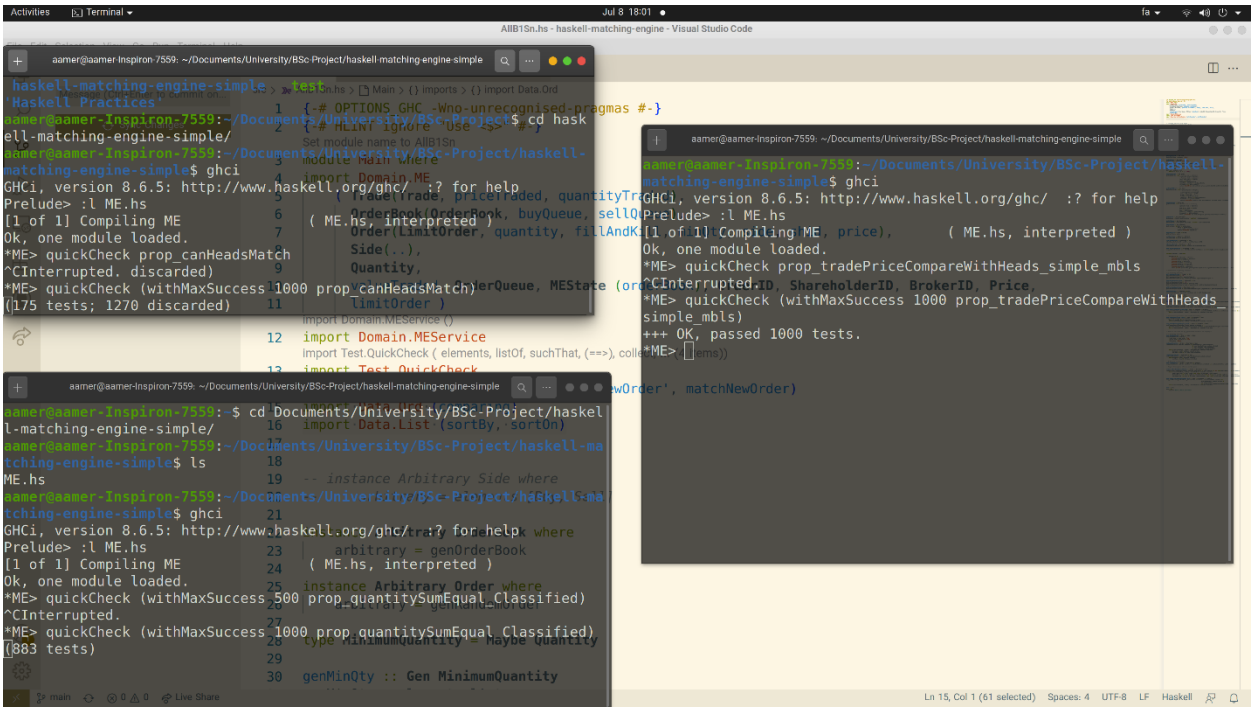
۱.۲.۴ نتایج حاصل از تست

- **تست در حالت تولید دفترچه سفارش به صورت از پیش مرتب شده**

در این حالت ۳ ویژگی یاد شده برقرار بوده اما زمان زیادی برای بررسی درستی آن‌ها صرف شد. مقایسه تصویر ۷ و ۸ نشان می‌دهد که برای بررسی فضای حالت کوچکی تنها برای رسیدن به ۱۰۰۰ مثال موفق، بیش از ۳ ساعت زمان نیاز صرف شده است. این در حالی است که در بخش ۱.۳.۴ برای رسیدن به صد هزار مثال موفق در حالت تولید دفترچه سفارش به صورت ارگانیک حداکثر به ۴ ساعت زمان نیاز داشتیم.



تصویر ۷ شروع تست سه ویژگی در ساعت ۱۵:۵۱



تصویر ۸ اواخر تست سه ویژگی در ساعت ۱۸:۰۱

- **تست در حالت تولید دفترچه سفارش به صورت ارگانیک**

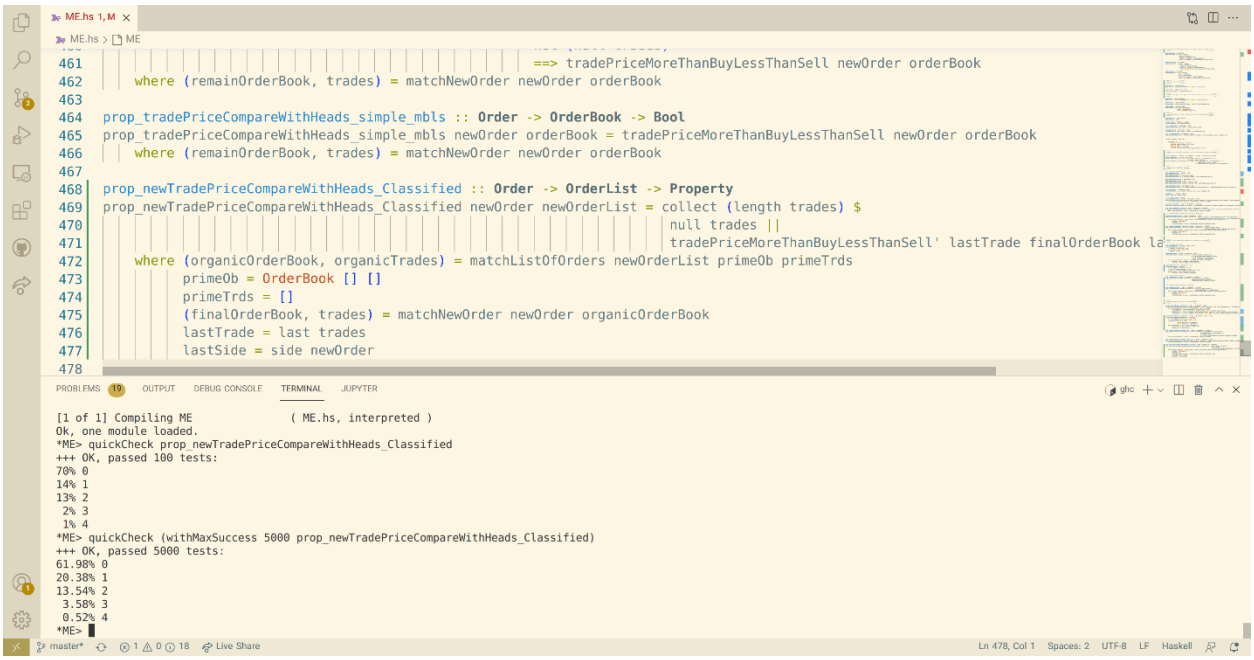
هر سه ویژگی در این حالت برقرار بوده و مدت زمان رسیدن به ۵۰۰۰ مثال موفق نیز، بسیار کمتر از ۳ ساعت بوده است.

تصویر ۹ بررسی برقراری ویژگی بقای مقدار

با توجه به استفاده از متد collect می‌توان در نتایج دید که در ۶۱/۴۴ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰/۴۰ درصد موارد یک معامله، در ۱۴/۴ موارد ۲ معامله و در حدود ۴ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۶ معامله داشته باشد، نداشته‌ایم.

تصویر ۱۰ بررسی ویژگی تطبیق‌پذیری دو سر صف خرید و فروش

با توجه به استفاده از متد collect می‌توان در نتایج دید که در ۶۳ درصد موارد هیچ معامله‌ای انجام نشده، در ۱۹/۷۴ درصد موارد یک معامله، در ۱۳/۰۶ موارد ۲ معامله و در حدود ۴/۳ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۵ معامله داشته باشد، نداشته‌ایم.



تصویر ۱۱ بررسی ویژگی وضعیت قیمت آخرین معامله نسبت به سر صف خرید و فروش

با توجه به استفاده از متد collect می‌توان در نتایج دید که در ۶۱٫۹۸ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰٫۳۸ درصد موارد یک معامله، در ۱۳٫۵۴ موارد ۲ معامله و در حدود ۴٫۳ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۴ معامله داشته باشد، نداشته‌ایم.

۲.۲.۴ نتایج حاصل از پیاده‌سازی

مهم‌ترین نتیجه حاصل از این قسمت از لحاظ پیاده‌سازی پیدا کردن مثال‌هایی بود که روی نسخه ساده‌تر به درستی نتیجه می‌دادند اما روی نسخه جدیدتر باعث ایجاد استثنا می‌شدند. هم‌چنین بررسی حالتی که دفترچه سفارش را به صورت از پیش مرتب شده تولید می‌کردیم، روی این نسخه از سیستم، باعث صرفه‌جویی در زمان شده و سریع‌تر ما را با مشکل تولید صف‌های از پیش مرتب شده روبرو کرد. در نتیجه این مشکل سریع‌تر حل شد.

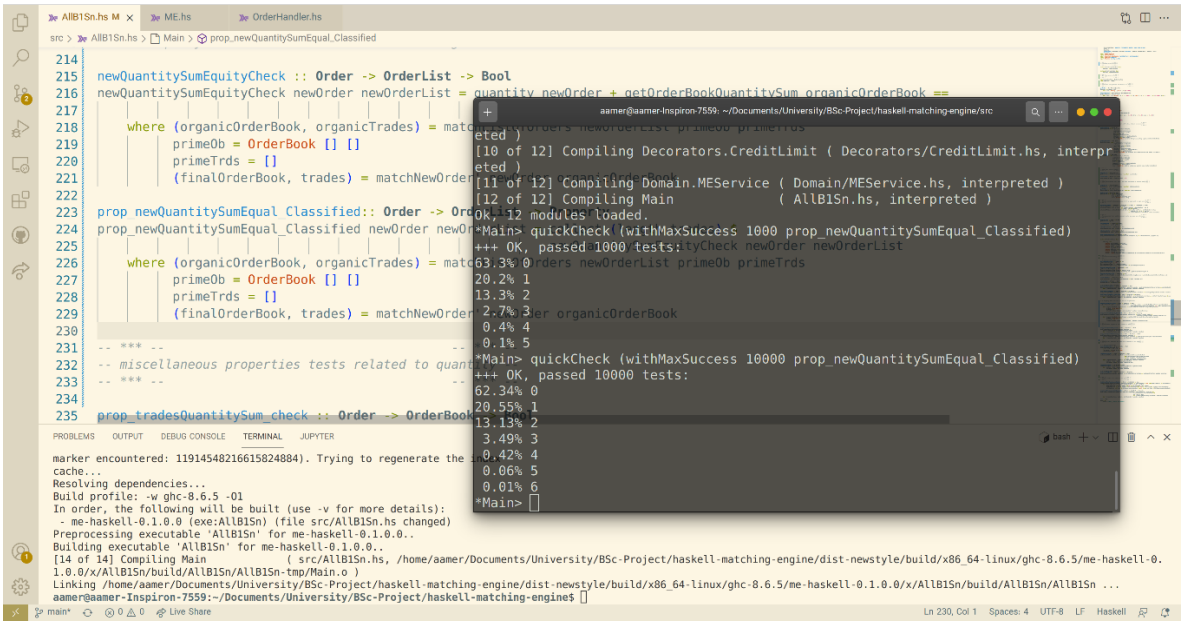
۳.۴ نتایج حاصل از بررسی ویژگی‌ها روی جدیدترین نسخه سیستم

۱.۳.۴ نتایج حاصل از تست

- **تست در حالت تولید دفترچه سفارش به صورت ارگانیک**

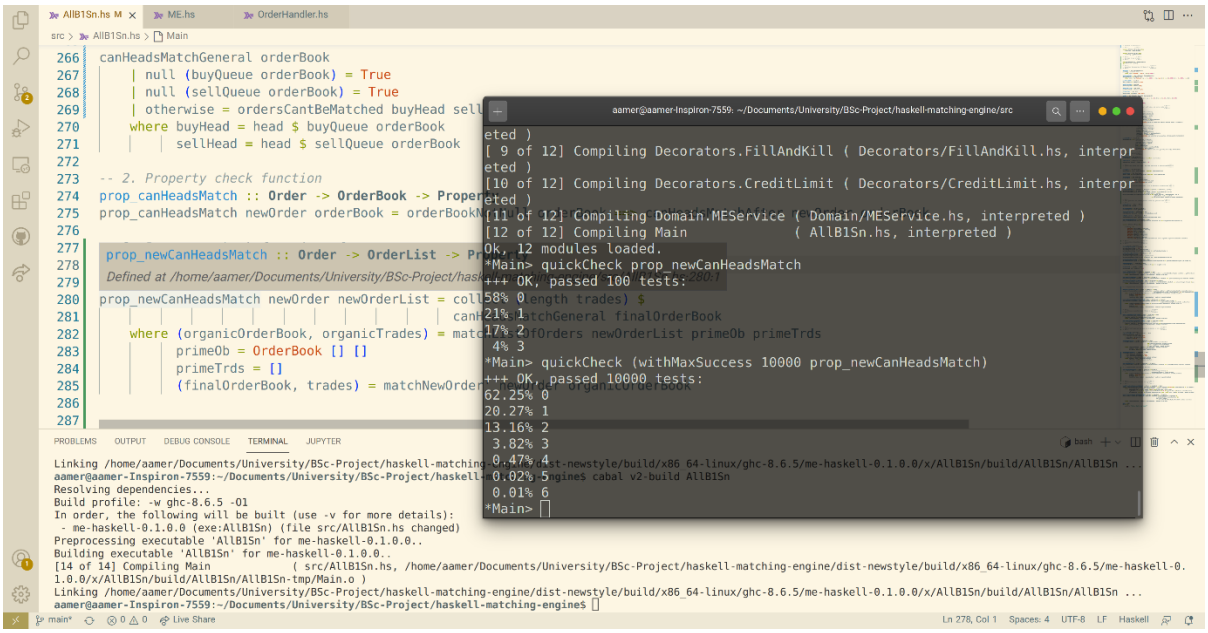
با توجه به این که حالت تولید دفترچه سفارش به صورت از پیش مرتب شده را روی نسخه ساده‌تر آزموده و عدم بهره‌وری آن را بررسی کردیم، در این‌جا تنها نتایج حالت تولید دفترچه سفارش به صورت ارگانیک را بررسی می‌کنیم. بدیهی است که حالت‌های قبل از این دو به دلیل رعایت نکردن پیش‌فرض‌های دامنه، برخی از ویژگی‌ها را برقرار نکرده و معتبر نیستند.

- نتایج آزمون با ده هزار مثال موفق



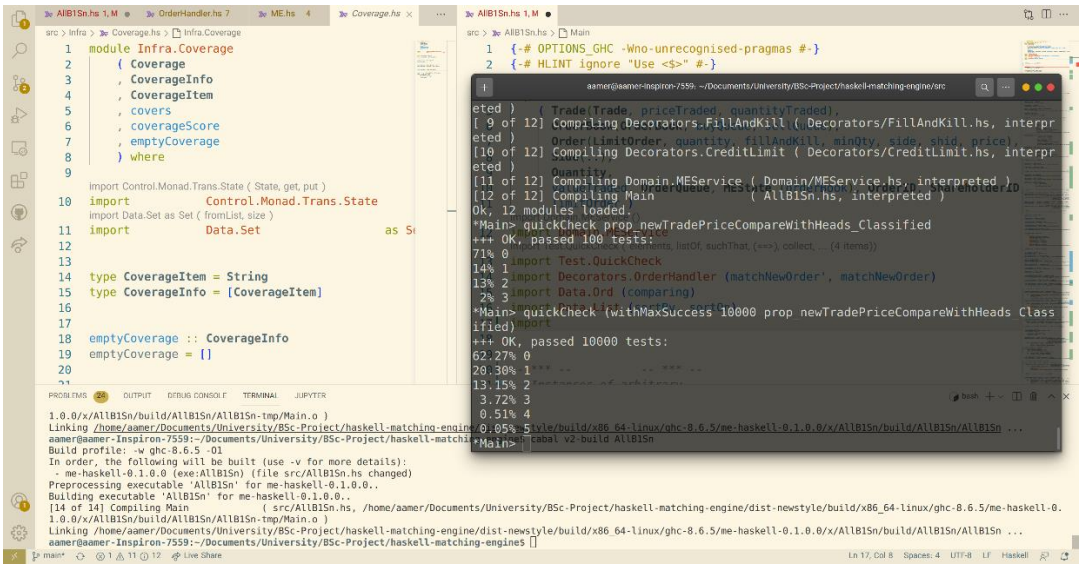
تصویر ۱۲ بررسی ویژگی بقای مقدار با ده هزار مثال موفق

می‌توان در نتایج دید که در ۶۲/۳۴ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰/۵۵ درصد موارد یک معامله، در ۱۳/۱۳ موارد ۲ معامله و در حدود ۴ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۶ معامله داشته باشد، نداشته‌ایم.



تصویر ۱۳ بررسی ویژگی تطبیق‌پذیری سر صف خرید و فروش با ده هزار مثال موفق

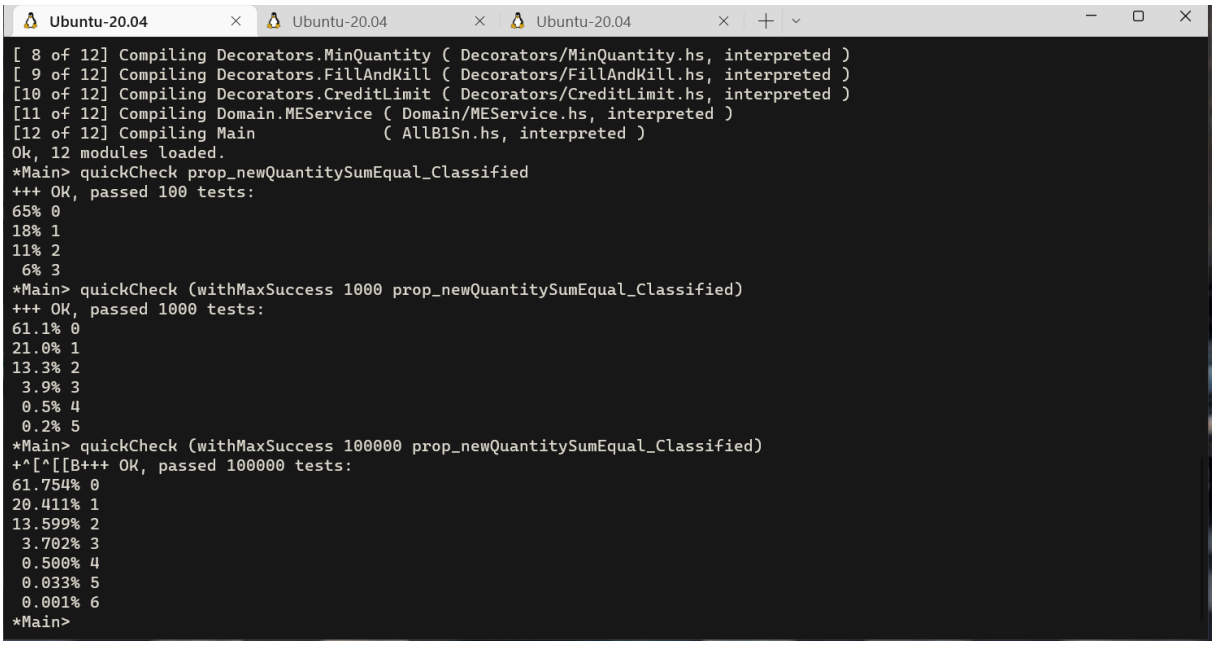
می‌توان در نتایج دید که در ۶۲/۲۵ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰/۲۷ درصد موارد یک معامله، در ۱۳/۱۶ موارد ۲ معامله و در حدود ۴/۳ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۶ معامله داشته باشد، نداشته‌ایم.



تصویر ۱۴ بررسی ویژگی وضعیت قیمت آخری معامله نسبت به سر صف خرید و فروش با ده هزار مثال موفق

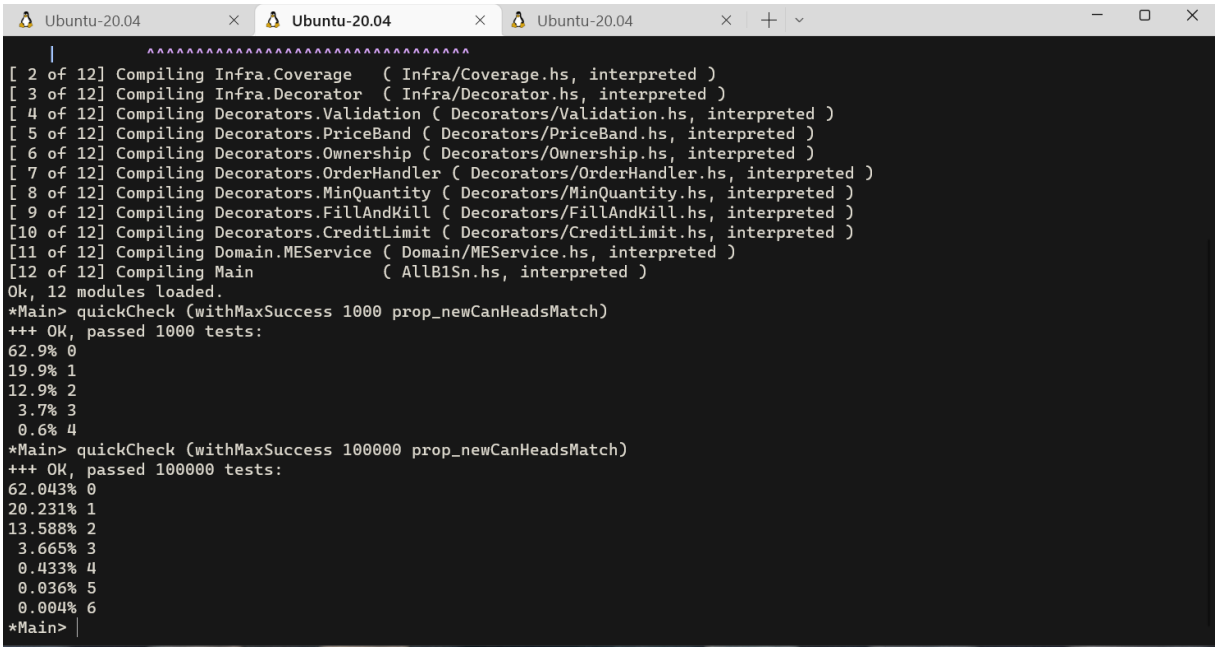
می‌توان در نتایج دید که در ۶۲/۲۷ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰/۳۰ درصد موارد یک معامله، در ۱۳/۱۵ موارد ۲ معامله و در حدود ۴/۳ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۵ معامله داشته باشد، نداشته‌ایم.

- نتایج آزمون با صد هزار مثال موفق



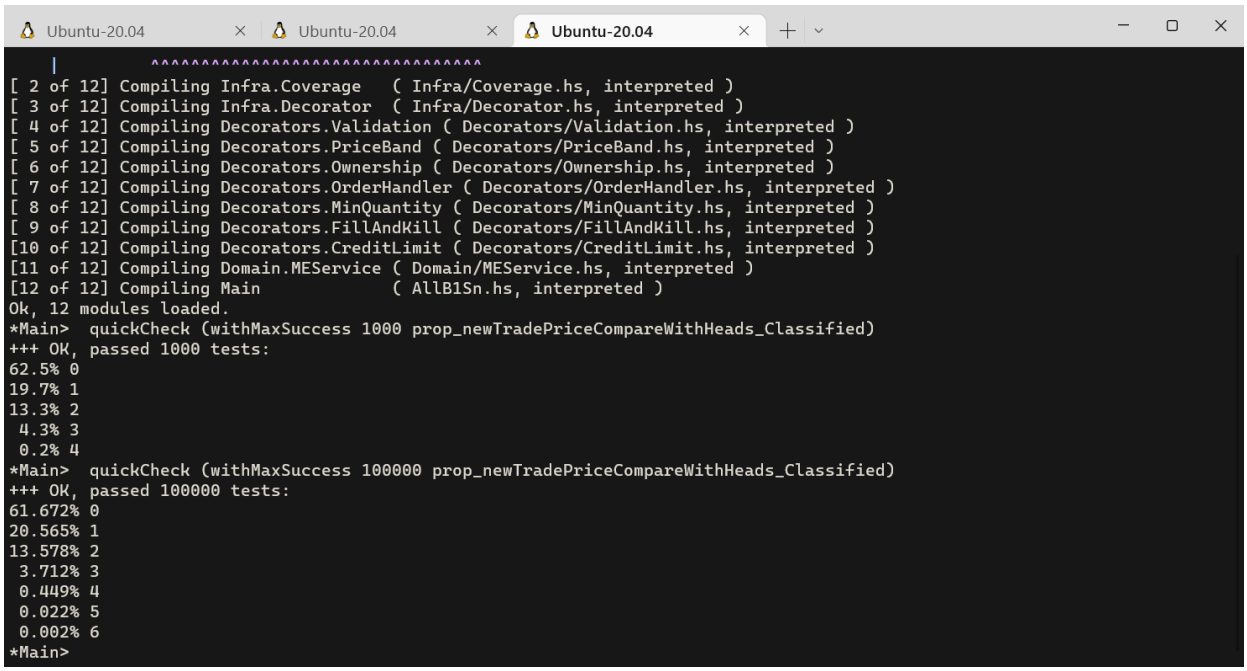
تصویر ۱۵ بررسی ویژگی بقای مقدار با صد هزار مثال موفق

می‌توان در نتایج دید که در ۶۲/۷۵ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰/۴۱ درصد موارد یک معامله، در ۱۳/۵۹ موارد ۲ معامله و در حدود ۴/۳ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۶ معامله داشته باشد، نداشته‌ایم.



تصویر ۱۶ بررسی ویژگی تطبیق‌پذیری سر صف خرید و فروش با صد هزار مثال موفق

می‌توان در نتایج دید که در ۶۲/۸۴ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰/۲۳ درصد موارد یک معامله، در ۱۳/۵۸ موارد ۲ معامله و در حدود ۴/۲ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۶ معامله داشته باشد، نداشته‌ایم.



تصویر ۱۷ بررسی ویژگی وضعیت قیمت آخری معامله نسبت به سر صف خرید و فروش با صد هزار مثال موفق

می‌توان در نتایج دید که در ۶۲/۶۷ درصد موارد هیچ معامله‌ای انجام نشده، در ۲۰/۵۶ درصد موارد یک معامله، در ۱۳/۵۷ موارد ۲ معامله و در حدود ۴/۲ درصد از موارد بیش از ۲ معامله انجام شده است. هم‌چنین هیچ حالتی که بیش از ۶ معامله داشته باشد، نداشته‌ایم.

نکته قابل توجه در آزمون‌هایی که با موفقیت انجام شده‌اند، توزیع تعداد معاملات انجام شده در حالت‌های مختلف است که در تمامی حالت‌ها با اختلاف بسیار کم اعداد مشابهی را نشان می‌دهد. این رفتار می‌تواند نشان‌دهنده‌ی توزیع یکسان فضای حالت تولید شده توسط کوییک‌چک در حالت‌های مختلف باشد.

۲.۳.۴ نتایج حاصل از پیاده‌سازی

مهم‌ترین نتایج حاصل از پیاده‌سازی در این حالت تولید فضای حالت و بهره‌گیری از توابع سازنده بود که مفصلاً در بخش ۳.۲.۳ به آن پراختیم. از روش پیاده‌سازی این پروژه می‌توان برای آزمون کارکردهای دیگر سیستم بهره گرفت.

فصل۵: نتیجه‌گیری

علاوه بر این که در قسمت ۴ به طور تقریباً مطمئن برقراری ویژگی‌های یاد شده در قسمت ۳.۱.۳ را نشان دادیم، از پروژه انجام شده می‌توان بهره‌وری بالای تست مبتنی بر خاصیت را نتیجه گرفت. این روش از تست درصورتی که به طور دقیق و صحیح پیاده‌سازی شود هزینه زمانی آزمون نرم‌افزار را به شدت کاهش می‌دهد. البته باید به قید «دقیق و صحیح» دقت داشت. چون همان‌طور که مثلاً در بخش ۳.۲.۳ و بعداً در ۳.۴ دیدیم، در صورتی که فضای حالت به صورت درست تعریف نشده و ویژگی‌های پیش‌فرض دامنه در آن‌ها رعایت نشود، ممکن است نتیجه مطوبی از آزمون نگرفته و یا هزینه زمانی صرف شده آن‌قدر زیاد شود که صرفه‌جویی زمانی را از دست بدهیم.

هم‌چنین ویژگی‌های بررسی شده در این کارکرد از سیستم هسته معاملات بورسی را می‌توان به حالت‌های دیگری از سیستم نیز تعمیم داد. هر چند که درمورد تعمیم عمومی آن به همه حالت‌های سیستم نمی‌توان اظهار نظر دقیق کرد.

مراجع

[۱] Dyson, P., Longshaw, A.: Architecting enterprise solutions: patterns for highcapability Internet-based systems. John Wiley & Sons (2004)

[۲] Zakeriyan, A., et al. (2021). Towards automatic test case generation for industrial software systems based on functional specifications. International Conference on Fundamentals of Software Engineering, Springer.

[۳] Wang, J. and W. Tepfenhart (2019). Formal Methods in Computer Science, Chapman and Hall/CRC.

[۴] https://en-academic.com/dic.nsf/enwiki/4860833

[۵] https://hackage.haskell.org/package/QuickCheck

[۶] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing AUTOSAR software with QuickCheck," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015.

[۷] https://www.youtube.com/watch?v=hXnS_Xjwk2Y

[۸] https://github.com/mehdijahani1998/haskell-matching-engine

[۹] https://www.haskell.org/cabal/

[۱۰] https://github.com/mehdijahani1998/trivial-matching-function

[۱۱] https://github.com/mehdijahani1998/haskell-matching-engine-simple

[۱۲] http://learnyouahaskell.com/making-our-own-types-and-typeclasses#record-syntax

[۱۳] http://learnyouahaskell.com/types-and-typeclasses#typeclasses-101

[۱۴] http://www.cse.chalmers.se/edu/year/2018/course/TDA452/lectures/TestDataGenerators.html

[۱۵] https://hackage.haskell.org/package/QuickCheck-2.6/docs/Test-QuickCheck-Gen.html

[۱۶] https://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck-Modifiers.html