توسعه نرم افزارهای امبدد
برای سیستم های بحرانی-ایمنی

کریس هابز

محمد حسین شجاع داودی

توسعه نرم افزارهای امبدد

برای سیستم های بحرانی-ایمنی

مولف:

کریس هابز

مترجم:

محمد حسین شجاع داودی

ویرایش اول

دی ماه 1401

فهرست مطالب

مقدمه نویسنده

من این کتاب را برای طراحان، پیاده سازان و گواهی کنندگان دارای تجربه در توسعه نرم افزاری سیستم‌های تعبیه‌شده معمولی نوشته ام، که قصد دارند یک برنامه کاربردی بحرانی-ایمنی برای یک سیستم مبتنی بر نرم‌افزار بنویسند. به طور خاص، هدف این کتاب افرادی است که محصولی را ایجاد می کنند که باید یک یا چند استاندارد بین المللی مربوط به کاربردهای بحرانی-ایمنی را برآورده کند. استانداردهایی نظیر IEC 61508 ، ISO 26262، EN 50128، IEC 62304 یا استانداردهای مرتبط دیگر

نرم افزار و ایمنی

به نظر می‌رسد رسانه‌ها از توصیف ماشین‌های رادیوتراپی که دوزهای اشتباهی را به بیماران داده‌اند، بلایای هوایی و برخوردهای احتمالی، موشک‌هایی که به دلیل جا انداختن کاما در کدنویسی FORTRAN نابود می‌شوند[1]، و بسیاری دیگر از این قبیل خرابی‌های مرتبط با نرم‌افزار، لذت می‌برند. موارد کمتری ثبت می شود که در آن، سخت افزار از کار افتاده باشد و نرم افزار بتواند از بروز فاجعه جلوگیری کند. یکی از نمونه های آن ایرباسی است که در ژانویه 2009 پس از از کار افتادن سخت افزار موتورها در رودخانه هادسون، ایالات متحده آمریکا فرود آمد. بدون در دسترس بودن مداوم نرم افزار کنترل پرواز، تقریباً به طور قطع تلفات قابل توجهی وجود داشت. بنابراین تقسیم سخت افزار/نرم افزار کاملاً یک طرفه نیست.

امیدوارم این کتاب ایده‌هایی را برای طراحان، پیاده سازان و گواهی کنندگان فراهم کند که به افزایش نسبت حوادثی که نرم‌افزار منجر به نجات می شود بیانجامد.

مراجع

تمام تکنیک های شرح داده شده در این کتاب را می توان از طریق صدها مقاله دانشگاهی دیگر مورد بررسی قرار داد. برای اینکه بتوانم راهی برای ورود شما به این مراجع ارائه کنم، در پایان هر فصل منابعی را ارائه کرده ام. با چند استثناء عجیب (به عنوان مثال، مقاله لاردنر[2] 1834 در مورد برنامه نویسی متنوع و مقاله بابیج[3]

[1] انواع مختلفی به فضاپیمای مارینر و مرکوری نسبت داده می شود.

[2] Lardner

[3] Babbage

1837 در مورد موتور محاسبه)، من فقط منابعی را گنجانده ام که شخصاً آنها را به عنوان یک توسعه دهنده نرم افزار مفید می دانم.

برخی از مقالات و کتاب‌هایی که به آنها ارجاع می‌دهم نظرم را در مورد موضوعی تغییر دادند و در بسیاری از موارد باعث شدند که برای بررسی اینکه آیا این مفاهیم واقعا مفید بوده اند یا خیر شروع به برنامه نویسی کنم.

باید اعتراف کنم که من با پرداخت پول به مجلات برای دسترسی به مقالات منتشر شده مخالفم و معتقدم تمام مقالاتی که در فهرست مراجع انتهای هر فصل به آنها اشاره می کنم را می توان آزادانه (و قانونی) از اینترنت دانلود کرد. در برخی موارد، به عنوان مثال، مقاله اصلی کالمن در سال 1960 که در فصل 8 به آن اشاره می کنم، می توان برای بازیابی مقاله اصلی 25 دلار به مجله پرداخت کرد یا به وب سایت مؤسسه آموزشی مرتبط (در این مورد، دانشگاه شمال کارولینا، ایالات متحده آمریکا) مراجعه کرد و مقاله را به صورت رایگان دانلود کرد. انتخاب را به خواننده واگذار می کنم.

من اغلب به استانداردهای مختلف بین المللی اشاره می کنم. این استانداردها به طور مداوم در حال تجدید نظر هستند و وقتی به استانداردی در این کتاب اشاره می کنم، منظورم نسخه های زیر است:

EN 50126 2017/2018 Edition
EN 50128 2011 Edition
EN 50129 2018 Edition
EN 50657 First Edition, 2017
IEEE 754 2008 Edition
IEEE 24765 2017 Edition
ISO 14971 Second Edition, 2007 with 2009 Update
ISO/PAS 21448 First Edition, 2019
ISO 26262 Second Edition, 2018
ISO 29119 First Edition, 2013{2016
IEC 61508 Second Edition, 2010 with 61508-3-1:2016
IEC 62304 First Edition, 2006

ابزارها

من از زبان برنامه نویسی C در مثال ها استفاده کرده ام زیرا این زبان بیشترین ارتباط را با برنامه نویسی امبدد دارد. بنابراین داشتن قدری از دانش زبان برنامه نویسی C برای خواننده مفید است، اما مثال ها کوتاه هستند و باید توسط مهندسان نرم افزار آشنا به زبان های دیگر نیز قابل خواندن باشند.

هر از گاهی به یک ابزار خاص اشاره می کنم. به طور کلی، به ابزارهای متن باز اشاره می‌کنم، نه به این دلیل که این ابزارها همیشه بر معادل‌های تجاری برتری دارند، بلکه به این دلیل که ذکر یک فروشنده تجاری بدون ذکر بقیه آن ها غیرقابل قبول است، مگر اینکه مقایسه دقیقی از محصولات موجود انجام داده باشم.

ابزارها نیز بسیار شخصی هستند. همه ما می دانیم که "جنگ" می تواند از بحث در مورد اینکه آیا vi ویرایشگر بهتری است یا emacs به وجود می آید. من از قبول طعمه در این بحث‌ها امتناع می‌کنم، چون می‌دانم vi بسیار بهتر از emacs است.

بنابراین، ابزار انتخابی من ممکن است ابزار انتخابی شما نباشد. امیدوارم هر زمان که ابزاری را ذکر می‌کنم، اطلاعات کافی را برای شما فراهم کنم تا بتوانید فروشنده‌های تجاری را جستجو کنید.

ویرایش دوم

اولین نسخه از این کتاب در سال 2015 منتشر شد و در سال‌های پس از آن دنیای نرم‌افزارهای بحرانی-ایمنی امبدد به‌طور اساسی تغییر کرده است:

- من از روش استدلال حذفی[1] برای تولید پرونده موارد ایمنی[2] کمک گرفته ام و دریافتم که این یک تکنیک بسیار قدرتمند است. بنابراین من بخشی را در صفحه 330 برای توصیف این تکنیک اضافه کردم.
- ایمنی از عملکرد مورد نظر[3] به یک موضوع مهم به ویژه در زمینه استقلال در صنعت خودرو تبدیل شده است. در زمان نگارش این مقاله، کل این منطقه هنوز بسیار ناپایدار است، اما من از این فرصت استفاده کردم و اطلاعاتی در مورد آن ارائه کردم. صفحه 45 را ببینید.
- امنیت[4] اهمیت فزاینده ای برای ایمنی[5] پیدا کرده است: مدت‌هاست که می‌دانستیم یک سیستم نا امن نمی‌تواند ایمن باشد، اما پیچیدگی روزافزون حملات علیه سیستم‌های نا امن، گنجاندن اطلاعات امنیتی را در بحث ایمنی ضروری‌تر کرده است. این کار مشکلی است، زیرا، اگرچه استدلال های ایمنی به آرامی تغییر می کنند، استدلال های امنیتی می توانند در عرض یک شب تغییر کنند. انتشار یک آسیب‌پذیری مانند Meltdown یا Spectre در یک دستگاه سخت‌افزاری می‌تواند یک استدلال امنیتی را فوراً باطل کند. صفحه 72 را ببینید.
- استانداردهای ایمنی جدید، مانند EN 50657 ، منتشر شده است و استانداردهای از قبل موجود، مانند ISO 26262، به روز شده است. من مراجع ISO 26262 را در این کتاب به روز کرده ام تا ویرایش 2 آن استاندارد را منعکس کند.
- یک مثال بسیار ساده از پردازش کدگذاری شده[6] را در پیوست C اضافه کرده ام.

---

[1] Eliminative argumentation
[2] Safety Case
[3] Safety of the Intended Functionality (SOTIF)
[4] Security
[5] Safety
[6] Coded processing

- در چاپ اول اشتباهات مختلفی وجود داشت که خوانندگان به آن اشاره کردند. من این اشتباهات را اصلاح کردم و بدون شک موارد جدیدی را معرفی کردم.

ما در زمان جالبی زندگی می کنیم - زمانی که در آن باید طوفانی از مشکلات امنیتی، مشکلات ناشی از سیستم های تصادفی و مشکلات ناشی از یادگیری ماشین[1] را مدیریت کنیم.

قدردانی

این کتاب از مواد استفاده شده توسط QNX Software Systems (QSS) برای یک ماژول آموزشی که ابزارها و تکنیک‌های ساخت سیستم‌های نرم‌افزاری تعبیه‌شده را در دستگاه‌های بحرانی-ایمنی مستقر در اتومبیل‌ها، دستگاه‌های پزشکی، سیستم‌های راه‌آهن، سیستم‌های اتوماسیون صنعتی و غیره پوشش می‌دهد، تکامل یافته است.

مطالب این کتاب از کار حرفه ای من بیرون آمده است و البته به تنهایی کار نکرده ام. من از مدیریت QSS و به ویژه مدیران مستقیمم، جف بیکر[2] و آدام مالوری[3]، سپاسگزارم که به من اجازه دادند تا بسیاری از ایده هایم را توسعه دهم و به من اجازه دادند از مطالب آموزشی به عنوان پایه ای برای این کتاب استفاده کنم.

افراد زیادی هستند که باید از آنها برای ارائه ایده ها و چالش ها تشکر کنم. به ویژه از (بر اساس حروف الفبا) دیو بنهام[4]، جان بل[5]، جان هودپول[6]، پاتریک لی[7]، مارتین لوید[8]، ارنست مانتر[9]، راب پترسون[10]، ویل اسنایپس[11] و یی ژنگ[12] - یک گروه محرک از افرادی که در طول سال ها با آنها کار کرده ام.

حتی با وجود ایده ها، ایجاد یک کتاب، کار ناچیزی نیست و می خواهم از همسرم آلیسون[13] تشکر کنم که هر کلمه را با مداد در دست خوانده است. نمی توانم شمارش کنم که او چند مداد فرسوده کرده و این کتاب را به شکلی که به نظرش قابل قبول است تبدیل کرده است. من از او تشکر می کنم. همچنین از چاک کلارک[14]، آدام مالوری[15] و خوانندگان ناشناس ارائه شده توسط ناشر برای تصحیح بسیار جزئی و دقیقشان تشکر می کنم.

---

[1] Machine learning
[2] Jeff Baker
[3] Adam Mallory
[4] Dave Banham
[5] John Bell
[6] John Hudepohl
[7] Patrick Lee
[8] Martin Lloyd
[9] Ernst Munter
[10] Rob Paterson
[11] Will Snipes
[12] Yi Zheng
[13] Alison
[14] Chuck Clark
[15] Adam Mallory

عکس روی جلد توسط چاک کلارک تهیه شده است. با تشکر فراوان برای اجازه استفاده از این عکس.

درباره نویسنده

تا حدودی سه اتفاق باعث شکل گیری من شده است. اولین موردی که می توانم دقیقاً تاریخ آن را بگویم: 24 دسامبر 1968، حدود ساعت 8 شب. من از دانشگاه به خانه برگشته بودم که در آن سال اول کارشناسی بودم و در یک دوره عمومی ریاضی محض درس می خواندم. ترم اول سال اولم را تمام کرده بودم و قبل از رفتن به تعطیلات، کتابی را کم و بیش به صورت تصادفی از کتابخانه اتاق مشترک ریاضیات برداشته بودم. من آن را در شب کریسمس باز کردم و برای اولین بار با نتایج ناقص کورت گودل[1] آشنا شدم. آن روزها این نتایج متا ریاضی[2] در دبیرستان تدریس نمی شد و هیجان فکری بحث گودل مثل یک نیروی عظیم به من ضربه می زد. من بعد از تعطیلات به قصد مطالعه بیشتر در این نوع ریاضیات برگشتم و فرصت باورنکردنی برای شرکت در آموزش های ایمره لاکاتوش[3] در مدرسه اقتصاد لندن را داشتم. در حال حاضر، حتی با دانستن بیشتر نسبت به زمانی که مقاله گودل منتشر شد، هنوز فکر می کنم که این نتایج بالاترین اوج خلاقیت فکری قرن بیستم بود.

رویداد اخیر مقدمه ملایم تری دارد. در سال 2002 از روی هوس، تصمیم گرفتم 24 آهنگی را یاد بگیرم که شامل سیکل آوازی سفر زمستانی[4] فرانتس شوبرت[5] است. هفده سال بعد، من هنوز به همراهی همسرم بر روی آنها کار می کنم، و هر از گاهی احساس می کنم که سرپوش را برداشته ام و اجازه یافته ام به عمق عظیم عاطفی و فکری آنها نگاه کنم.

رویداد میانی ارتباط مستقیم تری با این کتاب دارد و در حکایت 1 در صفحه 4 بازگو شده است. این رویداد من را به مطالعه توسعه نرم افزار برای سیستم های بحرانی-ایمنی سوق داد.

ممکن است عجیب به نظر برسد که نتایج گودل، نرم افزار بحرانی-ایمنی و سفر زمستانی را به هم مرتبط کنید، اما من احساس می کنم که هر کدام نماینده یک چالش روشن فکری درجه یک هستند.

من اخیراً با یک مدیر پروژه ملاقات کردم که گفت که توسعه نرم افزار برای یک برنامه کاربردی مهم ایمنی درست مانند توسعه نرم افزار برای هر برنامه دیگری است، با هزینه های زیاد کاغذبازی. شاید باید این کتاب را به او تقدیم می کردم.

کریس هابز

اوتاوا

---

[1] Kurt Godel

[2] Meta-mathematical

[3] Imre Lakatos

[4] Winterreise

[5] Franz Schubert

مقدمه مترجم

سیستم های بحرانی-ایمنی سیستم هایی هستند که ایمنی در آن ها مهم است. ایمنی زمانی فراهم می شود که کلیه توابع و عملیات ها در اینگونه سیستم ها به صورت ایمن عمل کنند. یک سیستم ایمن باید در زمان مورد نیاز در دسترس باشد و با قابلیت اطمینان بالا (صحیح و بدون خطا) کارها را انجام دهد و در صورت بروز خطا، خراب نشده و واکنش مناسب به خرابی ها نشان دهد. به سیستم هایی که در صنایع حساس مثل صنایع خودروسازی، راه آهن، مهندسی پزشکی، نفت، گاز و پتروشیمی وجود دارند به علت اینکه در صورت خراب شدن ممکن است خسارات جانی و مالی در پی داشته باشند، ایمنی بحرانی گفته می شود.

برای اضافه کردن مجموعه قابلیت های نامبرده شده به یک سیستم یعنی افزایش در دسترس بودن، اجرای درست عملیات ها، تحمل پذیری در برابر خطا، بایستی هم از نظر سخت افزاری و هم از نظر نرم افزاری تمهیداتی اندیشیده شود. در این کتاب بیشتر به معرفی و تشریح تمهیدات نرم افزاری سیستم های ایمنی بحرانی پرداخته است اما خواننده این کتاب در مورد سخت افزار هم دید کلی به دست می آورد.

برای سریعتر مشغول به کار شدن در یکی از صنایع نامبرده شده، یا برای ارتقای رزومه خود یا برای ایمن کردن محصول امبدد بحرانی-ایمنی شرکت خود خواندن این کتاب توصیه می شود. یکی دیگر از ویژگی های این کتاب جامع بودن آن است به طوری که مولف از منابع و مراجع مختلفی استفاده کرده است و سعی داشته است تا با رعایت تمامی استانداردهای مربوط به ایمنی نکات تلفیقی را ارائه کند. این کار باعث شده است تا خواننده با مطالعه این کتاب با بسیاری از مفاهیم و مباحث مطرح شده در تعداد زیادی از استانداردها آشنا شود.

زمانی که من به دنبال یک منبع مفید برای تفسیر مطالب استاندارد IEC 61508 برای گرفتن گواهینامه یک محصول بودم با این کتاب آشنا شدم و شروع به مطالعه آن کردم. از آن جایی که این کتاب بسیار مفید واقع شد و به طور کلی باعث بهبود و تسریع در روند اصلاح اشکالات پروژه شد، تصمیم گرفتم تا آن را ترجمه کنم. به نظرم در صنایع رو به رشد خصوصا صنایع پرحادثه نظیر خودروسازی نیاز اساسی برای بهبود کیفیت محصول جهت اخذ گواهینامه های مرتبط با ایمنی برای محصولات می باشد. امیدوارم که این کتاب مورد استفاده مهندسین و مدیران این شرکت ها واقع شود تا شاهد تولید محصولات با کیفیت ایرانی بیشتری باشیم.

نظرات، پیشنهادات و انتقادات بی پروای شما خوانندگان محترم قطعا باعث بهبود ترجمه کتاب و پیشرفت جامعه مهندسین در این حوزه خواهد بود. لذا خواهشمند است از طریق یکی از راه های ارتباطی درج شده در وبسایت الکترو ولت به نشانی electrovolt.ir با من در ارتباط باشید.

بخش اول : پس زمینه

فصل اول : مقدمه

ما در حال ورود به دنیای جدیدی هستیم که در آن داده ها ممکن است مهمتر از نرم افزار باشد.
Tim O'Reilly

این کتاب در مورد توسعه نرم افزارهای امبدد قابل اعتماد می باشد. مرسوم است که کتاب ها و مقاله ها در مورد نرم افزارهای امبدد با ارائه آمارهایی در خصوص اینکه تعداد خط کدهای نرم افزار امبدد در ماشین ها چقدر بیشتر از هواپیماها می باشد، شروع شود. همچنین مرسوم است که کتاب ها و مقاله ها در مورد کدهای قابل اعتماد و موعظه هایی در مورد معایب دیر پیدا کردن باگ ها در پروسه توسعه محصول و نمایش منحنی نمایی شناخته شده، شروع شود.
چیزی که من را از این رویکرد باز می دارد خواندن کتاب فوق العاده لورن بوساویت[1] (مرجع [1]) با نام لپرکان[2] مهندسی نرم افزار است که بیرحمانه چنین پیش فرض هایی در مهندسی نرم افزار را مورد بررسی قرار می دهد و بی اساس بودن آن ها را آشکار می کند. بویژه بوساویت به منطق دایره ای مرتبط با هزینه نمایی یافتن و رفع باگ ها در مراحل بعدی توسعه اشاره می کند. مهندسی نرم افزار یک فرآیند اجتماعی است نه یک فرآیند طبیعی بنابراین این ویژگی را دارد که اعتقاداتی که در مورد مهندسی نرم افزار داریم مستقیما روی واقعیت تاثیرگذار باشد. از آن جایی که ما انتظار داریم رفع باگ ها در مراحل بعدی توسعه هزینه بیشتری داشته باشد، رویه هایی ایجاد کرده ایم که آن را گران تر می کند. در صورتی که واقعا چنین نیست.
نظرات بوساویت مهم است و چندین بار در این کتاب مورد استناد قرار خواهد گرفت و امیدوارم که ایمان شما را نسبت به سایر جذام های مرتبط با نرم افزارهای امبدد متزلزل کنم. به ویژه "صد میلیون خط کد در یک خودرو امروزی" به نظر می رسد به یک شعار تبدیل شده است که باید از آن رهایی پیدا کنیم.

فرهنگ ایمنی

فرهنگ ایمنی فرهنگی است که به رئیس اجازه می دهد اخبار بد بشنود.
Sidney Dekker

این کتاب بیشتر به جنبه های فنی ساخت محصولی می پردازد که میتواند بر اساس استانداردهایی مانند IEC61508 و ISO26262 گواهی شود. یک جنبه بسیار مهم دیگر در مورد ساخت یک محصول وجود دارد

[1] Laurent Bossavit, The Leprechauns of Software Engineering
[2] در باورهای مردم ایرلندی لپرکان به جن کوچکی گفته می شود که به شکل پیرمردی ریش دراز ظاهر می شود و مکان گنج های نهفته را آشکار می سازد

که می تواند بر مسئولیت ایمنی عمومی مهندسان طراح، پیاده ساز و تایید کننده تاثیر بگذارد. خواندن استانداردهای ایمنی و برطرف کردن نیازهای آن صرفا به عنوان حلقه هایی که باید از آن عبور کرد آسان است اما این استانداردها برای خواندن توسط افرادی که با فرهنگ ایمنی کار می کنند، نوشته شده است.

حکایت 1 : اولین باری که من شروع به فکر کردن در مورد جنبه های طراحی نرم افزار بحرانی-ایمنی کردم در اواخر سال 1980 بود، زمانی که من توسعه بخشی از یک تجهیز مخابراتی را مدیریت می کردم.

یک برنامه نویس با خواندن کدهای روی میز خود متوجه شد که یکی از بررسی های ایمنی محصول قابل دور زدن است. این سیستم به گونه ای طراحی شده بود که زمانی که یک تکنسین روی تجهیزات کار می کرد، یک ولتاژ بالا به صورت تستی برای اندازه گیری ایمنی روی خط خارجی اعمال می شد. در صورت وجود ولتاژ بالا روی خط، نرم افزار از بستن رله هایی که تجهیزات تکنسین را به خط متصل می کرد خودداری می کرد. خطای یافت شده توسط برنامه نویس باعث شد تا بررسی ولتاژ بالا در شرایط بسیار غیرعادی حذف شود.

حذف این پروسه بررسی ولتاژ بالا در یک شرایط غیرعادی خطای پیدا شده توسط آن برنامه نویس بود.

من تحت فشار قابل توجهی از طرف مدیرم برای ارسال سریعتر محصول بودم. اشاره شد که ولتاژ بالا فقط در شرایط خاص وجود دارد و حتی اگر هم باشد در شرایط غیر عادی خاصی است که این پروسه حذف می گردد.

اکنون متوجه شده ام که در آن زمان هیچ یک از تکنیک های توصیف شده در این کتاب برای ارزیابی موقعیت و تصمیم گیری منطقی و موجه را نداشتم. این حادثه ای بود که مرا درگیر کرد و به سمت این کتاب سوق داد.

ضمیمه B از ISO 26262-2 لیستی از نمونه هایی را ارائه می کند که نشان دهنده فرهنگ های ایمنی خوب یا ضعیف است. از جمله تفکر گروهی (بد)، تنوع فکری در تیم (خوب)، شناسایی و افشای خطرات احتمالی برای ایمنی (خوب) و یک سیستم تشویق کننده که کسانی که با انتخاب راه های میانبر باعث به خطر افتادن امنیت می شوند را جریمه می کند. (خوب)

هرکسی که به توسعه یک دستگاه بحرانی-ایمنی علاقه دارد باید بداند که زندگی انسان ممکن است به کیفیت طراحی و اجرا بستگی داشته باشد.

در یک تحقیق رسمی در مورد تراژدی Deepwater Horizon (مرجع [2]) که به طور ویژه به فرهنگ ایمنی در صنعت نفت و گاز می پردازد بیان شده است که : علت بدیهی حادثه انفجار چاه ماکوندو را میتوان در یک سری از اشتباهات قابل شناسایی که توسط BP, Halliburton و Transocean ایجاد شده بود دانست. این حادثه با آشکار کردن چنین خرابی های سیستماتیکی در مدیریت ریسک، فرهنگ ایمنی کل صنعت را زیر سوال می برد.

اصطلاح "فرهنگ ایمنی" 116 بار در بررسی نیمرود[1] (مرجع [3]) پس از بررسی سقوط هواپیمای XV230 در سال 2006 آورده شده است. در این بررسی یک فصل کامل در خصوص فرهنگ ایمنی توضیح می دهد و به صراحت بیان می کند که در سیستم پروازی فعلی وزارت دفاع کاستی هایی از نظر فرهنگ ایمنی وجود دارد که قابلیت پرواز را تحت الشعاع قرار داده است.

در فرهنگ ایمنی سالم، هر توسعه دهنده ای روی یک محصول بحرانی ایمنی کار می کند و باید بداند که چگونه یک خطر را ارزیابی کند و وظیفه دارد ملاحضات ایمنی آن را ملاحضه نماید.

همانطور که لس چمبرز[2] در فوریه 2012 در وبلاگ خود هنگام اظهار نظر در مورد تراژدی Deepwater Horizon گفت :

"ما وظیفه اخلاقی داریم که از فضای ریاضیاتی ذهنی خود بیرون بیاییم و مسئولیت اجتماعی بیشتری را در قبال سیستم هایی که می سازیم بپذیریم، حتی اگر این به معنای درگیری و تهدید شغلی از جانب رئیس قدرتمند باشد. دانش به عنوان ابزار استفاده سنتی از مهندسی می باشد اما بایستی در حیطه اعتقادات به آن عمل شود."

یکی دیگر از سوالاتی که چمبرز در آن پست وبلاگ به آن پرداخت این بود که آیا تصویب یک تصمیم از پایین به سمت بالا قابل قبول است یا خیر. در اتفاقی که در حکایت 1 شرح داده شد، من از امضای اسناد مربوط به انتشار محصول امتناع کردم و تصمیم را به رئیسم واگذار کردم. آیا این امر مرا از لحاظ قانونی یا اخلاقی از گناهی که در صورت ارسال تجهیزات و آسیب رسیدن به کسی بوجود می آمد مبرا می کرد؟ البته خوشبختانه رئیس من هم از امضا امتناع کرد و ارسال تجهیزات با هزینه بالایی به تعویق افتاد.

[1] Nimrod
[2] Les Chambers

حکایت 2 : زمانی که چند سال پیش در کنفرانسی در مورد سیستم های بحرانی-ایمنی شرکت کرده بودم، یکی از شرکت کنندگان گفت دوستی دارد که وکیل است و اغلب از مهندسانی دفاع می کرد که متهم به تولید محصول معیوبی بودند که منجر به فوت یا جراحت شدید شده بود. ظاهرا وکیل مطمئن بود که اگر پرونده به دادگاه برسد، می تواند بی گناهی مهندسین را ثابت کند. اما در بسیاری از موارد پرونده به دادگاه نمی رسید چون تا قبل از دادگاه آن مهندس خودکشی می کرد. این حکایت کنفرانس را تحت تاثیر قرار داد چرا که هر یک از ما در مورد پیامدهای آن شخصا فکر می کردیم.

مسیر ما

من ساختار این کتاب را به صورت زیر تنظیم کرده ام:

اصول پیش نیاز

در فصل 2 برخی از اصطلاحاتی که در فصل های بعدی کتاب استفاده شده است، معرفی می شود. این موضوع از آن جهت مهم است که مفهوم کلماتی نظیر خطا، اشکال و خرابی که در زندگی روزمره به جای هم نیز به کار می روند، مشخص شود چرا که در بحث سیستم های امبدد معانی واضح و جداگانه ای دارند.

یک دستگاه برای استفاده در سیستم های بحرانی ایمنی مطابق با الزامات یک استاندارد بین المللی بایستی توسعه داده شود. در مرجع [4] جان مک درمید[1] و اندرو را[2] اشاره می کنند که صدها استاندارد مربوط به ایمنی وجود دارند. از میان همه آن ها من تعداد کمی را برای بحث در فصل 3 انتخاب کردم. به طور ویژه استاندارد IEC61508 که به ایمنی در سیستم های صنعتی مربوط می شود و پایه و اساس بسیاری از استانداردهای دیگر است. همچنین استاندارد ISO26262 که همان استاندارد IEC61508 را برای سیستم های داخلی خودرو اختصاصی کرده و گسترش می دهد. همچنین استاندارد IEC62304 که ایمنی نرم افزارهای حوزه پزشکی را پوشش می دهد. علاوه بر این ها من استانداردهای دیگری نظیر IEC29119 استاندارد تست نرم افزار و EN50128 استاندارد راه آهن را نیز به منظور حمایت از استدلال های خویش ذکر می کنم.

برای ملموس تر کردن بحث در فصل 4 دو شرکت فرضی را معرفی می کنم که یکی از آن ها تولید کننده دستگاهی در حوزه سیستم های بحرانی ایمنی است و دیگری تامین کننده قطعات آن دستگاه است. این کار به من اجازه می دهد تا نشان دهم که چگونه ممکن است این دو شرکت در چهارچوب استانداردها با هم کار کنند.

توسعه محصول برای کاربرد بحرانی-ایمنی

[1] John McDermid
[2] Andrew Rae

در فصل 5 تجزیه و تحلیل هایی نظیر تجزیه و تحلیل خطر و ریسک، تحلیل موارد ایمنی و تحلیل حالت خرابی و ... برای توسعه محصول بحرانی-ایمنی تشریح می شود. همچنین در فصل 6 مشکلات مربوط به ادغام اجزای خارجی (احتمالا شخص ثالث) در یک دستگاه بحرانی ایمنی مورد بحث قرار می گیرد.

تکنیک های توصیه شده در استانداردها

هر دو استاندارد IEC 61508 و ISO 26262 حاوی تعداد زیادی جدول هستند که تکنیک های مختلفی را برای استفاده در طول توسعه نرم افزاری توصیه می کنند. بسیاری از این موارد معمولاً در هر توسعه نرم افزاری (مثلا تست اینترفیس) استفاده می شوند، اما برخی از آنها کمتر شناخته شده هستند. فصل های 7 تا 19 برخی از این تکنیک های کمتر شناخته شده را پوشش می دهد.

برای راحتی، من اینها را به الگوهای مورد استفاده در معماری و طراحی یک محصول (فصل 7 تا 10)، ابزارهایی که برای اطمینان از اعتبار طرح استفاده می شود (فصل 11 تا 15)، تکنیک های مورد استفاده در حین اجرا (فصل های 16 تا 18) و مواردی که در حین تایید اجرا می شوند (فصل 19) تقسیم کرده ام.

ابزارهای توسعه

یکی از وظایف ضروری در طول توسعه که گاهی نادیده گرفته می شود، تهیه مدارکی از عملکرد صحیح ابزارهای مورد استفاده است. مثلا اگر کامپایلر دچار اشتباه در کامپایل کد شود دیگر اهمیت ندارد که برنامه نویس کد خوبی نوشته است یا خیر. در فصل 20 مطالبی در مورد چگونگی بدست آوردن چنین مدارکی ارائه می شود.

اضافه کنم که من در فصل 5 نماد ساختاری هدف[1] و شبکه های مبتنی بر نظریه بیز[2] را معرفی می کنم اما جزئیات مربوط به آن ها را به ترتیب به پیوست های A و B منتقل می کنم. در نهایت کمی ریاضیات در این کتاب وجود دارد که در ضمیمه C نمادهای مورد استفاده من تشریح شده است.

نحوه انتخاب تکنیک ها

من از سه ملاک زیر برای انتخاب تکنیک های مورد استفاده در این کتاب استفاده کرده ام :

1. تکنیک مورد نظر بایستی به صراحت در یکی از استانداردهای IEC61508 یا ISO26262 یا هر دو توصیه شود. البته از بین تمامی تکنیک ها فقط یک مورد بوده است که مستقیما در آن استانداردها توصیه نمی شود.
2. بایستی خودم از تکنیک مورد نظر استفاده مستقیم می کردم.

[1] goal structuring notation (GSN)
[2] Bayes

3. تکنیک مورد نظر نباید در اکثریت شرکت هایی که اخیرا از آن ها بازدید کردم رایج باشد.

رویکرد توسعه

برای توسعه محصول روش های بسیاری وجود دارد. از روش توسعه آبشار خالص[1] (که بوساویت در کتاب خود یک فصل کامل و یک ضمیمه را به آن اختصاص داده است) گرفته تا روش آبشار اصلاح شده، طراحی با زمانبندی، نظریه W ، توسعه برنامه مشترک، توسعه سریع برنامه، توسعه جعبه زمانی، نمونه سازی سریع، توسعه چابک[2] با اسکرام[3] و برنامه نویسی فوق العاده[4]

نمیخواهم در مورد اینکه کدام یک از این تکنیک ها برای توسعه سیستم های امبدد مناسب تر است جهت گیری کنم. بنابراین در تمام طول این کتاب از رویکرد ترکیبی که در شکل 1.1 آمده است استفاده خواهم کرد.

در طول فرآیند توسعه، وظایف خاصی باید تکمیل شوند (اگر چه معمولا به صورت متوالی نیستند) :

چرخه طراحی اولیه

در طول چرخه اولیه طراحی که در پایین شکل 1.1 نشان داده شده است، احتیاجات یک جزء یا کل سیستم تجزیه و تحلیل می شود و پیشنهادی امکان پذیر برای طراحی آن ارائه می شود. این موضوعات در معرض تجزیه و تحلیل های سریع نظیر مدل مارکوف[5] (فصل 11) هستند تا ببینیم آیا آن ها میتوانند قابلیت اعتماد[6] مورد نیاز برای تشکیل طرح نهایی را ارائه دهند.

چرخه جزئیات طراحی

پس از انتخاب یک یا چند طرح برتر، تجزیه و تحلیل خرابی انجام می شود. چه تجزیه و تحلیل خرابی به صورت ساده تر با استفاده از مدل مارکوف انجام شود و چه به صورت پیچیده تر با استفاده از تحلیل درختی خطا[7] (فصل 12) میزان خرابی پیش بینی شده نرم افزار بایستی مشخص باشد. تعیین میزان خرابی نرم افزار موضوعی است که در فصل 13 به آن می پردازم.

چرخه تایید طراحی

هنگامی که عناصر طرح انتخاب شدند، طرح باید تأیید شود: آیا این طراحی همه شرایط ممکن را کنترل می کند، آیا دستگاه هرگز قفل می شود و پیشرفت نمی کند ؟

چرخه پیاده سازی

پس از تکمیل طراحی، اجرا شروع می شود و باید الگوهایی برای اجرا و تأیید اجرای محصول انتخاب شود.

---

[1] Pure Waterfall
[2] Agile
[3] SCRUM
[4] eXtreme
[5] Markov Model
[6] Depandability
[7] fault tree analysis

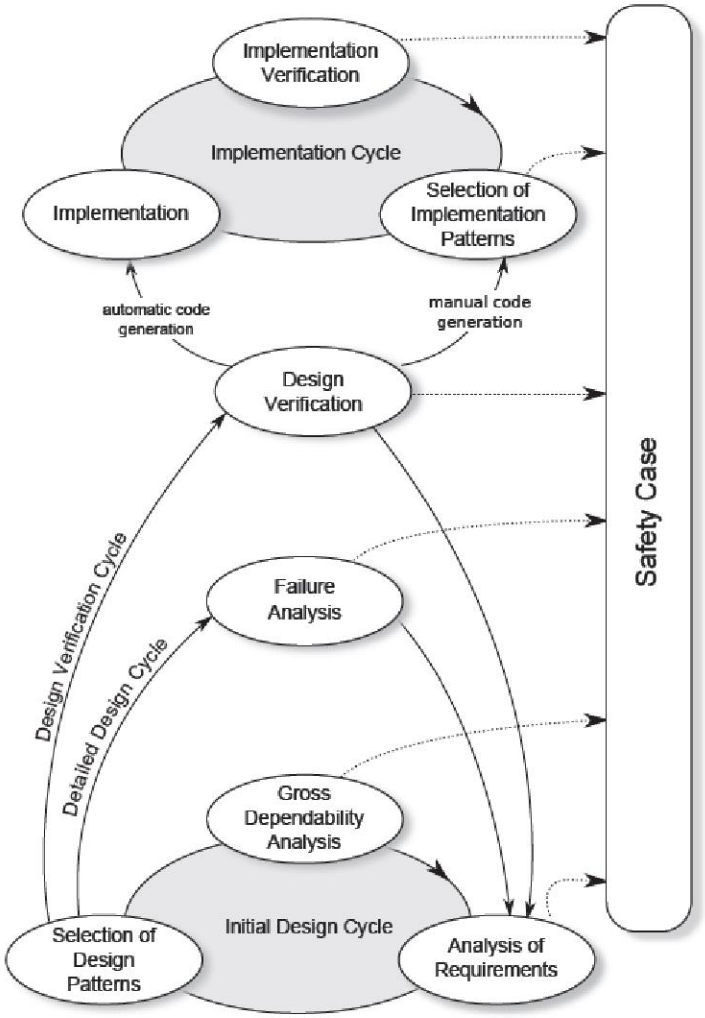


Figure 1.1 The suggested approach.

توجه داشته باشید که من با دقت از گفتن این که آیا این چرخه ها فقط یک بار برای توسعه یک دستگاه خاص انجام می شوند (فرآیند آبشار) یا برای هر قطعه شاید هر چند هفته یک بار بررسی می شوند (فرآیند چابک) یا حتی یک بار در روز تکرار می شوند (برنامه نویسی فوق العاده)، اجتناب کردم.

با توجه به این طیف گسترده از روش های توسعه محصول، من اعتقاد شخصی خودم را دارم که به احتمال زیاد منجر به یک پروژه موفقیت آمیز، تحویل به موقع و بر اساس بودجه و برآورده کردن الزامات ایمنی آن می شود، اما این باور مطلق به این کتاب منطقی نیست. در هر رویکردی که انتخاب شود، توصیف لازم است.

با این حال از لیست بالا میتوان به یک نکته اشاره کرد: تاکید من بر روی چرخه طراحی و تایید طراحی بیشتر از چرخه اجرا و تایید اجرا است. این مسئله عمدی است و در فصل 15 سعی می کنم که ادعای خود را توجیه کنم. چرا که انقدر مهارت های مورد نیاز برای طراحی و پیاده سازی متنوع است که بعید میدانم یک نفر بتواند

هر دو چرخه را به نحو احسن انجام دهد. ISO 26262-2 به طور خاص "وابستگی شدید به آزمایش در پایان چرخه توسعه محصول" را به عنوان نمونه ای از فرهنگ ایمنی ضعیف نام می برد.
نکته دیگری که در شکل 1.1 مورد تاکید قرار گرفته است. نقش محوری پرونده موارد ایمنی[1] می باشد. تمامی تحلیل ها و تصمیم های اتخاذ شده در طول پروژه بایستی جزئی از پرونده موارد ایمنی شوند. این سند ضروری در فصل 5 از صفحه 61 توضیح داده شده است.

چالش امروزی

اگرچه در دو دهه گذشته شاهد رشد چشمگیری در ابزارها و تکنیک‌هایی که می‌توانیم برای توسعه سیستم‌های ایمن و جاسازی شده به کار ببریم، بوده‌ایم، اما چالش‌ها همچنان پابرجا هستند و چالش‌های جدیدی در حال ظهور هستند.

امنیت[2]

در گذشته بسیاری از سیستم های امبدد از نظر فیزیکی ایمن بودند مثلا در کابین راننده قطار یا در پشت حصار مغازه ها قفل شده بودند. اما امروزه تقریبا اکثر دستگاه ها از طریق وای فای، بلوتوث، درایورهای USB، ماهواره های GPS با محیط در ارتباط هستند و این ها همگی کانال های آسیب پذیر امنیتی محسوب می شوند.
ایمنی و امنیت عمیقا با هم در ارتباط هستند. رفتار یک دستگاه ناامن به محض نفوذ یک هکر به طور موثر غیر قابل پیش بینی است. ایمنی و امنیت معمولاً متضاد هم هستند (افزایش یکی به قیمت کاهش دیگری می شود) و این تعادل امنیت/ایمنی در صفحه 101 توضیح داده شده است. اقداماتی برای ترکیب موارد ایمنی و امنیتی وجود دارد. به صفحه 72 مراجعه کنید. با این حال، هنوز هیچ روش کلی برای ترکیب مسائل امنیتی و ایمنی وجود ندارد.

ابزارهایی برای تعادل نیازهای معماری

در فصل 7 نیاز به یک معمار و طراح برای ایجاد تعادل بین در دسترس بودن، قابلیت اطمینان، کارایی، سودمندی، امنیت و ایمنی را بیان می کند. این ویژگی ها بر روی هم تاثیر می گذارند و تصمیم گیری طراحی در یک حوزه هم بر روی حوزه های دیگر اثر می گذارد. من هیچ ابزاری را نمی شناسم تا بتواند به یک تحلیلگر کمک کند تا بتواند این ویژگی های سیستم را دستکاری کند و یکی از آن ها را با سایرین مبادله کند.

[1] Safety Case
[2] Security

شناسایی خطای سخت افزار

همانطور که در صفحه 136 توضیح داده شده، امروز پردازنده ها و حافظه ها از نظر سخت افزاری نسبت به دهه گذشته از قابلیت اطمینان کمتری برخوردار هستند. این مسئله در نتیجه افزایش پیچیدگی تراشه ها (اکثر پردازنده ها با 20 صفحه یا بیشتر غلطنامه[1] ارائه می شوند) و کاهش اندازه سلول ها که آن ها را در برابر گفتگوی متقابل[2] (برای مثال چکش ردیفی[3] که در مرجع [5] توضیح داده شده است)، تداخل الکترومغناطیسی و اثرات ثانویه کیهانی مستعدتر کرده است. علاوه بر این، نقص‌های امنیتی اساسی در طراحی تراشه‌ها (مانند آسیب‌پذیری‌های Meltdown، Spoiler، Spectra و موتور مدیریت اینتل) به طور فزاینده‌ای پیدا و مورد سوء استفاده قرار می‌گیرند.

بدتر از این در بسیاری از پردازنده های مدرن، حافظه های نهان[4] و کمک پردازنده ها[5] از دید سیستم عامل و برنامه کاربردی پنهان هستند. بنابراین خطاها را نمیتوان مستقیما از طریق تاثیر آن ها بر عملکرد برنامه شناسایی کرد.

پردازنده های جدیدی در حال ظهور هستند که سطوح بهبود یافته تری از تشخیص خطا را ارائه می کنند. اما این پردازنده ها گران تر هستند و هنوز برای استفاده مناسب نشدند.

پیش بینی زمان موعد انجام[6]

بسیاری از سیستم‌های امبدد در گذشته بسیار ساده بودند و امکان انجام محاسبات بی درنگ[7] در آن ها بیشتر وجود داشته است. در این سیستم ها تحت هر شرایطی، زمان اجرای مشخصی رعایت می‌شود. اما اکنون برنامه هایی که نیاز به بی درنگ بودن دارند، در کنار سایر برنامه های کاربردی بر روی پردازنده های چند هسته ای با دستورات غیر قطعی[8] و حافظه های نهان اجرا می شوند. بنابراین تضمین های قطعی در حال از بین رفتن می باشد و جای خود را به تضمین های احتمالی نظیر "احتمال از دست رفتن موعد انجام X کمتر از 10^{-6} در هر ساعت است" داده است. ابزارهای کمی وجود دارد که امکان انجام محاسبات فراتر از شبیه سازی رویداد گسسته[9] (صفحه 190) را فراهم می کند.

[1] Errata
[2] Cross-talk
[3] row-hammering
[4] cache
[5] coprocessor
[6] deadline
[7] Real-time
[8] non-deterministic instruction
[9] Descrete-event

داده[1]

دنیای سیستم های حیاتی ایمنی اخیراً شروع به جدی گرفتن داده ها کرده است، چه داده های پیکربندی یا داده هایی مانند وزن در یک شبکه عصبی که رفتار سیستم را تعیین می کند .تا چند سال پیش، تجزیه و تحلیل بر دو جنبه از یک سیستم متمرکز بود سخت افزار و نرم افزار | و این در ساختار ISO 26262 و IEC 61508منعکس شده است که بخش هایی به آن موضوعات اختصاص داده شده است.

دنیای سیستم های بحرانی-ایمنی اخیرا شروع به جدی گرفتن داده ها، بویژه داده های پیکربندی یا داده هایی مانند وزن در یک شبکه عصبی که رفتار سیستم را تعیین می کند، کرده است. تا چند سال پیش، تجزیه و تحلیل بر دو جنبه از یک سیستم متمرکز بود: سخت افزار و نرم افزار. این موضوع در بخش های جداگانه ای در ساختار ISO 26262 و IEC 61508 پرداخته شده است.

اکنون تشخیص داده می شود که داده ها به عنوان عنصر سومی هستند که به همان اندازه مهم می باشند. داده ها تحت هدایت مارک پارسونز[2] گروهی تحت عنوان گروه کار در ایمنی ابتکاری داده ها[3] که بخشی از باشگاه سیستم های بحرانی-ایمنی[4] می باشد، دستورالعمل های بسیار مفیدی را در مورد استفاده از داده ها در سیستم های بحرانی ایمنی آماده کرده اند اما در این زمینه کار بیشتری مورد نیاز است. در ضمیمه H نسخه 3.1 دستورالعمل ارائه شده توسط DSIWG (مرجع [6]) حاوی لیستی وحشتناک از نمونه حادثه هایی است که در آن جان انسان ها به دلیل خطای داده به نسبت بیشتر از خطای سخت افزاری و نرم افزاری از دست رفته است.

سقوط هواپیمای ایرباس A400M که در صفحه 21 این کتاب توضیح داده شده است، نمونه ای از سقوط یک هواپیما به دلیل داده ها نه به دلیل نرم افزار می باشد.

یادگیری ماشین[5]

از ابتدا انسان‌ها موقعیت‌هایی را که دستگاه امبدد می خواهد در آن کار کند را شناسایی و تجزیه و تحلیل کرده‌اند و آن دستگاه را طوری برنامه‌ریزی کرده‌اند که در آن مکان خاص به درستی واکنش نشان دهد. این امر به صورت فزاینده ای در حال غیرممکن شدن است.

برای مثال، یک خودروی خودران باید در موقعیت‌های بیشتر از آنچه که می‌توانیم برشماریم، کارآمد و ایمن کار کند. من ده ها هزار ساعت رانندگی کرده ام، اما هرگز (هنوز) مردی را ندیده ام که کت و شلوار مرغی پوشیده

[1] Data
[2] Mike Parsons
[3] Data Safety Initiative Working Group (DSIWG)
[4] Safety Critical Systems Club (SCSC)
[5] Machine Learning

باشد و بخواهد که جلوی من از جاده عبور کند. با این حال، اگر فردا با آن موقعیت روبرو شوم، می دانم که چگونه واکنش نشان دهم. یک خودرو خودران چگونه با آن مرد برخورد می کند؟

در زمان نگارش این کتاب، جامعه سیستم های بحرانی-ایمنی به مفاهیم یادگیری ماشین همانند ISO 26262 به صراحت توجه نمی کنند. پاسخگویی به خواسته هایی مانند "کد را به من نشان بده که اگر یک جوجه بلند دو متری جلوی باشد، ماشین را متوقف می کند" غیرممکن خواهد بود. این حوزه از آموزش عبارت: ایمنی از عملکرد مورد نظر[1] را ایجاد کرده است. SOTIF ایمنی یک سیستم را هنگامی که عملکرد مورد نظر خود را بدون اشکال در قطعات انجام می دهد، پوشش می دهد. برای جزئیات بیشتر به صفحه 45 مراجعه کنید.

یادگیری ماشین تنها یکی از جنبه های توانایی ما برای ایجاد سیستم های پیچیده تری است که از توانایی ما برای توصیف فراتر است و حتی نمی توانیم برای آن الزامات تعیین کنیم.

مراجع

1. L. Bossavit, The Leprechauns of Software Engineering: How Folklaw Turns into Fact and What to Do about It. Leanpub, 2013
2. B. Graham, W. K. Reilly, F. Beinecke, D. F. Boesch, T. D. Garcia, C. A. Murray, and F. Ulmer, “DeepWater|The Gulf Oil Disaster and the Future of O_shore Drilling: Report to the President," 2011
3. C. Haddon-Cave, The Nimrod Review. UK Government, 2009
4. J. McDermid and A. Rae, “Goal-Based Safety Standards: Promises and Pitfalls," in 2012 Safety Critical Systems Symposium, SSS '12, (Bristol, UK), Safety-Critical Systems Club, 2012.
5. L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, “Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in IEEE Symposium on Security and Privacy, May 2019.
6. Data Safety Initiative Working Group, Data Safety Guidance. Available from data-safety.scsc.uk/comments: SCSC-127D, 2019

[1] Safety of the Intended Functionality (SOTIF)

فصل دوم : اصطلاحات ایمنی

امروز نام گذاری قطعات را داریم

Henry Reed

برای اینکه بتوانیم در مورد سیستم های بحرانی ایمنی صحبت کنیم ابتدا بایستی بین اصطلاحاتی که تقریبا به جای یکدیگر در روزمره استفاده می شوند، تمایز قائل شویم. برای مثال با درک کردن تمایز بین کلماتی نظیر Fault, Error, Failure که هر سه به معنای خطا هستند، میتوان با تکنیک های مختلف به مقابله با آن پرداخت. همچنین با درک کردن تمایز بین "در دسترس بودن" و "قابلیت اطمینان" میتوان این دو را در سیستم متعادل کرد. در این فصل این اصطلاحات را تشریح می کنیم.

اصطلاعات عمومی ایمنی

خطر[1]، ریسک، اقدامات کاهشی[2] و ریسک باقیمانده[3]

اصطلاحات "خطر" و "ریسک" در استانداردهای مختلف به طور متفاوتی استفاده می شوند. در ISO 26262-1 خطر را به عنوان "منبع بالقوه آسیب ناشی از رفتار نامناسب" تعریف می کند در حالی که در IEC 61508 و ISO 14971 همان اصطلاح را به عنوان "منبع بالقوه آسیب" بدون اشاره به رفتار نامناسب تعریف می کند.

در ادامه این کتاب از اصطلاحات به شرح زیر استفاده خواهم کرد.

کوه یخ خطر است. برخورد کشتی با این کوه یخ ریسک است. یکی از اقدامات کاهش ریسک این است که رنگ کوه زرد باشد تا بهتر دیده شود. اما حتی زمانی که رنگ کوه یخ زرد شود باز هم ریسک باقیمانده زمانی که کشتی در شب یا در مه ممکن است به آن برخورد کند وجود دارد.

[1] hazard
[2] mitigation
[3] Residual risk

به طور کلی من از اصطلاح "خطر" برای چیز غیرفعال در محیط استفاده می کنم که ممکن است علت ریسک باشد. ریسک ها خطرات بالقوه ای هستند که با رخدادن یک حادثه خطرناک[1] به صورت بالفعل درآمده و فعال شده باشند و باعث ایجاد شرایط خطرناک[2] شوند. وقتی ریسک شناسایی شد، اگر به اندازه کافی جدی باشد و از اندازه قابل قبول[3] بیشتر باشد، باید کاهش یابد. یعنی باید برای کاهش آن اقدام شود.

برای زدن مثالی که به بحث نرم افزارهای امبدد نزدیکتر باشد، حافظه استفاده شده توسط پردازنده میتواند یک خطر باشد. ریسک های مرتبط با آن میتواند شامل موارد زیر باشد:

ریسک : اثرات ثانویه پرتو های کیهانی

که ممکن است باعث چرخش بیتی[4] در حافظه پیکربندی سیستم شود و حالت عملکرد دستگاه را به طور غیرمنتظره ای تغییر دهد و به طور بالقوه موقعیت خطرناکی را بوجود آورد.

یکی از راه های کاهش این ریسک استفاده از بررسی خطا[5] و تصحیح[6] حافظه (ECC) می باشد. بنابراین خطاها می تواند شناسایی و چرخش های تک بیتی اصلاح شود.

ریسک های باقیمانده شامل این واقعیت است که همه خطاها توسط ECC شناسایی نمی شوند(به ویژه خطاهای 3 بیتی)، برنامه ممکن است به درستی وقفه[7] نشان دهنده خطای حافظه را مدیریت نکند و ECC ممکن است حافظه نهان[8] را پوشش ندهد.

ریسک : فرسودگی حافظه

یک برنامه کاربردی ممکن است نشت حافظه[9] داشته باشد و به دلیل عدم دسترسی به حافظه اضافی ممکن است به طور غیر منتظره ای از کار بیفتد و به طور بالقوه باعث ایجاد یک موقعیت خطرناک شود.

یکی از راه های کاهش این ریسک این است که هر برنامه کاربردی در هنگام راه اندازی حافظه کافی به صورت استاتیک رزرو کند و هرگز آن را آزاد نکند.

اما باز هم زمانی که شرایطی رخ دهد که هرگز در طول طراحی و آزمایش در نظر گرفته نشده باشد و مقدار حافظه استاتیک تخصیص یافته کافی نباشد، ریسک باقیمانده وجود دارد.

---

[1] Dangerous event
[2] Dangerous condition
[3] Tolerable risk
[4] bit-flip
[5] Error checking
[6] correcting
[7] interrupt
[8] cache
[9] memory leak

معمولا برای هر خطر چندین ریسک مرتبط و برای هر ریسک چندین روش کاهشی مرتبط وجود دارد.

در دسترس بودن[1]، قابلیت اطمینان[2] و قابلیت اعتماد[3]

هنگامی که یک زیر سیستم نرم افزاری فراخوانی می شود، ممکن است به یکی از این دو روش شکست بخورد: ممکن است اصلا به موقع پاسخ ندهد یا پاسخ بدهد اما پاسخ صحیح نباشد. عدم ارائه پاسخ در مدت زمان مشخص مشکل در دسترس بودن و ارائه به موقع پاسخ اشتباه را مشکل قابلیت اطمینان می نامیم.

افزایش در دسترس بودن عموما قابلیت اطمینان را کاهش می دهد و بالعکس. سروری را در نظر بگیرید که نوعی محاسبه را انجام می دهد. ما میتوانیم با داشتن دو سرور برای انجام محاسبات و مقایسه پاسخ آن ها قابلیت اطمینان سیستم را افزایش دهیم. یک تکنیک خوب این است که:

> "هنگامی که فرمول بسیار پیچیده باشد، ممکن است به صورت جبری برای محاسبه به دو یا چند روش مجزا تقسیم شود و دو یا چند مجموعه کارت ساخته شود. اگر از ثابت های یکسانی برای محاسبه هر مجموعه استفاده شود و اگر تحت این شرایط نتایج موافق هم باشد میتوان از صحت آن ها کاملا مطمئن شد"
>
> *چارلز بابیج[4] سال 1837 مرجع [1]*

هرچند که اضافه کردن سیستم باعث بهبود قابلیت اطمینان می شود اما ماهیت تکراری داشتن باعث کاهش دسترسی می شود، چرا که اگر یکی از دو سرور در دسترس نباشد کل سیستم در دسترس نیست. با توجه به این تضاد بین در دسترس بودن و قابلیت اطمینان، در طول فرآیند طراحی مهم است که بدانیم چه چیزی در آن طراحی باید مورد تاکید قرار گیرد و چگونه باید تعادل حاصل شود. این موضوع در صفحه 90 بیشتر بررسی شده است.

فکر کردن به این موضوع راحت است که قابلیت اطمینان برای داشتن ایمنی ضروری است حتی به قیمت از دست رفتن دسترسی. اما فکر کردن به سیستمی که در آن در دسترس بودن مهمتر است سخت تر ولی امکان پذیر است. در مرجع 2 متس هایمدال[5] یک مثال دور از ذهن از سیستمی را توصیف می کند که در آن هرگونه افزایش در قابلیت اطمینان ایمنی سیستم را کم می کند.

---

[1] Availability
[2] Reliability
[3] Dependability
[4] Charles Babbage
[5] Mats Heimdahl

توجه داشته باشید هر چند که کلاینت های یک سرور به طور کلی میتوانند در دسترس بودن یا نبودن سرور را با تنظیم یک Timeout اندازه گیری کنند اما اغلب اندازه گیری قابلیت اطمینان سرور غیر ممکن است. اگر کلاینت ها پاسخ صحیح را می دانستند احتمالا نیازی به فراخوانی سرور نداشتند.

بسیاری از برنامه های تست میتوانند قابلیت اطمینان واقعی را نیز آزمایش کنند و فقط در دسترس بودن را به طور گذرا بررسی می کنند.

در این کتاب من از واژه قابلیت اعتماد برای ترکیب دو عبارت در دسترس بودن و قابلیت اطمینان استفاده خواهم کرد که برای یک سیستم ترکیب هر دو مهم تر از هر یک به تنهایی می باشد. یک تعریف رایج از قابلیت اعتماد این است که یک سیستم قابل اعتماد است اگر خدماتی را ارائه دهد که به طور موجه قابل اعتماد باشد.

ایمنی عملکرد[1]

ایمنی را می توان به روش های مختلفی تامین کرد. لیزر موجود بر روی یک تجهیز انتقال مخابراتی را در نظر بگیرید: چنین لیزری یک خطر[2] است. یکی از خطرات مرتبط این است که اگر اپراتور فیبر را جدا کند و به فرستنده نگاه کند، ممکن است به چشم آسیب برساند.

یکی از راه های کاهش این خطر، نصب فرستنده رو به پایین و نزدیک به پایین قفسه لیزر می باشد. موقعیتی که امکان تراز کردن چشم با فرستنده را برای اپراتور غیرممکن می کند. این امر با تکیه بر یک سیستم ایمنی غیرفعال[3] ایمن خواهد بود.

یکی دیگر از راه های کاهش خطر این است که نرم افزاری بر کانکتور لیزر نظارت داشته باشد و در صورت حذف فیبر، لیزر را خاموش کند. برای ایمن بودن این سیستم، نیاز به یک بخش فعال[4] نرم افزاری وجود دارد تا سیستم به درستی کار کند. نرم افزار باید همیشه کانکتور را زیر نظر داشته باشد و برای جلوگیری از آسیب، لیزر را در عرض چند میلی ثانیه خاموش کند.

دومین مورد از این تکنیک ها بیانگر ایمنی عملکرد می باشد. چیزی است که باید به کار خود ادامه دهد تا سیستم را ایمن نگه دارد.

[1] Functional Safety
[2] hazard
[3] passive
[4] active

خطا[1] اشکال[2] و خرابی[3]

IEEE 24765 بین این سه عبارت به روش زیر تمایز قائل می شود.

یک برنامه نویس ممکن است با تایپ چیزی ناخواسته در یک ویرایشگر، خطایی را به برنامه وارد کند. در واقع خطا یک نقص غیرفعال است و به آن باگ نیز گفته می شود.[4]

گاهی اوقات یک خطا ممکن است باعث شود برنامه به گونه ای عمل کند که نتیجه ناخواسته ای ایجاد کند، در واقع خطا باعث ایجاد اشکال شده است.

گاهی اوقات رخ دادن اشکال باعث خرابی سیستم می شود. طبق استاندارد EN 50128 "خرابی زمانی اتفاق می افتد که یک جزء عملیاتی سیستم دیگر نتواند عملکرد صحیح خود را اجرا کند. در واقع خرابی یک اثر قابل مشاهده بیرونی است که از اشکال یا خطای درونی سیستم ناشی می شود. البته همیشه یک اشکال یا خطا لزوما منجر به خرابی سیستم نمی شود."

نکته مهم در متن بیان شده توسط استاندارد EN 50128 کلمه "گاهی اوقات" است. یک خطا همیشه باعث بروز اشکال نمی شود. مثلا اگر برنامه نویس قصد داشته باشد تا متغیر محلی[5] زیر را تایپ کند :

```
char x[10];
```

اما به صورت اتفاقی عبارت زیر را تایپ کند :

```
char x[11];
```

بعید است که باعث ایجاد اشکال شود مگر اینکه حافظه سیستم بسیار کم باشد. اما در هر صورت این یک خطا می باشد.

اگر برنامه نویس سیستمی ایجاد کند که هر 10 ثانیه یکبار فایلی را باز کند، اما نتواند آن فایل ها را ببندد. این خطا در توصیف گر فایل[6] باعث ایجاد اشکال هر 10 ثانیه یکبار می شود هر چند که این اشکال برای مدتی باعث ایجاد خرابی نمی شود – 90 روز برای کامپیوتری که این متن با آن تایپ شده است طول می کشد تا

---

[1] Fault
[2] Error
[3] Failure

[4] واژه Fault در کاربردهای سخت‌افزاری "عیب" و در کاربردهای نرم‌افزاری "خطا" در نظر گرفته می‌شود.

[5] local
[6] File descriptor

توصیف گر فایل حداکثر 777.101 فایل باز را اجازه دهد. برای مثال واقعی به حکایت 11 در صفحه 123 مراجعه کنید.

همانطور که در شکل 2.1 نشان داده شده است، یک اشکال در یک سطح از سیستم ممکن است باعث خطا در سطح دیگر سیستم شود. در این شکل نشان می دهد که یک خطا در سطح آموزشی فرد[1] باعث شده است که او باگی را در کد ایجاد کند (او با وارد کردن یک خطا در کد باعث بوجود آمدن اشکال می شود) فرد به خاطر این اشتباه شغل خود را از دست می دهد (حداقل تا آنجا که به فرد مربوط می شود) خطا در سطوح مختلف کد ادامه می یابد و باعث خرابی در سیستم کلی می شود.

تمایز بین خطا ها، اشکال ها و خرابی ها به ما این امکان را می دهد که به روش های مختلف با آنها مقابله کنیم. ما می خواهیم تعداد خطا های وارد شده به سیستم را کاهش دهیم و قبل از اینکه به اشکال تبدیل شوند آن ها را شناسایی و حذف کنیم و از به بار آمدن خرابی جلوگیری کنیم و در صورت بروز خرابی، آن را به ایمن ترین شکل ممکن مدیریت کنیم.

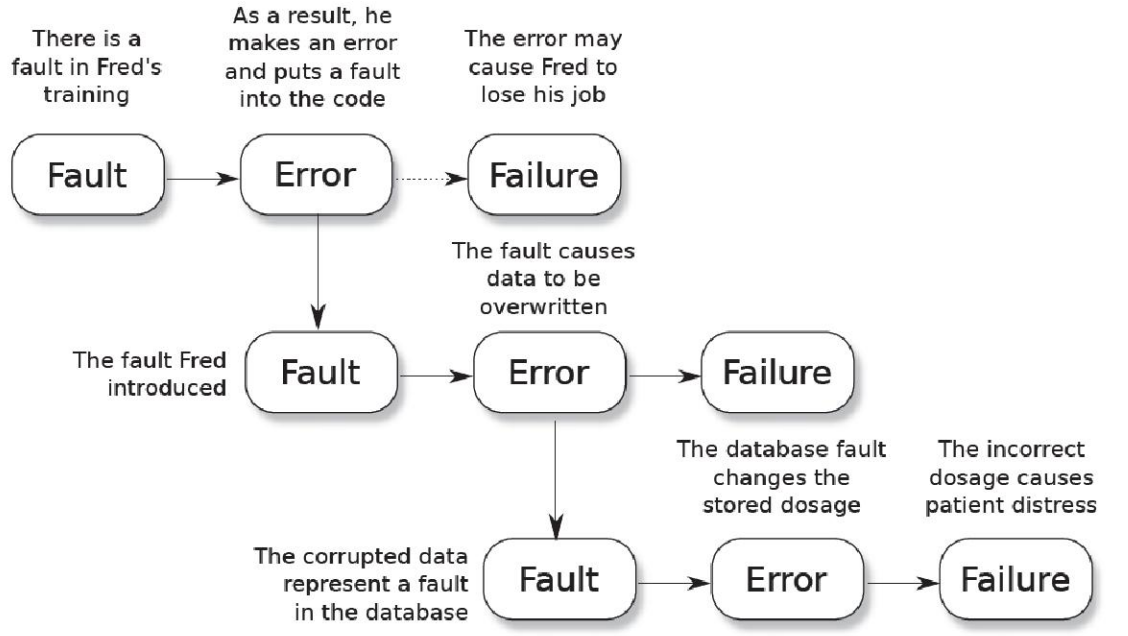


Figure 2.1 Faults, errors, and failures.

[1] Fred

زبان های رسمی[1]، نیمه رسمی[2] و غیر رسمی[3]

زبان های طبیعی – انگلیسی، ماندرین، آلمانی، اینوکتیتوت – مبهم و ظریف هستند. چرا که مرجع مکرر داده شده برای یک کارمند سابق، "خوش شانس خواهید بود که این شخص برای شما کار کند"، ابهام زبان انگلیسی را نشان می دهد. و مرجع شغلی دیگر، ظاهراً از وقتی که برای موقعیت شغلی استادی فلسفه درخواست داده شده است "این شخص دست خط بسیار خوبی دارد" ظرافت زبان انگلیسی را نشان می دهد.

چنین زبانی غیر رسمی در نظر گرفته می شود زیرا هم گرامر و هم معنای آن مبهم هستند.

می توان نحو[4] یک زبان را رسمی کرد، در حالی که انعطاف پذیری معنایی غیررسمی حفظ شود. چنین زبان هایی به عنوان نیمه رسمی شناخته می شوند. زبان مدل سازی جهانی[5] و زبان مدل سازی سیستم[6] هر دو اغلب به صورت نیمه رسمی استفاده می شوند. در این زبان ها نحو به خوبی تعریف شده است به این صورت که پیکان مثلثی همیشه به معنای تعمیم[7] و شکل چوب دستی همیشه به معنای بازیگر[8] می باشد. با این حال زمانی که یک بلوک در نمودار SysML با عنوان "سیستم ترمز" مشخص می شود، هیچ تعریفی از معنای آن وجود ندارد و به صورت استفاده روزمره انگلیسی فرض می شود.

زبان ریاضیات به ما این امکان را می دهد که ایده هایی در مورد جهان بیان کنیم. ریاضیات متوجه نمی شود که مثلث واقعا چیست اما می تواند ثابت کند که مجموع زوایای یک مثلث برابر 180 درجه است. اصل دوگانگی در هندسه بر انتزاع از اشیاء واقعی تأکید می کند و می گوید که با چند محدودیت می توانیم "نقطه" و "خط" را به هم تبدیل کنیم به طوری که تمام قضایای هندسی درست باقی بمانند. برای مثال دو نقطه یک خط را تعریف می کنند.

اگر بتوانیم طرح‌های خود را به زبان ریاضی (یعنی رسمی) بیان کنیم، باید بتوان درباره مفاهیم جهانی استدلال کنیم. با توجه به طراحی سیستم خود، ما باید بتوانیم از نظر ریاضی ثابت کنیم که برای هر ترکیب و دنباله ای از درخواست ها و وقفه ها، سیستم به پیشرفت خود ادامه می دهد و قفل نمی شود.

---

[1] Formal
[2] Semi-formal
[3] Informal
[4] syntax
[5] universal modeling language (UML)
[6] systems modeling language (SysML)
[7] generalisation
[8] Actor

اثبات عملکرد صحیح یک سیستم با آزمایش کردن آن متفاوت است: آزمایش کردن نشان می دهد که برای مجموعه خاصی از شرایطی که در طول آزمایش رخ داده است سیستم درست کار می کند، در حالی که اثبات نشان می دهد که تحت همه شرایط سیستم درست کار می کند.

ایمنی[1]، زندگی[2] و انصاف[3]

به راحتی می توان بین ویژگی های ایمنی، زنده بودن و انصاف یک سیستم تمایز قائل شد، اصطلاحاتی که برای اولین بار توسط لزلی لامپورت[4] در سال 1983 به طور غیررسمی ابداع شد:

ویژگی ایمنی یک سیستم می گوید که آن سیستم هرگز کار بدی انجام نمی دهد. برای مثال سیستم هرگز پیام A را ارسال نمی کند مگر اینکه پیام B را قبل از آن دریافت کرده باشد.

ویژگی زنده بودن یک سیستم می گوید که آن سیستم در نهایت کار خوبی انجام می دهد. یعنی به جلو حرکت می کند و دچار مرگ یا توقف نمی شود. برای مثال اگر پیام B دریافت شود سپس همیشه پیام A در نهایت ارسال می شود.

ویژگی انصاف یک سیستم می گوید که اگر سیستم اقداماتی را انجام دهد، آن اقدامات در نهایت با موفقیت انجام خواهد شد. یعنی سیستم اجازه نمی دهد یک فعالیت قدرت را در دست بگیرد و فعالیت های دیگر برای همیشه بدون پاسخ باقی بماند.

انواع مختلفی از انصاف وجود دارد که از انصاف قوی (اگر سیستم به دفعات بی نهایت برای انجام فعالیت X تلاش کند، اغلب اوقات بی نهایت بار موفق خواهد شد)گرفته تا انصاف ضعیف تر (مثلاً اگر سیستم به طور مداوم X را امتحان کند، حداقل یک بار موفق خواهد شد) وجود دارد.

برای معتبر بودن یک ویژگی ایمنی، حالت های محدود (هر چقدر هم که بزرگ باشد) وجود دارد که باید بررسی شود تا مشخص شود آن ویژگی برای یک سیستم خاص وجود دارد یا خیر. حتی در صورتی که بررسی تمام حالت ها عملا غیر ممکن باشد اما از نظر تئوری قابل انجام است. برای ویژگی زنده بودن تعداد حالت های احتمالی که بایستی بررسی شوند به علت وجود عبارت "در نهایت" در تعریف آن، بینهایت است.

---

[1] safety
[2] liveness
[3] fairness
[4] Leslie Lamport

بازیابی اشکال[1] معکوس[2] و مستقیم[3]

در زمانی که یک خطا باعث بوجود آمدن اشکال می شود، آن اشکال گاهی اوقات می تواند به دام بیوفتد و برخی از انواع مختلف بازیابی میتواند قبل از وقوع خرابی انجام شود. دو شکل اصلی بازیابی اشکال عبارتند از:

بازیابی اشکال معکوس

هنگامی که یک اشکال کشف می شود، سیستم به حالت ذخیره شده قبلی باز می گردد که پایدار و معقول باشد (یا اینگونه فرض می شود). این مسئله که آن ورودی که ظاهراً باعث خطا شده است مجددا اعمال شود یا کنار گذاشته شود، و این مسئله چگونه به کاربران اطلاع رسانی شود، بین پیاده سازی های مختلف متفاوت است. یکی از معایب بازیابی اشکال معکوس، نیاز به ذخیره اطلاعات وضعیت سیستم قبل از رسیدگی به درخواست ورودی است تا در صورت نیاز به بازگشت مجدد، در دسترس باشد.

بازیابی اشکال مستقیم

در این نوع بازیابی اشکال، مشکل اصلی بازیابی اشکال معکوس – یعنی نیاز به ذخیره آخرین وضعیت سیستم – با بردن سیستم به یک حالت معقول از پیش تعریف شده و مستقل از حالت قبلی و ورودی ایجاد کننده اشکال حل می شود.

سیستم های تصادفی

در سال 2010 آزمایشی در دریای شمال، در سواحل شرقی بریتانیا انجام شد. کشتی 500 تنی THV Galatea از منطقه ای عبور کرد که سیگنال های GPS عمداً در آن مسدود شده بود. همانطور که انتظار می رفت، موقعیت کشتی نامشخص شد (نمایشگر روی پل نشان می داد که با سرعت مافوق صوت از وسط نروژ تا وسط ایرلند حرکت می کند!)، اما به طور غیر منتظره، رادار کشتی نیز از کار افتاد. تیم آزمایش کننده با شرکت سازنده رادار تماس گرفت و به آنها گفته شد که GPS در آن استفاده نشده است. آنها سپس با سازندگان قطعات موجود در رادار تماس گرفتند و دریافتند که یک قطعه کوچک از سیگنال های زمان بندی GPS استفاده می کند.

[1] Error Recovery
[2] Backward
[3] Forward

رادار یک سیستم تصادفی است. سازنده قطعات کوچک می دانست که هر سیستمی که در آن گنجانده شده است به یکپارچگی سیگنال های GPS متکی است، اما از سیستم راداری که قطعه در آن گنجانده شده است آگاه نبود. همچنین طراح رادار هم از وابستگی آن قطعه به GPS بی اطلاع بود. هیچکس تصویر کاملی نداشت و سیستمی با وابستگی های تصادفی ساخته شده بود. این سیستم در مواجهه با پارازیت GPS آزمایش نمی شد، زیرا اتکا به GPS ناشناخته بود.

در حال حاضر اغلب سیستم های مبتنی بر نرم افزار امبدد با افزایش پیچیدگی سیستم های تصادفی هستند. شاید طراحان یک سیستم کامل از اینکه یک قطعه تک وظیفه ای[1] می باشد یا مقدار بازگشتی آن از اندازه حافظه پشته[2] اختصاص داده شده به آن بیشتر باشد، بی خبر بوده اند.

اصطلاحات ویژه نرم افزار

نرم افزار

شاید عجیب بنظر برسد در کتابی که به نرم افزارهای امبدد اختصاص داده شده است، بپرسید نرم افزار چیست. افراد مستقیما احساس می کنند می توانند بین نرم افزار و سخت افزار تفاوت قائل شوند. اگر هنگام افتادن روی پا درد داشته باشد، سخت افزار است.

اما در مورد مدار مجتمع با کاربرد خاص[3] که توسط یک ابزار نرم افزاری طراحی شده است چطور؟ در مورد 5000 خط کد C که توسط یک ابزار مدل سازی تولید می شود چطور؟

در مرجع [3] نسبت به تفاوت میان نرم افزار و سخت افزار، با توجه به اینکه نرم افزار ذاتا پیچیده تر از سخت افزار می باشد، به طور عمل گرایانه برخورد می کند:

به عنوان یک قاعده کلی، پیشنهاد می کنیم اگر وسیله ای حالت های عملکردی ذخیره شده داخلی کمی دارد، به طوری که میتوان در آزمایش عملی همه آن ها را پوشش داد، ممکن است بهتر باشد آن را به عنوان سخت افزار در نظر بگیریم و با تجزیه و تحلیل طراحی و آزمایش وسیله کامل شده از جمله آزمایش ترکیب کردن تمام

[1] single-threaded
[2] Stack
[3] Application-Specific Integrated Circuit (ASIC)

حالت های مختلف ورودی و حالت های مختلف سیستم نشان دهیم که الزامات ایمنی مورد نیاز را برآورده می کند.

این تعریفی است که ما در این کتاب از آن استفاده خواهیم کرد. اگر سیستم به اندازه کافی ساده باشد که بتوان آن را به طور کامل آزمایش کرد، آن گاه به عنوان سخت افزار و در غیر این صورت به عنوان نرم افزار در نظر گرفته می شود.

توجه داشته باشید که تعریف فوق یک رابطه یک طرفه است و برعکس آن لزوما درست نیست. قطعا سیستم های امبدد در نظر گرفته شده در این کتاب برای آزمایش جامع بسیار پیچیده هستند و بنابراین به عنوان نرم افزار تلقی می شوند. هرچند که آزمایش کامل برخی از وسیله های سخت افزاری سالیان سال است که عملا غیر ممکن می باشد. برای مثال یک حافظه RAM کوچک 64 کیلوبیتی که برای استانداردهای امروزی بسیار کوچک است $2^{64x1024}$ حالت مختلف دارد. این عدد دارای 19728 رقم می باشد و طبق تعریف مرجع [4] می تواند نرم افزار در نظر گرفته شود. اما من این تفسیر را نادیده خواهم گرفت.

جنبه دیگری از نرم افزار وجود دارد که اغلب نادیده گرفته می شود: داده های پیکربندی[1]

مقررات کمیسیون (EC) شماره 2008/482 اتحادیه اروپا، به صراحت واژه "نرم افزار" را برای داده های وابسته به پیکربندی مرتبط آن تعریف می کند:

"نرم افزار" به معنی برنامه های کامپیوتری و داده های پیکربندی مربوط با آن است....

داده‌های پیکربندی به معنای داده‌هایی است که یک سیستم نرم‌افزاری عمومی را برای یک نمونه خاص از استفاده از آن پیکربندی می‌کند.

پس از سقوط یک هواپیمای ترابری نظامی ایرباس مدل A400M در 9 می 2015، یکی از مدیران گروه ایرباس در مصاحبه ای با روزنامه آلمانی هندلزبلت[2] تایید کرد که این سقوط به دلیل یک پیکربندی نرم افزاری معیوب[3] بوده است. به طور خاص، مدیر اجرایی گفت که "خطا در خود کد نبود، بلکه در تنظیمات پیکربندی برنامه ریزی شده در واحد کنترل الکترونیکی[4] موتورها بود".

[1] configuration data
[2] Handelsblatt
[3] Faulty
[4] ECU

اعتبار سنجی[1] یک برنامه همراه با داده های پیکربندی آن به دلیل مشکلات ترکیب شدن مرتبط با پیکربندی به خوبی درک نشده است. اگر یک برنامه 5 مورد برای پیکربندی داشته باشد و هر مورد دارای 3 مقدار مختلف باشد، در این صورت $3^5=243$ حالت مختلف برای تایید پیکربندی وجود دارد. به بحث تست ترکیبی در صفحه 257 مراجعه کنید. اهمیت گنجاندن داده های پیکربندی در مرحله راستی آزمایی برنامه به طور فزاینده ای بیشتر شناخته می شود و احتمالاً استانداردهای جدیدی در چند سال آینده در این رابطه ظاهر می شوند.

هایزن باگ[2] و بور باگ[3]

نیلز بور[4] یک مدل جامد و توپی شکل از ذرات سازنده اتم را توصیف کرد و جایزه نوبل سال 1922 را برای این کار خود دریافت کرد. ورنر هایزنبرگ[5] شاگرد و همکار بور، این مدل را تقویت کرد تا اتم ها را بسیار نامشخص کند، به طوری که یک مدل احتمالی از جایی که الکترون ها، نوترون ها و پروتن ها ممکن است باشند را توصیف کرد. این کار باعث شد تا هایزنبرگ در سال 1932 جایزه نوبل را دریافت کند.

ریشه یابی کلمات هایزن باگ و بور باگ نامشخص است، اگر چه جیم گری[6] از اصطلاح هایزن باگ در سال 1983 استفاده کرد. اعتقاد بر این است که شروع استفاده از این اصطلاحات از اوایل دهه 1980 می باشد.

مدل بور همانطور که در مورد اتم ها وجود دارد، در مورد باگ ها نیز وجود دارد. بور باگ یک باگ به خوبی تعریف شده و جامد با ویژگی هایی است که وقتی کد دیباگ برای یافتن باگ اضافه می شود، تغییر نمی کند. هر بار که یک خط از کد خاص با مقادیر خاصی از متغیرهای ورودی اجرا می شود، سیستم بد رفتار می کند.

در مقابل، هایزن باگ، یک باگ گریزان و گمراه کننده است که ظاهر و ناپدید می شود به طوری که آن را بسیار ناپایدار می کند. هایزن باگ به دلیل مشکلات ریز زمان بندی ایجاد می شود. برای مثال رشته A که روی هسته پردازشی 0 اجرا می شود، کاری را انجام می دهد که روی رشته B در حال اجرا روی هسته پردازشی 1 تاثیر می گذارد، اگر و فقط اگر وقفه ای از نوع C رخ دهد و ...

---

[1] Validating
[2] Heisenbug
[3] Bohrbug
[4] Niels Bohr
[5] Werner Heisenberg
[6] Jim Gray

هایزن باگ به عنوان باگ غیر قابل تولید مجدد توسط گروه های انجام آزمایش گزارش می شود: "در هشتمین باری که تست شماره 1243 اجرا شد، سیستم از کار افتاد. از آن زمان تا کنون ما آن تست را بارها انجام دادیم اما در هیچ یک شکستی رخ نداد. هیچ حافظه خالی در دسترس نیست. لطفا باگ را برطرف کنید"

یکی از ویژگی های هایزن باگ این است که یک خطا می تواند منجر به خرابی های مختلف شود و خرابی ها می توانند بسیار دیرتر از زمانی که خطا بوجود آمده بود (و در تکه کدی کاملا متفاوت) رخ دهند. این همان چیزی است که دیباگ کردن را دشوار می کند: آزمایش خرابی ها را شناسایی می کند، در حالی که توسعه دهندگان باید خطاها را برطرف کنند. برای همه خطاها بویژه برای هایزن باگ، مسیر بین خطا و خرابی چند به چند است و این امر ردیابی خطا را با توجه به خرابی بسیار دشوار می کند. شکل 2.2 را ببینید.

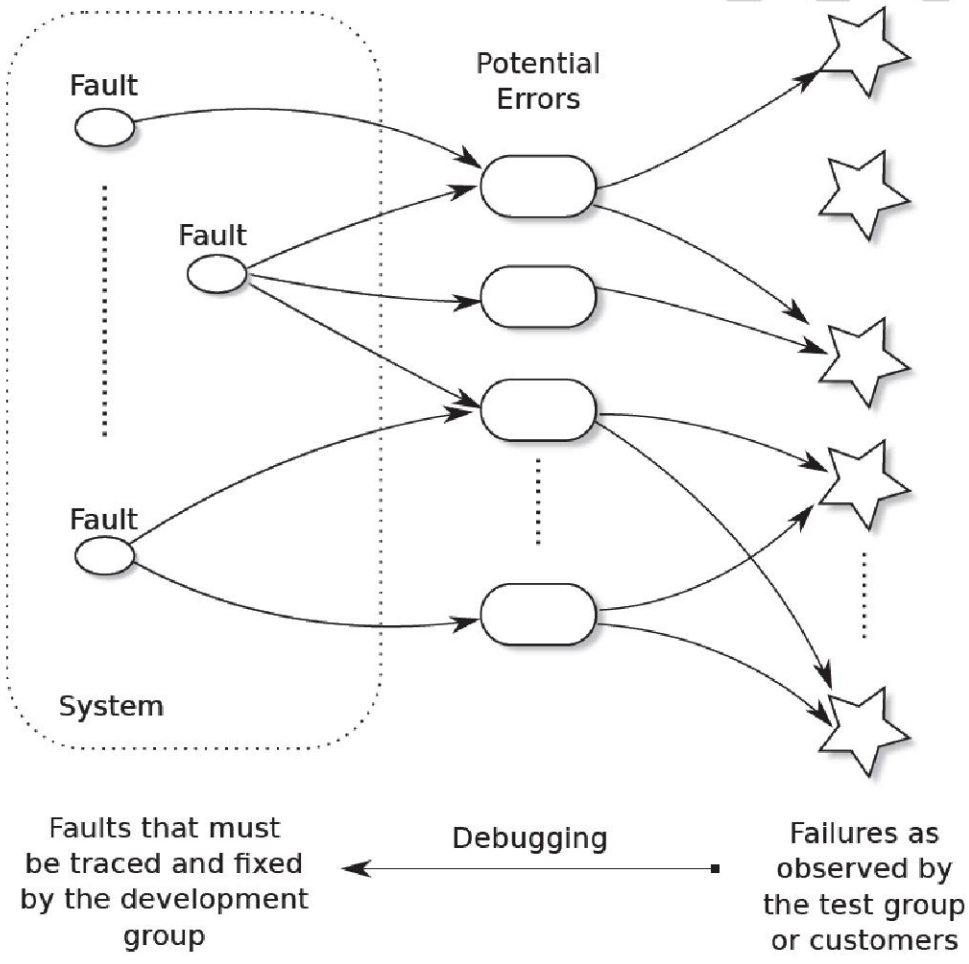


Figure 2.2 Tracing the failure to the fault.

در سطحی که توسط یک آزمایش کننده قابل مشاهده است، به نظر می رسد بسیاری از خرابی ها یکسان هستند. اشکال هایی که باعث خرابی می شوند هم برای آزمایش کننده و هم برای برنامه نویس نامرئی هستند و خطاهای مختلف ممکن است باعث ایجاد اشکال های اساسا یکسان شوند.

جیم گری به این نکته اشاره کرد که کاربران در بقیه زمینه ها به جز در زمینه سیستم های بحرانی ایمنی، هایزن باگ را به بور باگ ترجیح می دهند. یعنی آنها ترجیح می دهند که یک خرابی تصادفی در حدود یک بار در ماه اتفاق بیوفتد تا اینکه هر بار یک دستور خاص در هنگام ورود به زمینه خاص دچار خرابی شود. به همین دلیل تحقیقاتی از جمله در مرجع شماره [4] در مورد تبدیل بور باگ به هایزن باگ انجام شده است. به عنوان مثال، اگر بیشتر از مقدار مورد نیاز برای یک شی، حافظه تخصیص داده شود و حافظه به صورت احتمالی تخصیص داده شود، در این صورت سر ریز حافظه[1] به جای بور باگ به هایزن باگ تبدیل می شود.

ماهیت دست نیافتنی هایزن باگ ها رفع آن ها را به ویژه دشوار می کند، و برخی از محققان، به ویژه آندریا بور[2] (به عنوان مثال به مرجع [6] مراجعه کنید)، دریافته اند که تلاش برای رفع هایزن باگ ها به طور کلی وضعیت را بدتر از قبل می کند.

برنامه نویسان شوخ طبع اصطلاحات بور باگ و هایزن باگ را در جهت های خنده دار تری گسترش دادند. به عنوان مثال شرودینگ باگ[3] که به نام اروین شرودینگر[4] (برنده جایزه نوبل در سال 1993) نامگذاری شده است. شرودینگ باگ خطایی است که سال ها در کد وجود دارد و هرگز در این زمینه مشکلی ایجاد نکرده است. یک روز در حین خواندن کد، برنامه نویس خبر باگ را دریافت می کند. سپس گزارش های میدانی شروع به سرازیر شدن می کند که این باگ پیدا شده است. دقیقا مانند گربه شرودینگر که ناگهان در محصول ارسال شده بیدار می شود.

بلادرنگ[5]

اصطلاح بلادرنگ شاید یکی از گیج کننده ترین و بد استفاده ترین اصطلاحات در توسعه نرم افزار باشد. اگر بخواهیم ساده بیان کنیم یک سیستم در صورتی بلادرنگ است که در یک محدودیت زمانی خاصی به رفتارهای مورد نیاز ما پاسخ دهد.

---

[1] Overflow
[2] Andrea Borr
[3] Schrodingbug
[4] Erwin Schrodinger
[5] Real-Time

از این رو، برنامه‌ای که آب و هوا را از روی داده‌های موجود پیش‌بینی می‌کند، یک برنامه بلادرنگ است، زیرا یک پیش‌بینی عالی برای یک دوره 12 ساعت آینده، اگر تولید آن 14 ساعت طول بکشد، بی‌فایده است. به طور مشابه، محاسبات حقوق و دستمزد که توسط شرکت ها برای پرداخت به کارکنان خود انجام می شود، بلادرنگ است زیرا عدم تکمیل به موقع آنها می تواند منجر به نارضایتی کارکنان شود.

در دنیای امبدد، سیستمی بلادرنگ است که بخشی از زمانی که پاسخ خواهد داد را به طور قراردادی مشخص کند. مثلا این سیستم سنسور را در هر 2±10 میلی ثانیه می خواند و داده های پردازش شده را 5 میلی ثانیه بعد از خواندن در دسترس قرار می دهد.

باور غلط 1 : بلادرنگ به معنای سریع بودن است

سیستم‌های بلادرنگ نسبت به سیستم‌های دارای درنگ کندتر هستند، زیرا صرف نظر از زمان‌بندی سایر زیرسیستم‌ها، نیازمند بافر زمانی است تا اطمینان حاصل شود که ضرب‌الاجل‌ها همیشه رعایت می‌شوند.

برنامه نویسی به صورت قراردادی

مفهوم برنامه نویسی قراردادی یا توسعه اولین برنامه نویسی به صورت قراردادی توسط برتراند مایر[1] برای زبان برنامه نویسی ایفل[2] در اواخر دهه 1980 ابداع شد و اکنون در زبان های دیگر مانند زبان D و Ada به صورت یک قرارداد رسمی سه بخشی، بین کلاینت و سرور در دسترس است:

1. کلاینت موظف است اطمینان حاصل کند که درخواستش با پیش شرط های سرور مطابقت دارد. به عنوان مثال، پیش شرط برای فراخوانی تابعی که موردی را به صف سرور اضافه کند ممکن است این باشد که صف از قبل پر نشده باشد.
2. سرور تضمین می‌کند که شرایط خاص برای بعد از پاسخ آن به کلاینت اعمال خواهد شد. در مورد افزودن مورد جدیدی به صف سرور، این واقعیت وجود دارد که صف دقیقاً یک عنصر بیشتر از قبل دارد و آخرین عنصر در صف همان عنصری است که کلاینت ارائه کرده است.

---

[1] Bertrand Meyer

[2] توجه کنید که نرم افزار ایفل حق کپی رایت طراحی به صورت قراردادی را دارد

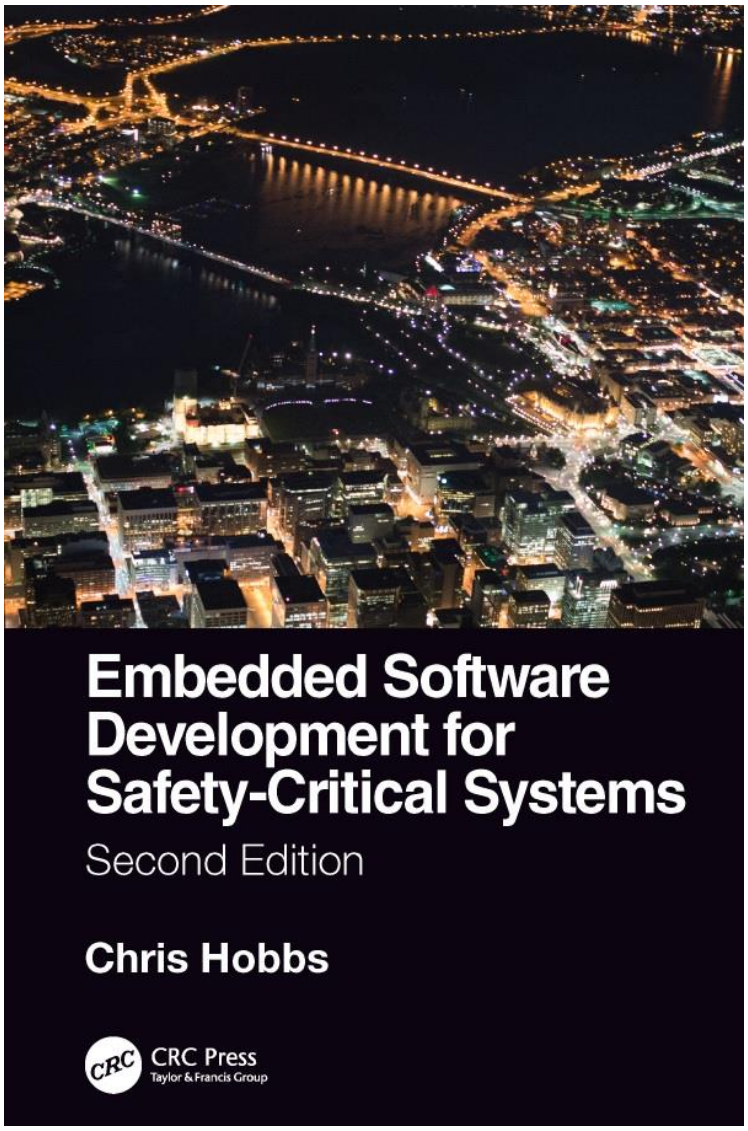
3. سرور تضمین می کند که برخی از متغیرهای داخلی در هنگام خروج اعمال می شود تا برقراری ارتباط بعدی به آن نیز به درستی پاسخ داده شود.

اگر قراردادها هنگام نوشتن برنامه در سرور کدنویسی شوند، نه تنها می توان آنها را در زمان اجرا بررسی کرد تا یافتن باگ ها دشوار باشد، بلکه می توان از آنها برای کمک به ابزارهای تجزیه و تحلیل استاتیک در کارشان استفاده کرد. مرجع [6] و فصل 18 را ببینید.

برای مثال در هنگام اضافه شدن موردی به صف سرور، یک ابزار تجزیه و تحلیل استاتیک می تواند کل زیرساخت های کد را تجزیه و تحلیل کند تا مشخص کند که آیا ممکن است زمانی که صف از قبل پر شده است، تابع فراخوانی شود یا خیر.

مراجع

1. C. Babbage, “On the Mathematical Powers of the Calculating Engine." (Published in Origins of Digital Computers: Selected Papers (ed. B. Randell) Springer-Verlag, 1973), 1837
2. M. P. E. Heimdahl, “Safety and software intensive systems: Challenges old and new," in 2007 Future of Software Engineering, FOSE '07, (Washington, DC, USA), pp. 137{152, IEEE Computer Society, 2007
3. Rail Safety and Standards Board, “Engineering Safety Management (The Yellow Book)," 2007. Available from http://www.yellowbook-rail.org.uk
4. E. D. Berger, “Software needs seatbelts and airbags," Commun. ACM, vol. 55, pp. 48{53, Sept. 2012
5. A. J. Borr and C. Wilhelmy, \Highly-Available Data Services for UNIX Client-Server Networks: Why Fault Tolerant Hardware Isn't the Answer," in Hardware and Software Architectures for Fault Tolerance, pp. 285{304, 1993.
6. R. Ceballos, R. M. Gasca, and D. Borrego, “Constraint Satisfaction Techniques for Diagnosing Errors in Design by Contract Software," in Proceedings of the 2005 conference on speci_cation and veri_cation of component-based systems, SAVCBS '05, (New York, NY, USA), ACM, 2005



این نسخه واترمارک شده شامل دو فصل اول بود

جهت خرید نسخه کامل به سایت الکترو ولت مراجعه نمایید

لینک دریافت از فروشگاه الکترو ولت