

دانشگاه صنعتي امیرکبیر  
(پلی تکنیک تهران)

دانشكده مهندسی کامپیوتر و فن­­آوری ­اطلاعات

پایان‌نامه کارشناسی

گرایش نرم­افزار

عنوان

طراحی و پیاده‌سازی ابزاری به منظور اِعمال خط مشی امنیتیِ عدم تداخل مبتنی بر روش بازنویسی برنامه

نگارش

سید محمدمهدی احمدپناه

استاد راهنما

جناب آقای دکتر مهران سلیمان فلاح

شهریور 1394

اينجانب سید محمدمهدی احمدپناه متعهد مي‌شوم كه مطالب مندرج در اين پايان نامه حاصل كار پژوهشي اينجانب تحت نظارت و راهنمايي اساتيد دانشگاه صنعتي اميركبير بوده و به دستاوردهاي ديگران كه در اين پژوهش از آنها استفاده شده است، مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذكر گرديده است. اين پایان نامه قبلاً براي احراز هيچ مدرك هم‌سطح يا بالاتر ارائه نگرديده است.

در صورت اثبات تخلف در هر زمان، مدرك تحصيلي صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پيگيري قانوني خواهد داشت.

كليه نتايج و حقوق حاصل از اين پایان نامه متعلق به دانشگاه صنعتي اميركبير مي‌باشد. هرگونه استفاده از نتايج علمي و عملي، واگذاري اطلاعات به ديگران يا چاپ و تكثير، نسخه‌برداري، ترجمه و اقتباس از اين پایان نامه بدون موافقت كتبي دانشگاه صنعتي اميركبير ممنوع است.   
نقل مطالب با ذكر مآخذ بلامانع است.

سید محمدمهدی احمدپناه

امضا

تقدیم به پدرم

کوهی استوار و حامی من در طول تمام زندگی 

تقدیم به مادرم

سنگ صبوری که الفبای زندگی به من آموخت

تقدیم به خواهر و برادرم

همراهان همیشگی و پشتوانه های زندگیم

**تقدیر و تشکر:**

سپاس خدای را که سخنوران، در ستودن او بمانند و شمارندگان، شمردن نعمت های او ندانند و کوشندگان، حق او را گزاردن نتوانند. سلام و درود بر محمّد و خاندان پاك او، طاهران معصوم، هم آنان که وجودمان وام‌دار وجودشان است.

بدون شک جایگاه و منزلت معلم، بالاتر از آن است که در مقام قدردانی از زحمات بی شائبه‌ او، با زبان قاصر و دست ناتوان، چیزی بنگارم. اما از آنجا که تجلیل از معلم، سپاس از انسانی است که هدف آفرینش را تامین می‌کند، به رسم ادب دست به قلم برده‌ام، باشد که این خردترین بخشی از زحمات آنان را سپاس گوید.

از پدر و مادر مهربانم، این دو معلم بزرگوار که همواره بر کوتاهی من، قلم عفو کشیده و کریمانه از کنار غفلت‌های گذشته‌اند و در تمام عرصه‌های زندگی یار و یاورم بوده‌اند؛

از استاد فرزانه و دلسوز، جناب آقای دکتر سلیمان فلاح که در کمال سعه صدر، با حسن خلق و فروتنی، از هیچ کمکی در این عرصه بر من دریغ نداشتند؛

از اساتید محترم، جناب آقای دکتر محمدرضا رزازی و جناب آقای دکتر بهمن پوروطن که زحمت داوری این پایان‌نامه را متقبل شدند؛

و در پایان، از حمایت‌ها و کمک‌های دوستان عزیزم، آقایان....

**کمال تشکر و قدردانی را دارم.**

چكيده

شسیشس

واژه‌های کلیدی:

امنیت جریان اطلاعات؛ خط مشی عدم تداخل؛ گراف وابستگی برنامه؛ بازنویسی برنامه

|  |  |
| --- | --- |
| فهرست عناوین | صفحه |

[1 ‌ فصل اول مقدمه 1](#_Toc428959128)

[1‌.1‌ نمای کلی پروژه 2](#_Toc428959129)

[1.1.1‌ هسته تشخیص و استخراج موضوع متن 2](#_Toc428959130)

2.[1‌.1‌ جمع آوری و نمایش متن وب‌سایت‌ها 2](#_Toc428959131)

[1‌.1‌.3‌ واحد ارتباط با پایگاه داده 2](#_Toc428959132)

[4.1.1 پایگاه داده متن‌ها به همراه موضوعات استخراج شده آن‌ها 3](#_Toc428959133)

[1‌.2‌ ساختار پایان نامه 4](#_Toc428959134)

[2 فصل دوم تشخیص و استخراج موضوع متن‌های کوتاه 5](#_Toc428959135)

[2‌.1‌ یادگیری با نظارت 6](#_Toc428959136)

[2‌.1‌.1‌ روش DOR 7](#_Toc428959137)

[2‌.1‌.2‌ روش TCOR 8](#_Toc428959138)

[2‌.1‌.3‌ روش‌های دسته بندی متون 9](#_Toc428959139)

[2‌.2‌ یادگیری بدون نظارت 11](#_Toc428959140)

[2‌.2‌.1‌ دسته بندی‌ متن‌های کوتاه بر اساس آنالیز معنایی 12](#_Toc428959141)

[2‌.2‌.2‌ مدل تخصیص پنهان دیریکله 13](#_Toc428959142)

[3 فصل سوم طراحی و پیاده سازی هسته‌ ی اصلی و پایگاه داده فارسی 16](#_Toc428959143)

[3‌.1‌ طراحی و پیاده سازی پایگاه داده 16](#_Toc428959144)

[1.1.3 جدول اطلاعات کاربران 16](#_Toc428959145)

[3‌.1‌.2‌ جدول قابلیت‌های مختلف تشخیص موضوع متن 17](#_Toc428959146)

[3‌.2‌ پایگاه داده آموزش و آزمون 18](#_Toc428959147)

[3‌.3‌ پیش پردازش 20](#_Toc428959148)

[3‌.3‌.1‌ حذف کلمات متداول 20](#_Toc428959149)

[2.3.3 فرآیند ریشه یابی 21](#_Toc428959150)

[3‌.4‌ متن‌های جدید منتشر شده بر روی وب‌سایت‌ها 24](#_Toc428959151)

[3‌.5‌ معرفی روند اصلی کار سیستم تشخیص موضوع متن 25](#_Toc428959152)

[3‌.5‌.1‌ یادگیری با نظارت، روش DOR 25](#_Toc428959153)

[3‌.5‌.2‌ یادگیری بدون نظارت، روش تخصیص پنهان دیریکله 26](#_Toc428959154)

[6.3‌ قابلیت توسعه پذیری سیستم 28](#_Toc428959155)

[4 فصل چهارم پیاده سازی و ایجاد رابط کاربری 30](#_Toc428959156)

[4‌.1‌ تحلیل و طراحی نرم‌افزار 30](#_Toc428959157)

[4‌.2‌ ابزارهای مورد استفاده 37](#_Toc428959159)

[4‌.3‌ ایجاد رابط کاربری گرافیکی 38](#_Toc428959160)

[5 فصل پنجم اندازه گیری و آزمایش و جمع بندی 43](#_Toc428959161)

[5‌.1‌ معیار اندازه گیری 43](#_Toc428959162)

[1.1.5 دقت 43](#_Toc428959163)

[5‌.2‌ نتایج 44](#_Toc428959164)

[5‌.3‌ جمع بندی و کار‌های آینده 45](#_Toc428959165)

[منابع و مراجع 46](#_Toc428959166)

|  |  |
| --- | --- |
| فهرست اشكال | صفحه |

[شکل ‏1‑1: نمای کلی سامانه 3](#_Toc428781304)

[شکل ‏2‑1: نمودار کلی روند کار بخش آموزش و آزمون سیستم‌های یادگیری با نظارت 7](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781305)

[شکل ‏2‑2: الگوریتم ماشین بردار پشتیبان 10](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781306)

[شکل ‏2‑3: روند کلی کار سیستم‌های یادگیری بدون نظارت 11](#_Toc428781307)

[شکل ‏2‑4: مدل تخصیص پنهان دیریکله 13](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781308)

[شکل ‏3‑1:جدول اطلاعات کاربران 16](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781309)

[شکل ‏3‑2: جدول قابلیت‌های مختلف موجود در سامانه 17](#_Toc428781310)

[شکل ‏3‑3: نمونه ‌ای از داده‌های آموزش برای دو موضوع خوشنویسی و سینما 20](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781311)

[شکل ‏3‑4: کاراکترهای زایدی که در زمان پیش پردازش از متن ورودی حذف می‌شوند. 24](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781312)

[شکل ‏4‑1: فرآیند تولید نرم‌افزار به صورت آبشاری 31](#_Toc428781313)

[شکل ‏4‑3: نموار Usecase سیستم 32](#_Toc428781315)

[شکل ‏4‑4: نمودار Sequence ورود به سیستم 32](#_Toc428781316)

[شکل ‏4‑5: نمودار Sequence ثبت نام 33](#_Toc428781317)

[شکل ‏4‑6: نمودار Sequence، خروج از سیستم، انتخاب یک موضوع و جست و جو میان موضوعات 33](#_Toc428781318)

[شکل ‏4‑7: نمودار Sequence تعیین حداکثر بازه‌ی زمانی برای ذخیره شدن اطلاعات 34](#_Toc428781319)

[شکل ‏4‑8: نمودار Sequence برای مدیریت کاربری و مدیریت پایگاه داده 34](#_Toc428781320)

[شکل ‏4‑9: نمودار Sequence برای افزودن عملیات جدید 35](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781321)

[شکل ‏4‑10: نمودار Activity مدیریت کاربری، مدیریت پایگاه داده و ورود به سیستم 35](file:///F:\Uni\Term%208\payan%20nameh\HamidReza%20Ramezani.docx#_Toc428781322)

|  |  |
| --- | --- |
| فهرست جداول | صفحه |

[جدول ‏3‑1: اطلاعات موجود در پایگاه داده فارسی برای سیستم یادگیری با نظارت 19](#_Toc428781346)

[جدول ‏3‑2: قواعد ریشه یابی 22](#_Toc428781347)

[جدول ‏3‑3: پارامترهای LibSVM 26](#_Toc428781348)

[جدول ‏3‑4: پارامترهای سیستم تخصیص پنهان دیریکله 27](#_Toc428781349)

[جدول ‏3‑5: پایگاه داده اطلاعات فارسی برای آموزش در روش تخصیص پنهان دیریکله 27](#_Toc428781350)

[جدول ‏5‑1: نتایج خروجی برای کم و زیاد کردن تعداد متن‌‌های آموزش سیستم 44](#_Toc428781351)

[جدول ‏5‑2: نتایج خروجی برای طول متن‌های متفاوت ورودی 44](#_Toc428781352)

[جدول ‏5‑3: نتایج خروجی برای طول متن‌های متفاوت ورودی 45](#_Toc428781353)

فهرست علائم اختصاری

|  |  |  |
| --- | --- | --- |
| -- | Document occurrence Representation | DOR |
| -- | Term co-occurrence representation | TCOR |
| -- | Bag of words | BOW |
| ماشین بردار پشتیبان | Support vector machine | SVM |
| تنک | Sparse | -- |
| تنکی | Sparseness | -- |
| k نزدیک‌ترین همسایه | K-nearest neighbors | KNN |
| نایو بیز | naïve Bayes | -- |
| -- | term frequency–inverse document frequency | tf-idf |
| تخصیص دیریکله نهفته | latent Dirichlet allocation | LDA |
|  |  |  |

# ‌ فصل اول مقدمه

**مقدمه**

با گسترش روزافزون سیستم‌های کامپیوتری، امنیت ذخیره‌سازی و انتقال اطلاعات بیش از پیش اهمیت پیدا کرده است. امنیت اطلاعات در جنبه‌های گوناگونی نظیر امنیت شبکه‌های کامپیوتری، امنیت پایگاه داده، امنیت برنامه‌های کاربردی و غیره مورد توجه پژوهشگران این رشته است. در گذشته، مسائل امنیتی بیشتر مورد توجه مراکز نظامی و سیاسی بوده است، اما اکنون برای مردم و کاربران عادی سیستم‌ها نیز حائز اهمیت است.

یکی از زمینه‌های مطرح در امنیت اطلاعات و ارتباطات، امنیتِ برنامه‌های کاربردی و به پیروی آن، امنیتِ زبان‌های برنامه‌نویسی یا امنیتِ زبان‌مبنا[[1]](#footnote-1) می‌باشد. امنیت زبان‌مبنا را می‌توان مجموعه‌ای از تکنیک‌های مبتنی بر نظریه زبان‌های برنامه‌سازی و پیاده‌سازی آن‌ها، شامل معناشناخت[[2]](#footnote-2)، نوع‌ها[[3]](#footnote-3)، بهینه‌سازی و راستی‌آزمایی[[4]](#footnote-4)، برای به کار گیری در مسائل امنیتی تعریف کرد. [1] تلاش این حوزه بر این است که برنامه‌های کاربردی تولید شده توسط برنامه‌نویسان و توسعه‌دهندگان، با توجه به رویکردهای مختلف امنیتی، قابل اعتماد و اطمینان باشند. به همین دلیل، طراحی و توسعه زبان‌های برنامه‌نویسی امن یا ایجاد ابزارهایی بر روی زبان‌های برنامه‌نویسی موجود باعث می‌شود تا توسعه‌دهندگان نرم‌افزار، کمتر درگیر مشکلات امنیتی برنامه‌های خود شده و به کمک این ابزارها، با تلاش کمتری به تولید برنامه‌های امن بپردازند، که این خود هزینه‌های تولید و توسعه نرم‌افزارها را کاهش می‌دهد.

روش‌های مختلفی برای تولید ابزارهای مرتبط با زبان‌های برنامه‌نویسی با رویکرد برآورده کردن نیازها و خط مشی‌های امنیتی وجود دارد که به طور کلی می‌توان به دو دسته روش‌های تحلیل ایستا[[5]](#footnote-5) یا زمان‌کامپایل[[6]](#footnote-6) و تحلیل پویا[[7]](#footnote-7) یا زمان‌اجرا[[8]](#footnote-8) دسته‌بندی کرد. هر کدام از این روش‌ها نقاط قوت و ضعف مربوط به خود را دارند که بسته به کاربرد، استفاده از هر یک از آن‌ها متفاوت خواهد بود. گرچه شایان ذکر است که تعریف و مشخص کردن دقیق مفهوم امن بودنِ یک سیستم یا برنامه یکی از چالش‌های پیشِ روی متخصصان این حوزه می‌باشد. چنان‌که نحوه و رویکرد اِعمال آن نیازمندی امنیتی، وابستگی زیادی به تعریف ارائه شده خواهد داشت.

هدف از این پروژه، تولید ابزاری برای تشخیص برقراری خط مشی عدم تداخل در کد مبدأ ورودی است که در صورت نقض این خط مشی، با بهره‌گیری از روش بازنویسی برنامه، کد مبدأ به نحوی اصلاح ‌شود تا این نیازمندی برآورده گردد. در این‌جا، خط مشی امنیتیِ عدم تداخل[[9]](#footnote-9) به عنوان نیازمندیِ امنیتی در نظر گرفته می‌شود و برای اِعمال این خط مشی در برنامه‌ها، از یکی از روش‌های تحلیل ایستا؛ یعنی، روش بازنویسی برنامه[[10]](#footnote-10)، استفاده می‌شود که در فصل‌های بعدی، به شرح و توضیح آن‌ها می‌پردازیم.

فصل دوم این پایان‌نامه به توضیح خط مشی امنیتی عدم تداخل و تعریف آن پرداخته خواهد شد و در ادامه، مکانیزم‌های اعمال آن و به ویژه، روش بازنویسی برنامه شرح داده خواهد شد. فصل سوم به توصیف زبان مدل مطرح شده تخصیص یافته است. در فصل چهارم، درباره گراف وابستگی برنامه[[11]](#footnote-11) و کاربرد آن در پروژه بحث خواهد شد. فصل پنجم به توضیح الگوریتم مورد نظر برای بازنویسی کد مبدأ[[12]](#footnote-12) در دو حالت خط مشی عدم تداخل و فصل ششم به فرآیند پیاده‌سازی و تولید ابزار می‌پردازیم. در نهایت، فصل هفتم دربرگیرنده جمع‌بندی و کارهای پیشنهادی آینده پروژه خواهد بود.

# فصل دوم خط مشیِ امنیتیِ عدم تداخل و اِعمال آن

**خط مشی امنیتی عدم تداخل و اعمال آن**

به طور کلی، خط‌ مشیِ[[13]](#footnote-13) امنیتی، امن‌بودنِ یک سیستم یا برنامه را تعریف می‌کند. خط مشی امنیتی، قیود روی توابع و جریان‌‌های بین آن‌ها را مشخص می‌کند؛ مثل قیود دسترسی‌ بر روی برنامه‌ها و سطوح دسترسیِ داده‌های بین کاربران که مانع از بروز مشکلات امنیتی از طریق سیستم‌های خارجی و نفوذگران شود.

از دیدگاهی دیگر، یک خط مشی امنیتی را می‌توان به‌ عنوان یک زیرمجموعه از مجموعه‌ توانیِ همه اجراها تعریف کرد که هر اجرا یک دنباله دلخواه از حالت‌[[14]](#footnote-14)ها است. ضمناً می‌‌توان آن را به ‌عنوان مجموعه‌ برنامه‌هایی در نظر گرفت که آن خط مشی را برآورده می‌کنند. بعضی از خط مشی‌های امنیتی، *خاصیت*[[15]](#footnote-15) هستند؛ به‌خاطر این‌که قابل دسته‌بندی و تشخیص توسط مجموعه اجراهای جداگانه می‌باشند. از این نوع خط مشی‌ها می‌توان به خط‌ مشی‌های کنترل دسترسی اشاره کرد. [2] برخی از نیازمندی‌های مهم امنیتی، خاصیت نیستند. یک نمونه مهم از این‌گونه نیازمندی‌ها، خط مشیِ امنیتی عدم تداخل[[16]](#footnote-16) است. عدم تداخل گوگن-مسگر [3]، عدم تداخل تعمیم یافته [4] و قطعیت مبتنی بر مشاهده [5] از مثال‌های خط مشی‌هایی هستند که نمی‌‌توان آن‌ها را در قالب خاصیت بیان کرد. نکته حائز اهمیت این است که روش اِعمال خاصیت‌ها با نحوه اعمال خط مشی‌هایی که خاصیت نیستند، متفاوت است.

به زبان ساده‌تر‌، خط مشی عدم تداخل بیان می‌کند که یک مشاهده‌گرِ[[17]](#footnote-17) سطحِ پایین که فقط به برنامه و مقادیر عمومیِ زمانِ اجرا دسترسی دارد، نتواند ورودی‌های سطح بالا یا خصوصیِ برنامه را بفهمد. به عبارت دیگر، این خط مشی بیان می‌کند که در هر جفت اجراهای برنامه‌ که ورودی‌های عمومی یکسان دارند، مستقل از ورودی‌های خصوصی متفاوت، باید خروجی‌های عمومی یکی باشند. به طور کلی، طبق این خط مشی، تغییرات ورودی‌های سطح بالا، نباید برای مشاهده‌گر سطح پایین قابل تشخیص و درک باشد.

نکته مهم این است که خط مشی عدم تداخل، یک خاصیت نیست؛ زیرا توسط اجراهای جداگانه که این خط مشی را برآورده می‌کند، قابل تعریف نیست. [2] این نکته باعث ایجاد محدودیت‌هایی برای اِعمال این خط مشی در برنامه‌ها می‌شود.

خط مشی عدم تداخل را می‌توان به دو دسته *حساس به پیشرفت[[18]](#footnote-18)* و *غیر حساس به پیشرفت[[19]](#footnote-19)* تقسیم کرد. در عدم تداخلِ غیر حساس به پیشرفت، مشاهده‌گرِ سطح پایین، تنها می‌تواند خروجی‌های میانیِ سطح پایین را ببیند؛ در حالی‌که یک مشاهده‌گرِ سطح پایین در عدم تداخلِ حساس به پیشرفت، علاوه بر دسترسی‌های قبلی، به *وضعیت پیشرفتِ[[20]](#footnote-20)* برنامه نیز دسترسی دارد. این باعث می‌شود تا بتواند تفاوت بین واگرایی[[21]](#footnote-21) برنامه با موقعیتی که برنامه خاتمه[[22]](#footnote-22) می‌یابد یا در حال محاسبه مقادیر قابل‌مشاهده‌ بعدی است، را تمیز دهد. [2]

با تعاریف بالا، سیستم یا برنامه‌ای که خروجی‌های سطح پایین آن از ورودی‌های سطح بالا تاثیر نگیرد، خط مشی عدم تداخل را برآورده می‌کند. حال باید توجه داشت که جریان اطلاعات[[23]](#footnote-23) از سطح بالا به پایین ممکن است صریح[[24]](#footnote-24) یا ضمنی[[25]](#footnote-25) باشد. انتساب[[26]](#footnote-26) یک مقدار سطح بالا به یک متغیر سطح پایین، نمونه‌ای از جریان اطلاعات صریح است. همچنین، جریان از بالا به پایین در زمانی‌که مقدار یک متغیر سطح پایین مشروط به یک مقدار سطح بالا باشد یا صرفاً زمان‌بندی و رفتار خاتمه برنامه، می‌تواند نمونه‌ای از جریان اطلاعات ضمنی باشد.

در این سیستم، توانایی‌های کاربران سطح پایین بیانگر مدل نفوذگر[[27]](#footnote-27) خواهد بود. با توجه به این توانایی‌ها، روش‌های زیرکانه و مختلفی برای جریان اطلاعات از بالا به پایین وجود خواهد داشت. یک مکانیزم اعمال خط مشی، باید همه انواع مختلف جریان‌های غیرمجاز ناشی از مدل نفوذگر را در نظر بگیرد.

وِنکَتَکریشنَن و همکارانش [6] یک روش تبدیل برنامه‌ی ترکیبی برای اِعمال عدم تداخل ارائه داده‌اند. برنامه تغییر داده‌ شده، سطوح امنیتی انتساب[[28]](#footnote-28) را دنبال می‌کند و زمانی‌که یک جریان غیرمجاز در حال وقوع باشد، خاتمه می‌یابد. این روش، تنها در فرمول‌بندی‌هایی قابل استفاده است که از عدم تداخلِ بدون توجه به رفتار خاتمه‌ی برنامه‌ها مطرح می‌شود.

مَگَزینیوس و همکارانش [7] یک چارچوب برای ناظر‌[[29]](#footnote-29)های امنیتی پویای درون‌برنامه‌ای- در حالی‌که برنامه در حال اجراست- ساخته‌اند. این روش، عدم تداخلِ غیر حساس به خاتمه را تضمین می‌کند و قابل به‌کارگیری در زبان‌های پرل[[30]](#footnote-30) و جاوااسکریپت[[31]](#footnote-31) است که از ارزیابی پویای کد پشتیبانی می‌کنند. ضمناً این روش نیاز دارد که تغییردهنده‌ی برنامه در زمانِ اجرا در دسترس باشد که یک ناظرِ مناسب بتواند در کدی که به صورت پویا تولید می‌شود، ورود کند.

چادنُوف و نومَن [8] یک ناظرِ ترکیبی برای عدم تداخلِ حساس به جریان در زبان‌های با ارزیابی پویای کد پیشنهاد داده‌اند. این روش ممکن است باعث وجود یک سربارِ غیرقابل‌ قبول در زمانِ اجرا شود. همچنین این روش، اجازه وقوع مجراهای خاتمه[[32]](#footnote-32) را نمی‌دهد. سانتوس و رِزک [9] نیز این روش را برای یک هسته JavaScript گسترش دادند.

این موضوع اثبات‌ شده است که هیچ روش کاملاً پویایی برای اِعمال عدم تداخل حساس به جریان وجود ندارد. [10] این موضوع باعث می‌شود که پروژه‌هایی که محدودیت‌های نَحوی[[33]](#footnote-33) بر روی کد دارند، از اطلاعات ایستا در ناظری بر اجراهای چندگانه‌ی برنامه‌ها استفاده کنند.

بِلو و بونِلی [11] یک ناظرِ اجرایی[[34]](#footnote-34) پیشنهاد دادند که از یک تحلیل وابستگیِ زمانِ اجرا بهره می‌برد. برای یافتن یک جریان غیرمجاز، همان‌طور که در طرح پیشنهادی آن‌ها و کارهای مشابه دیگر آمده است، ممکن است نیاز به چندین اجرا از برنامه مورد نظر داشته باشد که در بسیاری از کاربردها این امکان وجود ندارد.

لِگوئِرنیک و همکارانش [12]یک ماشین طراحی کرده‌اند که رخداد‌های انتزاعی[[35]](#footnote-35) در زمان اجرا را دریافت می‌کند و اجرا را توسط بعضی از اطلاعات ایستا، ویرایش می‌کند. این روش نیز جالب است اما اجازه وقوع مجراهای خاتمه را می‌دهد.

همان‌طور که در بالا آمده است، برای اعمال خط مشی عدم تداخل روش‌ها و مکانیزم‌های گوناگونی وجود دارد. اما باید توجه داشت که به طور کلی، مسئله تشخیص برنامه‌هایی که عدم تداخل را برآورده می‌کنند، تصمیم‌ناپذیر[[36]](#footnote-36) است. پس در حالت کلی، عدم تداخل توسط روش‌های ایستا قابل اعمال نیست؛ به همین دلیل است که نوع‌سامانه[[37]](#footnote-37)‌های ارائه‌شده برای این مسئله، محافظه‌کارانه[[38]](#footnote-38) هستند و ممکن است بعضی برنامه‌های امن را نیز نپذیرند. از طرفی، این مسئله هم‌بازگشتی شمارش‌پذیر[[39]](#footnote-39) نیز نیست. بنابراین، توسط ناظر‌های اجرایی که نقضِ خط مشی عدم تداخل در یک برنامه‌ی در حال اجرا را بررسی می‌کنند، قابل اعمال نیست. [2]

یکی از روش‌های مورد استفاده برای اعمال خط مشی‌هایی که خاصیت نیستند، روش بازنویسی برنامه می‌باشد. بازنویسی برنامه دربرگیرنده مکانیزم‌هایی است که یک برنامه داده شده را به برنامه‌ای تبدیل می‌کند که ویژگی‌های درخواستی را برآورده می‌کند. این روش ابتدا برای انتقال کد بین پایگاه[[40]](#footnote-40)‌های سخت‌افزاری، دستگاه‌ها و بهینه‌سازی کارایی استفاده می‌شد. [13] این روش اخیراً به عنوان وسیله‌ای برای اعمال خط مشی‌های امنیتی پیشنهاد شده است. [46] از طرفی می‌توان از روش بازنویسی برنامه برای اعمال خاصیت‌های امنیتی گوناگونی استفاده کرد. از طرف دیگر، علی‌رغم تلاش‌های بسیاری که در این باره صورت گرفته است، جنبه‌های مهمی از سرشت‌نمایی[[41]](#footnote-41) صوری بازنویسی برنامه کماکان از جمله مباحث باز این حوزه به شمار می‌رود.

در این روش، برخلاف بیشتر مکانیزم‌های امنیتی، یک برنامه مغایر با خط مشی، چه قبل و چه در طول زمان ‌اجرا، رد نمی‌کند؛ بلکه آن‌ها با تغییراتی متناسب با نیاز امنیتی خواسته شده بازنویسی می‌شوند و برنامه‌های ناامن به برنامه‌های امن تبدیل خواهند شد. می‌توان نشان داد که از نظارت اجرایی و تحلیل ایستا قدرت بیشتری دارد. [15] در واقع، می‌توان روش بازنویسی برنامه را روشی بین روش‌های ایستا و روش‌های پویا دانست.

از طرفی، روش بازنویسیِ برنامه تغییراتی را به ذاتِ عدم تداخل وارد نمی‌کند؛ بلکه به جای آن، یک برنامه جدید که عدم تداخل را برآورده می‌کند، با حداقل تغییرات ممکن نسبت به برنامه اصلی تولید می‌کند.‌ به این ترتیب، برتری این روش بر مکانیزم‌های ایستا و نظارتی مشخص خواهد شد.

یک بازنویس برنامه باید با توجه به خط مشی امنیتی مورد نظر، سالم[[42]](#footnote-42) و شفاف[[43]](#footnote-43) باشد. سالم بودن به این معنا که کد تولید شده توسط آن، خط مشی را برآورده کند و شفاف بودن به معنای این‌که معناشناخت و رفتار مناسب برنامه، فارغ از امن بودن یا نبودن آن، حفظ بماند. به بیان ساده‌تر، شفاف بودن یک بازنویس بدین معناست که تا حد ممکن، مجموعه اجراهای ممکنِ برنامه‌ تبدیل‌شده، خواه امن یا ناامن، مشابه برنامه‌ی ورودی باشد. البته تا زمانی‌که سلامت و شفافیت دچار خدشه نشود، بازنویس برنامه می‌‌تواند هر تغییری را به کد داده شده اعمال کند. لازم بذکر است که این تعاریف را می‌توان به شکل ریاضی نیز بیان کرد. [2] بدیهی است که یک بازنویس حتما باید سالم باشد تا نیاز امنیتی مورد نظر را برآورده سازد، اما شفافیت بیشتر آن بازنویس، باعث برتری آن روش بر دیگر بازنویس‌های مشابه خواهد بود.

روش بازنویسی برنامه مورد استفاده در این پروژه، از گرافِ وابستگیِ برنامه [16] استفاده می‌کند که قبلاً اثبات شده است که در تشخیص جریان‌های اطلاعاتی احتمالی، دست‌کم به قدرت نوع‌سامانه‌های امنیتی است. [17] برخلاف مکانیزم‌های کنترل جریان اطلاعات مطرح شده در قبل، بازنویس مورد استفاده در این‌جا، رفتارهای معتبر برنامه‌ها را نگه می‌دارد؛ بدین معنا که آن دسته از اجراهای برنامه که به نقض امنیتی منجر نمی‌شود، تغییری نخواهند کرد. این به دلیل رویکرد تعریف و اعمال شفافیت -چالش‌برانگیزترین نیازمندی برای یک مکانیزم اعمال کارا- است. این روش، جریان‌های غیرمجاز صریح و ضمنیِ کد داده شده را به همان خوبی که در صورت بروز آن‌ها در زمان اجرا بازداشته می‌شود، با اصلاح کد برطرف می‌کند. همچنین، از شرط‌های مسیر[[44]](#footnote-44) به منظور بهبود تشخیص شروط مورد نیاز برای برقراری جریان غیرمجاز استفاده می‌شود. با این کار، رفتارهای معتبر بیشتری از برنامه داده شده حفظ خواهد شد. شفافیت به گونه‌ای تعریف شده است که مجموعه اجراهای ممکن برنامه تبدیل‌شده، به نزدیکی و شباهت مجموعه اجراهای ممکن برنامه ورودی باشد. ضمناً در رابطه با مسئله شفافیت، مفهوم *اِعمال اصلاحی*[[45]](#footnote-45) برای خط مشی عدم تداخلِ مطرح شده، به شکل دقیق و صوری و با بهره‌گیری از رابطه پیش‌ترتیبی[[46]](#footnote-46) بیان و اثبات شده است. یک بازنویس به شکل اصلاحی یک خط مشی را اعمال می‌کند، اگر رفتارهای معتبر و مناسب برنامه ورودی -مستقل از اینکه برنامه امن است یا خیر- نگاه داشته شود. اثبات شده است که روش مورد استفاده در این پروژه سالم است، پس قطعاً برنامه‌های امن را تولید خواهد کرد. [2]

توضیحات بیشتر و نحوه دقیق الگوریتم بازنویسی برنامه مورد استفاده در این پروژه، در فصل پنجم به تفصیل آمده است.

# فصل سوم توصیف زبان برنامه‌نویسی WL

**توصیف زبان برنامه‌نویسی WL**

در مقاله اصلی مورد استفاده در این پروژه [2]، برای تعریف دقیق و صوری خط مشیِ عدم تداخل و روش اعمال بازنویسی برنامه، زبان مدلی به نام While Language یا به اختصار WL ارائه شده است. در این پروژه، این زبان برنامه‌نویسی طراحی و پیاده‌سازی شده است و برنامه‌های به این زبان، از طریق ابزار پیاده‌سازی شده، به زبان برنامه‌نویسی C قابل تبدیل می‌باشد. برای این کار، از ابزارهای jflex [18] و bison [19] استفاده شده که در فصل ششم توضیحات کاملی در این خصوص ارائه می‌گردد.

نکته مهم در خصوص زبان برنامه‌نویسی WL این است که برنامه‌های نوشته‌شده به این زبان، می‌توانند مقادیر را در هر زمان دلخواه در طول زمان‌اجرا به عنوان خروجی نمایش داده شود. این در حالیست که بیشتر کارهای مرتبط با امنیت جریان اطلاعات، محدودیت نمایش خروجی در حالت نهایی را دارند.

program ::= **program** **;** clist

clist ::= c | clist **;** c

exp ::= b | n | x | exp **==** exp | exp **<** exp | exp **<=** exp | exp **>=** exp | exp **>** exp

| exp **+** exp | exp **-** exp | exp **or** exp | exp **and** exp | **!** exp

c ::= **NOP** | x **=** exp | **inL** varlist | **inH** varlist | **outL** x | **outH** x | **outL** **BOT** | **outH** **BOT**

|**if** exp **then** clist **endif** | **if** exp **then** clist **else** clist **endif** | **while** exp **do** clist **done**

varlist ::= x | x **,** varlist

b ::= **true** | **false** | **TRUE** | **FALSE**

n ::= integer\_number

x ::= identifier

شکل 1 - ساختار نحوی زبان برنامه‌‌نویسی WL

در شکل 1، ساختار نحوی تجریدی[[47]](#footnote-47) زبانِ WL آمده است. یک عبارت، یا یک عدد صحیح ثابت یا مقدار منطقی بولی[[48]](#footnote-48)، یا یک متغیر عددی و یا یک عملیات یا ارتباط بین یک یا چند عبارت دیگر خواهد بود. مجموعه دستورات شامل NOP برای دستور *هیچ عملیات[[49]](#footnote-49)*، x = exp برای انتساب، inL varlist، inH varlist، outL x و outH x برای ورودی گرفتن و خروجی دادن در نظر گرفته شده است که L و H به ترتیب به معنای سطح امنیتی پایین[[50]](#footnote-50) یا بالا[[51]](#footnote-51) می‌باشد. clist; c برای ایجاد دنباله دستورات است که در آن‌ها ممکن است دستورات شرطی یا حلقه باشد. دستور outL BOT یا outH BOT، یک دستور خروجی منحصر به فردی است که مقدار ثابت BOT یا ⊥ را به عنوان خروجی می‌دهد که از همه مقادیر ثابت موجود در زبان متمایز است. همان‌طور که در نام‌گذاری این زبان نیز مشهود است، مهم‌ترین ساختار کنترلی زبان WL، ساختار while است که مشابه ساختارهای موجود در زبان‌های برنامه‌نویسی رایج، امکان ایجاد حلقه تکرار را برای برنامه‌نویس فراهم می‌آورد. لازم به یادآوری است که گرامر فوق، با حفظ بخش‌های اصلی زبان، تغییراتی نسبت به ساختار نحوی موجود در مقاله اصلی پروژه [2] یافته است.

در این زبان، این فرض صورت گرفته است که فقط در ابتدای کد مبدأ برنامه، دستورات گرفتن ورودی‌های مورد نظر نوشته خواهد شد؛ گرچه دستورات خروجی در هر نقطه‌ای از برنامه می‌توانند وجود داشته باشند. یک رد[[52]](#footnote-52) دنباله‌ای از حالت‌هاست که انتقال به یک حالت ممکن است همراه با وقوع یک رویداد[[53]](#footnote-53) باشد. به این ترتیب، می‌توان یک اجرای برنامه را توسط دنباله‌ای از رویدادهای تولیدی آن نمایش داد. در این‌جا، رویدادها همان ورودی گرفتن از محیط یا خروجی دادن به آن، برحسب استفاده از دستورات in و out است. از طرفی می‌توان واگرایی آرام[[54]](#footnote-54) را نیز در شرایطی که توسط مشاهده‌گر سطح پایین قابل مشاهده باشد، به عنوان یک رویداد تلقی کرد. پس می‌توان یک رد را دنباله‌ای از رویدادهای تولید شده توسط دنباله‌ای از حالت‌ها دانست.

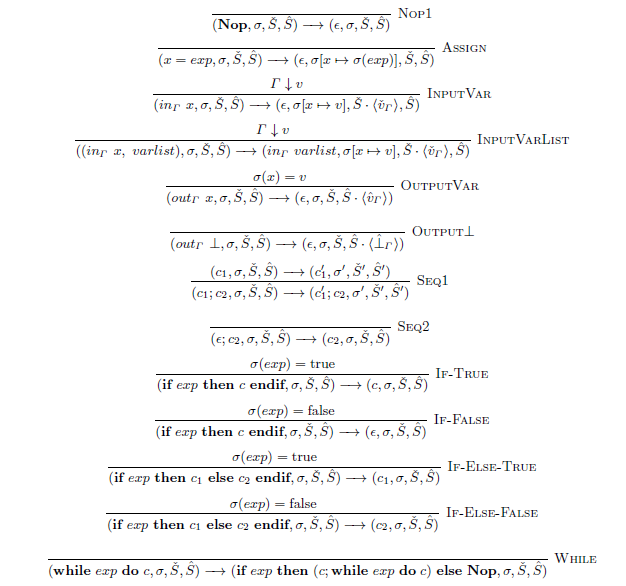
برای بیان معناشناخت این زبان برنامه‌نویسی، از مفهوم پیکربندی[[55]](#footnote-55) استفاده می‌شود و معناشناخت بر اساس آن بیان می‌شود. یک پیکربندی در یکی از دو دسته زیر قرار می‌گیرد:

- پیکربندی پایانی[[56]](#footnote-56) مانند (ϵ, σ, Š, Ŝ) که در آن، ϵ به معنای یک رشته خالی، σ بیانگر حالتِ پیکربندی، Š نشانه دنباله ورودی‌ها و Ŝ برای دنباله خروجی‌ها می‌باشد.

- پیکربندی میانی[[57]](#footnote-57) مانند (c, σ, Š, Ŝ) که در آن، c به معنای یک رشته غیرخالی از نمادهای پایانی است.

مجموعه همه پیکربندی‌ها توسط نماد C نمایش داده می‌شود. به این ترتیب و با توجه به تعاریف بالا، در قاعده‌های معناشناخت رابطه انتقالیِ کوچک‌گامیِ[[58]](#footnote-58) ‘→’ روی C تعریف می‌شود که در آن عبارت‌ها به عبارت‌های ریاضی یا منطق بولی تعبیر می‌شود. حکمِ[[59]](#footnote-59) (c, σ, Š, Ŝ) → (cˊ, σˊ, Šˊ, Ŝˊ) به این معناست که اجرای دستور c در حالت σ و با دنباله ورودی‌ها و خروجی‌های Š و Ŝ، پیکربندی (cˊ, σˊ, Šˊ, Ŝˊ) را نتیجه می‌دهد. البته ممکن است یک انتقال هیچ رویدادی را تولید نکند؛ یعنی Šˊ = Š و Ŝˊ = Ŝ باشد. در صورتی‌که یک انتقال باعث تولید یک رویداد شود، Šˊ = Š . 〈e〉 یا Ŝˊ = Ŝ . 〈e〉 رخ خواهد داد که در آن‌ e بیانگر رویداد ورودی یا خروجی تولید شده توسط انتقال است و ‘.’ نماد عملگر پیوند است.

فرض کنید X مجموعه همه متغیرها باشد. یک حالت σ یک نگاشت به شکل σ : X → N ∪ {null} تعریف می‌شود که در آن، N مجموعه همه اعداد صحیح و ‘null’ مقدار متغیرها قبل از انتساب به ورودی‌های گرفته شده از محیط باشد. σ(x)، مقدار x در حالت σ خواهد بود و σ [x ↦ ν] حالت حاصل از σ توسط به‌روزرسانی مقدار x به ν است. مقدار عبارت exp در σ نیز توسط σ(exp) نمایش داده می‌شود. منظور از ‘Γ ↓ ν’ این است که ورودی بعدی که توسط محیط فراهم خواهد شد ν با سطح امنیتی Γ می‌باشد. در شکل 2، معناشناخت کوچک‌گامیِ زبان WL آمده است که در آن نماد Γ به معنای سطح امنیتی است که در ساختار نحوی ارائه شده در قبل، منظور همان H یا L برای سطح امنیتی بالا یا پایین است. برای اختصار، قواعد مربوط به به‌روزرسانی محیط آورده نشده است.



شکل 2 - معناشناخت کوچک‌گامی برای زبان WL

# فصل چهارم گراف وابستگی برنامه

**گراف وابستگی برنامه**

بازنویس‌های برنامه مورد استفاده در این پروژه، از گراف‌های وابستگی برنامه بهره می‌برند. در این فصل به معرفی گراف وابستگی برنامه، نحوه تولید آن و کاربرد آن در الگوریتم بازنویسی خواهیم پرداخت.

برای هر مکانیزمِ اعمال خط مشی عدم تداخل، به ماشینی برای تشخیص جریان‌های اطلاعات ممکنِ از ورودی‌های سطح بالا به خروجی‌های سطح پایین نیاز است. گراف‌های وابستگی برنامه یا به اختصار PDG[[60]](#footnote-60)ها می‌توانند این امکان را برای ما فراهم کنند. گراف وابستگی برنامه، برنامه را به شکل یک گراف جهت‌دار نمایش می‌دهد که در آن، گره‌ها بیانگر عبارت‌ها یا گزاره[[61]](#footnote-61)‌های برنامه هستند و یال‌ها بیانگر وابستگی‌های کنترلی یا داده‌ای بین گره‌ها می‌باشند. گراف وابستگی برنامه تمامی وابستگی‌های بین گزاره‌های آن برنامه را منعکس می‌کند. این در حالیست که عکس این جمله لزوماً برقرار نیست.

پایه اصلی تولید گراف وابستگی برنامه، گراف جریان کنترل[[62]](#footnote-62) یا به اختصار CFG است. گراف جریان کنترل دنباله اجرای گزاره‌ها را بیان می‌کند. این گراف، یک گراف جهت‌دار است که گره‌ها در آن نمایانگر گزاره‌های برنامه و یال‌ها بیانگر جریان‌های کنترلی بین گره‌ها هستند. در گراف جریان کنترل، دو گره مشخص به نام‌های شروع و پایان در نظر گرفته می‌شود که نقاط ورود و خروج برنامه را تعیین می‌کنند. با به دست آوردن وابستگی‌های کنترلی و داده‌ای، گراف جریان کنترل به گراف وابستگی برنامه تبدیل می‌شود.

در گراف وابستگی برنامه، یک یال وابستگی داده‌ای از گره X به گره Y، که با نمایش داده می‌شود، به این معناست که گره Y دارای متغیری است که در گره X انتساب داده شده است. همچنین، یک یال وابستگی کنترلی از X به Y، که با نمایش داده می‌شود، به معنای این است که اجرای گزاره Y، توسط مقدار محاسبه‌شده در گزاره X کنترل می‌شود. ضمناً می‌توان یک مسیر[[63]](#footnote-63) از X به Y در گراف وابستگی برنامه را به شکل علامت‌گذاری کرد. هر مسیر مشخص‌کننده یک وابستگی کنترلی یا داده‌ای است که بستگی به نوعِ آخرین یال آن مسیر دارد. در ضمن، مسیر ساخته‌شده به واسطه اضافه کردن یال -که به این معناست که نوع وابستگی آن اهمیتی ندارد- به مسیر ، به شکل نمایش داده می‌شود. مسیری مانند در گراف وابستگی برنامه بیانگر این است که ممکن است جریانی از X به Y وجود داشته باشد.

می‌توان چنین تعریف کرد که اگر مقدار محاسبه‌شده در Y یا صرفِ اجرای Y به مقدار محاسبه‌شده در X بستگی داشته باشد، آنگاه گوییم جریانی از X به Y در گراف وابستگی برنامه مربوط به برنامه M وجود دارد. باید توجه داشت که اگر جریانی از X به Y برقرار باشد، آنگاه یک مسیر از X به Y در گراف وابستگی برنامه وجود خواهد داشت؛ در حالی‌که برعکس آن لزوماً صحیح نیست.

همان‌طور که در قبل مطرح شد، می‌توان جریانِ از X به Y بر روی مسیر به دو نوع دسته‌بندی کرد: صریح و ضمنی. یک جریان صریح زمانی برقرار است که مقدار محاسبه‌شده در X، مستقیماً به گره Y منتقل شود. این جریان می‌تواند ناشی از زنجیره انتساب‌های روی آن مسیر باشد. از طرف دیگر، یک جریان ضمنی زمانی برقرار خواهد بود که مقدار محاسبه‌شده در Y، به اجرا شدن یا نشدنِ یک گزاره خاص در مسیر وابسته باشد و اجرای آن گزاره، توسط مقدار محاسبه‌شده در X کنترل شود.

پس با تعاریف فوق می‌توان چنین گفت که مسیر روی گراف وابستگی برنامه تعیین‌کننده یک جریان صریح از X به Y است اگر همه یال‌های موجود در مسیر، از نوع وابستگی داده‌ای باشند. در غیر این صورت، آن مسیر به یک جریان ضمنی دلالت خواهد داشت.

program;

inH h1, h2;

inL l1;

while h1 > l1 do

l1 = l1 + 1;

while h2 > 0 do

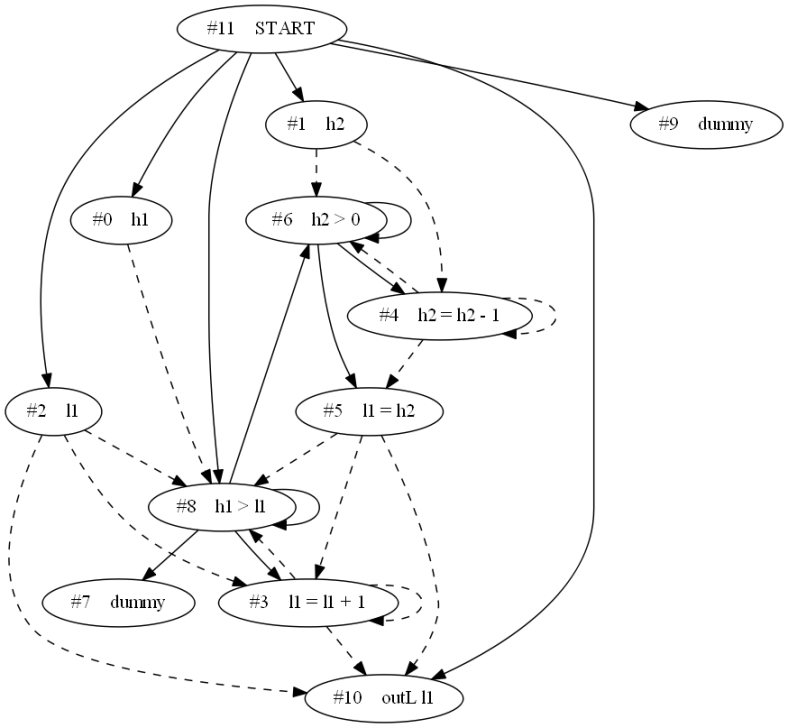
h2 = h2 - 1;

l1 = h2

done

done;

outL l1



شکل 1 - نمونه کد مبدأ به زبان WL و گراف وابستگی برنامه مربوط به آن

همان‌طور که در شکل شماره 3 مشاهده می‌شود، یال‌های با خطوط ساده نماد وابستگی‌های کنترلی و یال‌های خط‌چین نمایانگر وابستگی‌های داده‌ای هستند.

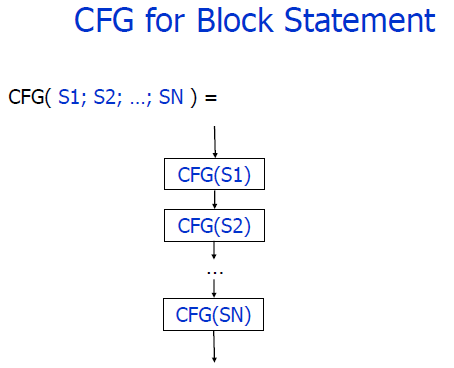
در ادامه این فصل به توضیح نحوه ساختن گراف وابستگی برنامه از روی کد مبدأ برای زبان برنامه‌نویسی WL می‌پردازیم و در فصل بعدی، نحوه استفاده از این گراف در الگوریتم بازنویسی برنامه را به تفصیل شرح خواهیم داد.

برای ساخت گراف وابستگی برنامه، مطابق با شکل شماره 4، گراف‌های مورد نیاز برای تحلیل ساخته می‌شود.



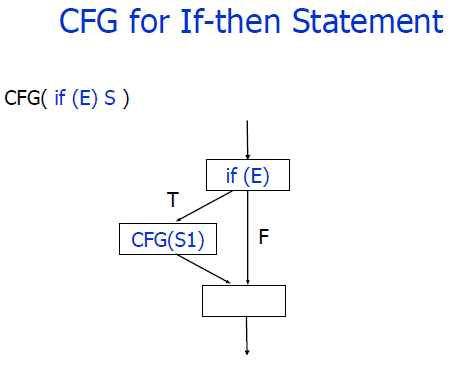
شکل 2 - نمودار کلی نحوه تولید گراف وابستگی برنامه از روی کد مبدأ برنامه [20]

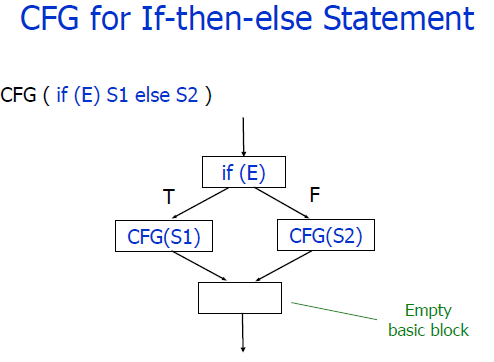
ابتدا از روی کد مبدأ، گراف جریان کنترل یا CFG به دست می‌آید. نحوه تولید این گراف به این نحو است که گزاره‌ها و عبارت‌های برنامه، به عنوان یک گره در گراف در نظر گرفته می‌شود. در این گراف، مفهومی به نام بلوک پایه[[64]](#footnote-64) مطرح می‌شود. هر بلوک پایه، شامل تعدادی گره است که تنها یک گره ورودی و یک گره خروجی در آن وجود دارد. به این منظور، برای گزاره‌‌های ساده که اجرای آن‌ها مشروط نیست، به صورت دنباله پشت سرِ همی از گره‌ها در نظر گرفته می‌شود. پس برای این‌گونه گزاره‌ها و عبارت‌ها، تنها ساختن یک گره جدید و متصل کردن آن‌ها به گراف کفایت می‌کند. شکل شماره 5، نحوه تولید زیرگراف بلوک‌های پایه را نمایش می‌دهد. با همین روش، بلوک‌های پایه با یکدیگر ادغام می‌شوند و در نهایت، گراف جریان کنترل نهایی تولید خواهد شد.



شکل 3 - نحوه تولید زیرگراف بلوک پایه و اتصال به یکدیگر [20]

برای گزاره‌های شرطی، دو حالت ممکن زیر وجود دارد:

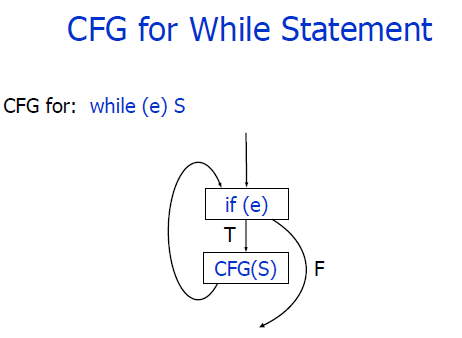




شکل 4 - نحوه تولید زیرگراف گزاره­های شرطی - حالت دوم [20]

شکل 5 - نحوه تولید زیرگراف گزاره­های شرطی – حالت اول [20]

مطابق با شکل‌های شماره 6 و 7، یک گره برای عبارت شرطی تولید می‌شود. گزاره‌های مربوط به برقراری عبارت شرطی در یک بلوک پایه و گزاره‌های متعلق به حالت عدم برقراری شرط، در بلوک پایه‌ دیگری در نظر گرفته می‌شود و این دو بلوک پایه، به گره مربوط به عبارت شرطی متصل خواهند شد. پس از این کار، برای گره پایانی این بلوک پایه، یک گره مجازی[[65]](#footnote-65) ایجاد می‌شود. وجود این گره برای این است که این زیرگراف تنها یک مجرای خروجی داشته باشد.

 ساختار دیگری که در این زبان وجود دارد، ساختار حلقه یا همان while است. برای تولید زیرگراف جریان کنترل مربوط به این عنصر زبان، مطابق با شکل شماره 8 عمل می‌شود.

شکل 6 - نحوه تولید زیرگراف گزاره­های حلقه while [20]

مشابه قبل، یک گره مجازی به عنوان گره پایانی به زیرگراف اضافه می‌شود.

حال به ازای هر قاعده موجود در زبان، مطابق با توضیحات بالا، زیرگراف‌های کنترل جریان در هنگام تولید درخت تجزیه[[66]](#footnote-66) ساخته می‌شوند و با اتصال آن‌ها به یکدیگر، گراف کنترل جریان برنامه به دست خواهد آمد.

مطابق با شکل شماره 4، برای تولید گراف وابستگی کنترل یا به اختصار CDG[[67]](#footnote-67)، به درخت غلبه رو به جلو[[68]](#footnote-68) یا درخت پس‌غلبه[[69]](#footnote-69) نیاز خواهد بود. برای تولید این درخت، الگوریتم ساخت درخت غلبه[[70]](#footnote-70) بر روی معکوسِ گراف جریان کنترل؛ یعنی همان گره‌ها ولی با جهت یال‌های معکوس شده، اعمال می‌گردد.

برای این کار، ابتدا منظور از غلبه کردن دو گره را بیان می‌کنیم. گره M بر گره N غلبه می‌کند، اگر و تنها اگر همه مسیرهای با شروع از از گره آغازین تا گره N، حتماً و الزاماً از گره M بگذرند. همچنین، گره M بر گره N اکیداً غلبه[[71]](#footnote-71) می‌کند، اگر و تنها اگر بر آن گره غلبه کند و M، همان گره N نباشد. واضح است که یک گره در گراف جریان کنترل می‌تواند چندین غلبه‌کننده[[72]](#footnote-72) داشته باشد، اما برای تولید درخت غلبه، نزدیک‌ترین غلبه‌کننده یا غلبه‌کننده بی‌درنگ[[73]](#footnote-73) اهمیت دارد. با استفاده از شبه‌کد زیر می‌توان غلبه‌کنند‌ه‌های یک گره در گراف جریان کنترل را به دست آورد:

Compute Dominators(){

For (each n ∈ NodeSet)

Dom(n) = NodeSet

WorkList = {StartNode}

While (WorkList ɸ ≠) {

Remove any node Y from WorkList

New = {Y} ∪ ∩x∈Pred(Y) Dom(X)

If New ≠ Dom(Y) {

Dom(Y) = New

For (each Z ∈ Succ(Y))

WorkList = WorkList U {Z}

}

}

}

شکل 7 - محاسبه گره‌های غلبه‌کننده برای هر گره [21]

سپس با توجه به مجموعه غلبه‌کنندگانِ به دست آمده از الگوریتم بالا و مقایسه با مجموعه غلبه‌کنندگانِ سایر گره‌ها، می‌توان گره غلبه‌کننده بی‌درنگ را یافت و درخت غلبه را تشکیل داد اما در این‌جا، تولید درخت پس‌غلبه[[74]](#footnote-74) مورد نظر است. به این صورت که، گره Z، گره Y را پس‌غلبه می‌کند، اگر و تنها اگر همه مسیرهای از Y تا گره پایانی، حتماً و الزاماً از Z عبور کنند. حال، در صورتی که این الگوریتم برای معکوس گراف جریان کنترل اعمال شود، درخت پس‌غلبه تولید می‌شود.

سپس برای ساخت گراف جریان کنترل، مرزِهای پس‌غلبه[[75]](#footnote-75) یا اختصاراً PDF، مورد نیاز است. مرز پس‌غلبه گره X، مجموعه‌ گره‌هایی هستند که توسط X اکیداً پس‌غلبه نمی‌شوند اما گره‌های مابَعدی[[76]](#footnote-76) دارند که توسط X پس‌غلبه می‌شوند. تعریف ریاضی این گره‌ها بدین شرح است:

PDF(X) = {y | (∃z ∈ Succ(y) such that x post-dominates z) and x does not strictly post-dominate y}

که این مجموعه بیانگر نزدیک‌ترین نقاط انشعابی[[77]](#footnote-77) است که به گره X منجر می‌شوند.

با استفاده از قضیه زیر، می‌توان وابستگی‌های کنترلی برنامه را برای هر گره موجود در گراف جریان کنترل، به دست آورد:

**قضیه** – گره y به مجموعه PDF(X) تعلق دارد، اگر و تنها اگر X به Y وابستگی کنترلی داشته باشد.

حال با استفاده از الگوریتم زیر، می‌توان مجموعه مرزهای پس‌غلبه هر گره را به دست آورد که بیانگر وابستگی‌های کنترلی نیز خواهند بود.

شکل 8- محاسبه گره‌های پس‌غلبه‌کننده مرزی برای هر گره

For each x in the bottom-up traversal of the postdominator tree do

PDF(X) = ɸ

Step 1: For each y in Predecessor(X) do

If X is not immediate post-dominator of y then

PDF(X) ← PDF(X) ∪ {y}

Step 2: For each z that x immediately post-dominates, do

For each y ∈ PDF(Z) do

If X is not immediate post-dominator of y then

PDF(X) ← PDF(X) ∪ {y}

پس از این محاسبات، وابستگی‌های کنترلی برنامه برای هر گره – گزاره یا عبارت برنامه ورودی– به دست آمده است. در نتیجه، تا این مرحله، گراف وابستگی کنترلی برنامه تولید شده است.

اکنون نوبت تولید گراف وابستگی داده‌ای یا به اختصار [[78]](#footnote-78)DDG است. وابستگی‌های داده‌ای مختلفی وجود دارد اما برای این پروژه، ارتباط بین گره‌هایی که شامل مقداردهی یک متغیر و استفاده از آن متغیر هستند، اهمیت دارد؛ یعنی گره X به گره Y وابستگی داده‌ای دارد، اگر و تنها اگر در گره Y متغیری وجود داشته باشد که در گره X مقداردهی شده باشد. پس با توجه به همین تعریف و مطابق با قواعد زبان، در گره مربوط به هر گزاره، متغیری که به آن مقداری نسبت داده شده یا استفاده شده است، نگه‌داری می‌شود. حال برای به دست آوردن وابستگی‌های داده‌ای، در صورتی که در یک گره از متغیری استفاده شود که در گره دیگری مقداردهی شده است، یک وابستگی داده‌ای لحاظ می‌شود. برای افزایش دقت و عدم محافظه‌کارانه بودن وابستگی‌ها، تنها نزدیک‌ترین گزاره‌ای که آن متغیر در آن مقداردهی شده است، وابستگی را خواهد داشت، و نه همه گزاره‌هایی که آن متغیر را مقداردهی کردند که این کار با پیمایش گراف جریان کنترل امکان‌پذیر است.

پس از این مرحله، گراف وابستگی داده‌ای برنامه نیز آماده است. بدین ترتیب گراف‌های وابستگی کنترلی و داده‌ای از روی کد مبدأ ساخته شده‌اند. با ترکیب این دو گراف که دارای گره‌های یکسان هستند، گراف وابستگی برنامه تولید خواهد شد.

در فصل بعدی، نحوه بازنویسی برنامه‌ها با استفاده از گراف وابستگی برنامه تولید شده تا این مرحله بیان می‌شود.

# فصل پنجم الگوریتم بازنویسی برنامه

# الگوریتم بازنویسی برنامه

در این فصل ابتدا الگوریتم کلی مورد استفاده برای اعمال خط مشی عدم تداخل بیان می‌شود. سپس به الگوریتم دقیق مربوط به حالت غیرحساس به پیشرفت یا به اختصار PINI[[79]](#footnote-79) و حالت حساس به پیشرفت یا PSNI[[80]](#footnote-80) خواهیم پرداخت.

با داشتن گراف وابستگی برنامه کد مبدأ ورودی، می‌توان وابستگی‌های موجود بین گزاره‌های مختلف برنامه را بررسی کرد. می‌توان تابع متأثر کردن[[81]](#footnote-81) یا affect را تابعی در نظر گرفت که یک عبارت یا گزاره از یک برنامه یا گره مربوط به آن در گراف وابستگی برنامه را به عنوان ورودی می‌گیرد وگزاره‌ها و عباراتی که به گزاره یا عبارت گرفته‌شده وابستگی دارند را به عنوان خروجی برمی‌گرداند. با بیانی دیگر، تابع affect بر روی گره داده‌شده X از گراف وابستگی برنامه اجرا می‌شود و همه گره‌های مانند Y را که یک مسیر از X به آن وجود دارد را برمی‌گرداند. الگوریتم کلی بازنویسی برای خط مشی عدم تداخل را می‌توان براساس تابع پیشنهادشده ارائه کرد. تعبیر رویداد قابل‌مشاهده برای کاربر سطح پایین در این پروژه، مطابق با بیان صوری مطرح شده در مقاله اصلی مورد استفاده در پروژه است. [2]

**foreach** statement X producing a high input event h­in **do**

**foreach** statement Y producing a low observable event eL **do**

**if**  **then**

transform Y into Yˊ such that in the new program

end

end

end

شکل 9 - الگوریتم کلی بازنویسی برای اعمال خط مشی عدم تداخل [2]

همان‌طور که بیان شد، یک گراف وابستگی برنامه که یک بیان ایستایی از وابستگی‌های برنامه را نشان می‌دهد، جریان‌های غیرمجاز ممکن را در خود دارد. گرچه ممکن است این جریان‌ها در همه اجراهای برنامه رخ ندهند. بنابراین در الگوریتم شکل شماره 9 باید شرط‌هایی که تعیین‌کننده وقوع جریان غیرمجاز احتمالی هستند را لحاظ کرد.

## بازنویسی برای حالت غیرحساس به پیشرفت

ایده اصلی استفاده شده در این بخش، دستورهای outLای که از ورودی‌های سطح بالا متأثر شد‌ه‌اند، با گزاره‌های outL ⊥ یا NOP جایگزین شوند. چنین تغییراتی که مستقل از اطلاعات زمان‌اجرا هستند، ممکن است بیش از حد مورد نیاز و کمی سخت‌گیرانه باشد. به خاطر دسترسی برنامه‌ها به اطلاعات زمان‌اجرا، می‌توان از این اطلاعات در برنامه بازنویسی‌شده استفاده کرد. به همین منظور، از گونه‌ای از شرط‌های مسیر[[82]](#footnote-82) استفاده شده است. در مقاله اصلی پروژه، یک شرط مسیر p(X,Y) روی متغیرهای برنامه تعریف می‌شود و همان شرط‌هایی هستند که باعث می‌شوند تا جریان واقعاً رخ بدهد. به این معنا که شرط‌های مسیر باید برقرار باشند تا جریان مربوط به آن مسیر در اجرا نیز اتفاق بیفتد. این تعریف از شرط‌های مسیر می‌تواند نشان دهد که آیا یک مسیر واقعاً در زمان‌اجرا پیمایش می‌شود یا خیر. همین نکته برای تشخیص جریان‌های صریح بسیار مفید خواهد بود. در حالی‌که برای جریان‌های ضمنی، ممکن است جریان در زمان‌اجرا به وقوع بپیوندد، حتی اگر مسیر مربوط به آن به طور کامل پیمایش نشده باشد. این مورد زمانی اتفاق می‌افتد که یک گره روی مسیر با یال وارد شونده[[83]](#footnote-83) وابستگی کنترلی، به خاطر مقدار عبارت کنترلی اجرا نشود. پس اجرای همه نود‌های روی مسیرِ تعیین‌کننده یک جریان ضمنی، برای وقوع آن جریان الزامی نیست. به همین ترتیب، جریان مربوط به مسیر روی گراف وابستگی برنامه یک برنامه به زبان WL فقط زمانی رخ می‌دهد که همه گره‌های آن مسیر که یال وارد شونده وابستگی داده‌ای دارند، اجرا شوند. می‌توان چنین گفت که همه گره‌های میانی روی مسیرِ بیان‌کننده یک جریان صریح، باید در زمان‌اجرا پیمایش شوند. همچنین، یک گره میانی روی مسیر مشخص‌کننده یک جریان ضمنی، فقط باید زمانی پیمایش شود که یال واردشونده به آن از نوع وابستگی داده‌ای باشد. همان‌طور که در ادامه مطرح خواهد شد، بازنویس‌های استفاده شده در این پروژه، تغییراتی را در برنامه داده شده انجام می‌دهند چنان‌که بررسی می‌شود تا همه گره‌های میانی با یال وارد شونده وابستگی داده‌ای بر روی مسیری که به دستورات outL l ختم می‌شوند، در طول برنامه اجرا ‌شوند. اگر مسیر چنین باشد، outL ⊥ یا NOP به جای آن دستور اجرا خواهد شد، وگرنه خود دستور outL l به اجرا در می‌آید.

برنامه‌های نوشته‌شده به زبان WL حاوی شرط‌های مسیر ساده هستند که از شرط‌های اجرای[[84]](#footnote-84) گره‌ها به دست می‌آیند. به طور کلی، شرط اجرا برای گره X با پیمایش معکوس[[85]](#footnote-85) از گره X تا گره آغازین از طریق یال‌های وابستگی‌های کنترلی روی مسیر حاصل می‌شود. شرط اجرا یک عبارت منطقی بولی است که برقرار خواهد بود، اگر و تنها اگر گزاره X اجرا شود. به همین شیوه، شرط مسیر p(X,Y) برای ، ترکیب عطفی شرط‌های اجرای گره‌های روی آن مسیر تعریف می‌شود. شرط‌های مسیر را می‌توان بر اساس ترکیب عطفی گره‌های روی مسیر با یک یال وارد شونده وابستگی داده‌ای تعریف کرد. اگر چنین گره‌ای نباشد، شرط مسیر همواره درست محسوب می‌شود.

با توجه به قاعده‌های زبان و استفاده مناسب‌تر از شرط‌های مسیر، در برنامه‌ها تنها انتساب‌های یگانه ایستا[[86]](#footnote-86) مجاز است. به معنای آن‌که برنامه‌ها حاوی انتساب‌های چندگانه برای یک متغیر نخواهد بود. البته این برنامه‌ها به برنامه‌های صرفاً حاوی انتساب‌های یگانه قابل تبدیل هستند. از طرفی، چنین فرض می‌شود که هیچ وابستگی داده‌ایِ حلقه نقلی[[87]](#footnote-87) در مسیرهای از ورودی‌های سطح بالا به خروجی‌های سطح پایین برنامه وجود ندارد.

در الگوریتم شکل شماره 10، بازنویس برای اعمال خط مشی عدم تداخل در حالت غیرحساس به پیشرفت، کد مبدأ برنامه M و گراف وابستگی برنامه G مربوط به آن را به عنوان ورودی می‌گیرد و کد مبدأ Mˊ را که عدم تداخل غیرحساس به پیشرفت را برآورده می‌کند، به عنوان خروجی برمی‌گرداند. به بیان دیگر، باید دنباله خروجی‌های برنامه بازنویسی‌شده برای هر دو اجرای دلخواه از آن برنامه که ورودی‌های سطح پایین یکسان دارند، نسبت به حالت غیرحساس به پیشرفت معادل باشند. در ادامه، نمونه‌ای از نحوه عملکرد این الگوریتم را بر روی یک برنامه داده شده به زبان WL در شکل شماره 11 آورده شده است.

***RWPINI (M, G):***

Initialize to the set of all paths in the PDG *G* of *M* where *P* is the node representing a high input and is the node representing outL l for some l;

**if** **then**

**return** M;

**end**

create a copy of M, name it Mˊ, and change it as follows:

determine the type of flow indicated by each path ;

**foreach** **do**:

Generate the path condition of as the conjunction of the execution conditions of node N satisfying if there are such nodes on the path and true otherwise;

**end**

**foreach** node n on G representing outL l for some l **do**

let c be the disjunction of the path conditions of all which terminate at n;

**if** all paths terminating at n indicate an explicit flow **then**

replace outL l with the statement “if c then outL ⊥ else outL l endif”;

**else**

replace outL l with the statement “if c then NOP else outL l endif”;

**end**

**end**

**return** Mˊ;

شکل 10 - الگوریتم بازنویسی عدم تداخل حالت غیرحساس به پیشرفت که برنامه M و گراف وابستگی برنامه مربوط به آنِ G را می‌گیرد [2]

program;

inL l1, l2;

inH h1;

if l1 == 0 then

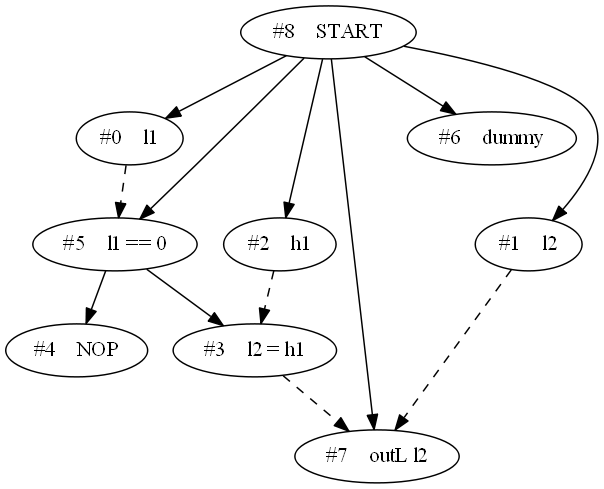
l2 = h1

else

NOP

endif;

outL l2



program;

inL l1, l2;

inH h1;

if l1 == 0 then

l2 = h1

else

NOP

endif ;

if ( (l1 == 0) ) then

outL BOT

else

outL l2

endif

(ب) (الف)

(ج)

شکل 11 – (الف) نمونه کد به زبان WL؛ (ب) گراف وابستگی برنامه الف؛ (ج) برنامه بازنویسی‌شده برنامه الف در حالت غیرحساس به پیشرفت

## بازنویسی برای حالت حساس به پیشرفت

حالت حساس به پیشرفت نسبت به حالت غیرحساس به پیشرفت محدودیت و قیود بیشتری روی رفتار مشاهده‌گر سطح پایین اعمال می‌کند. می‌توان با بررسی مثالی بیشتر به آن پرداخت. در برنامه شکل شماره 12، الگوریتم بازنویسی برای حالت غیرحساس به پیشرفت دستور outL l1 در خط ششم را با دستور NOP جایگزین می‌کند. گرچه نتیجه بازنویسی حالت غیرحساس به پیشرفت را برآورده می‌کند، اما با توجه به حالت حساس به پیشرفت، یک برنامه ناامن خواهد بود؛ چرا که حلقه while موجود در برنامه ممکن است بسته به مقدار سطح بالای h1، واگرا شود یا نشود. به عبارت دیگر، یک مشاهده‌گر سطح پایین می‌تواند با بررسی روند پیشرفت برنامه، مقدار h1 را استنباط کند.

program;

inH h1;

inL l1;

if h1 == l1 then

while true do

outL l1

done

else

NOP

endif;

outL l1

program;

inH h1;

inL l1;

if h1 == l1 then

while true do

if true then

NOP

else

outL l1

endif

done

else

NOP

endif;

outL l1

شکل 12 - (الف) نمونه برنامه به زبان WL؛ (ب) برنامه بازنویسی‌شده الف برای حالت غیرحساس به پیشرفت، که حالت حساس به پیشرفت را برآورده نمی‌کند

(ب) (الف)

بنابراین، برای اعمال حالت حساس به پیشرفت، باید مطمئن شد که نحوه پیشرفت برنامه نیز هیچ اطلاعات سطح بالایی را افشا نمی‌کند. پس برنامه باید با شروع از حالت‌های آغازینِ معادل از نظر مشاهده‌گر سطح پایین، یا همواره خاتمه یابد یا همواره واگرا باشد. گرچه ابزارها و روش‌هایی برای دسته‌های خاصی از برنامه‌ها وجود دارد که بررسی شود که آیا یک برنامه خاتمه می‌یابد یا خیر، اما این مسئله در حالت کلی تصمیم‌ناپذیر است. شاید به همین خاطر است که راه‌حل‌های ارائه شده برای حالت حساس به پیشرفت، که تعداد کمی هم هستند، بسیار محافظه‌کارانه است و هر برنامه‌ای که یک حلقه وابسته به مقدار سطح بالا باشد، پذیرفته نمی‌شود. منظور از حلقه وابسته به مقدار سطح بالا، یعنی حلقه‌ای که اجرای بدنه‌ آن یا تعداد تکرارهای اجرای آن به یک مقدار سطح بالا وابسته باشد. به عنوان نمونه، مور و همکارانش [22] یک نوع‌سامانه به همراه یک مکانیزم زمان‌اجرا به نام پیش‌گویی خاتمه[[88]](#footnote-88) ارائه کرده است تا حلقه‌هایی تشخیص داده شود که وضعیت پیشرفت آن‌ها فقط به مقادیر سطح پایین وابسته است. گرچه چنین مکانیزمی در قیاس با راه‌حل‌های ایستا از دقت بالاتری برخوردار است، اما هزینه سربار اضافه زمان‌اجرا را در پی خواهد داشت. از طرفی، اگر مکانیزم پیش‌گویی نتواند وضعیت پیشرفت حلقه را پیش‌بینی کند، اجرای برنامه گیر خواهد کرد.

بازنویس مورد استفاده در این پروژه، برنامه‌ها را به نحوی تغییر می‌دهد که وضعیت پیشرفت برنامه بازنویسی‌شده به مقادیر سطح بالا وابستگی نداشته باشد. باید توجه داشت که در برنامه‌ای که ممکن است هنگام اجرا مقادیر سطح بالا از طریق وضعیت پیشرفت برنامه نشت پیدا کنند، به دلیل سلامتِ الگوریتم بازنویسی، معناشناخت برنامه دچار تغییراتی شود. در زبان برنامه‌نویسی WL، عنصر while تنها ساختاری است که می‌تواند باعث واگرایی برنامه‌ها شود. پس به ابزار یا تابعی برای تحلیل حلقه‌های while نیاز خواهد بود. در الگوریتم بازنویسی مورد استفاده چنین فرض می‌شود که یک تحلیل‌گر حلقه[[89]](#footnote-89) وجود دارد که می‌تواند به طور ایستا با گرفتن کد حلقه، آن را تحلیل و ارزیابی کند. این الگوریتم این تضمین را می‌دهد که برنامه بازنویسی‌شده برای حالت‌های آغازینِ معادل از نظر مشاهده‌گر سطح پایین، یا همواره خاتمه می‌یابد یا همواره واگرا می‌شود. [2]

تحلیل‌گر حلقه مورد نظر در این الگوریتم چنین در نظر گرفته می‌شود که کد یک حلقه را می‌گیرد و یک عبارت منطقی بولی به عنوان نتیجه تحلیل برمی‌گرداند. این عبارت منطقی برای حالت‌هایی درست خواهد بود که اجرای آن حلقه قطعاً خاتمه می‌یابد. بدین معنا که تابع تحلیل‌گر حلقه عبارت همواره درست یا True را برمی‌گرداند، اگر حلقه همواره خاتمه می‌یابد و عبارت همواره نادرست یا False را برمی‌گرداند، اگر حلقه در همه حالت‌ها واگرا باشد. این در حالیست که این تابع تحلیل‌گر، به عنوان مثال، برای حلقه موجود در برنامه شکل شماره 13، عبارت h1 >= l1 or l1 < 0 را به عنوان نتیجه تحلیل برمی‌گرداند.

program;

inH h1;

inL l1;

while h1 < l1 do

NOP;

h1 = h1 - l1

done;

outL l1

شکل 13 - برنامه‌ای که حلقه موجود در آن در حالتی که h1 >= l1 or l1 < 0 باشد، خاتمه خواهد یافت

الگوریتم بازنویسی مورد استفاده منوط به وجود یک تحلیل‌گر حلقه قدرتمند است. ابزاری که بتواند بسیاری از حلقه‌ها را با موفقیت تحلیل کند. تولید چنین ابزاری کار دشواری است که در فصل آینده، به نحوه پیاده‌سازی آن خواهیم پرداخت. با این حال و با وجود تلاش‌هایی که در این زمینه صورت گرفته است، باز هم ممکن است حلقه‌هایی در برنامه‌ها وجود داشته باشند که تحلیل‌گر مورد استفاده، از تحلیل آن‌ها عاجز باشد. در این‌جا فرض می‌شود که چنین برنامه‌هایی برای ورودی الگوریتم بازنویسی در نظر گرفته نمی‌شود. [2]

پس با توضیحات فوق، بازنویس مورد استفاده در این پروژه، مسیرهای روی گراف وابستگی برنامه با شروع از متغیرهای سطح بالا تا عبارت شرطی حلقه‌ها[[90]](#footnote-90) را می‌پیماید و با استفاده از نتیجه تحلیل‌گر حلقه، به بازنویسی کد مبدأ برنامه می‌پردازد. به این صورت که اگر نتیجه تحلیل‌گر حلقه برای یک حلقه داده شده، عبارت همواره درست باشد، حلقه را بدون تغییر رها می‌کند و مشابه این رفتار را برای یک حلقه همواره واگرا نیز خواهد داشت، به شرطی که هیچ مسیر از نوع وابستگی کنترلی از مقادیر سطح بالا به عبارت شرطی حلقه وجود نداشته باشد. در واقع، حلقه‌ای که همواره واگراست، ممکن است باعث افشای اطلاعات سطح بالا شود، اگر آن حلقه توسط یک عبارت کنترل شود که به ورودی‌های سطح بالا وابسته است. اگر چنین باشد، آن حلقه با یک ساختار if-then با همان عبارت شرطی حلقه و همان بدنه حلقه جایگزین می‌شود. در شرایطی که تحلیل‌گر حلقه عبارتی غیر از همواره درست یا همواره نادرست را برگرداند، بازنویس اجرای آن حلقه را به همان عبارت برگردانده شده مشروط می‌کند. پس به این ترتیب، کد برنامه جدید قطعاً خاتمه می‌یابد.

شکل 14 - الگوریتم بازنویسی عدم تداخل حالت حساس به پیشرفت که برنامه M و گراف وابستگی برنامه مربوط به آنِ G را می‌گیرد [2]

***RWPSNI(M, G)***:

Initialize *D* to the set of all paths in *G* where is a path terminating at a loop guard and *P* is the node representing a high input;

*Mˊ* = *RWPINI*(M, G);

**if**  **then**

**return** Mˊ;

**end**

*H* = max {*height*(n) | n is a node on G}, where *height* is a function that returns the height of a given node on the tree obtained by removing data dependence edges from G;

Change Mˊ as follows:

**for** *h* = *H* to 1 **do**

**foreach** node n with *height(n)* = *h* representing a loop on some path **do**

r = *LoopAnalyzer*(*loop*(n));

**if** r = False **then**

**if** appears on at least one path **do**

replace *loop*(n) with the statement “if *guard*(n) then *body*(n) endif”;

**end**

**else**

**if** r ≠ True **then**

replace *loop*(n) with the statement “if r then *loop*(n) endif”;

**end**

**end**

**end**

h = h – 1;

**end**

**return** Mˊ;

الگوریتم بازنویسی مطرح‌شده در شکل شماره 14، کد مبدأ برنامه M و گراف وابستگی برنامه متناظر آن را می‌گیرد و کد Mˊ بازنویسی‌شده را برمی‌گرداند که خط مشی عدم تداخل را در حالت حساس به پیشرفت برآورده می‌کند. منظور از تابع LoopAnalyzer همان ابزاری است که در بالا توضیح داده شده بود و تحلیل حلقه‌ها را برعهده داشت. این بازنویس، ابتدا بازنویس مربوط به حالت غیرحساس به پیشرفت را فراخوانی می‌کند. نتیجه این کار، برنامه‌ای خواهد بود که اگر M حلقه‌های وابسته به مقادیر سطح بالا نداشته باشد، در حالت حساس به پیشرفت نیز پذیرفته می‌شود. در غیر این صورت، حلقه‌هایی که در مسیرهای به شکل هستند، که در آن h یک ورودی سطح بالا و یک مسیر منتهی به یک عبارت شرطی حلقه است، ممکن است بازنویسی شوند. البته ممکن است این‌گونه مسیرها حاوی گره‌های میانی باشند که خود بیانگر عبارت‌ شرطی حلقه‌های دیگری هستند. توابع *guard*(n)، *body*(n) و *loop*(n) به ترتیب مقدار عبارت شرطی حلقه، بدنه حلقه و کل حلقه موجود در Mˊ متناظر با گره n در گراف وابستگی برنامه را برمی‌گرداند.

همان‌طور که مشاهده می‌شود، بازنویس مطرح شده برای حالت حساس به پیشرفت ممکن است یک حلقه وابسته به مقادیر سطح بالا را به یک گزاره شرطی با همان بدنه حلقه جایگزین کند که این باعث می‌شود تا بدنه حلقه تنها یک بار در برنامه بازنویسی‌شده اجرا شود. گرچه راهبردهای دیگری مثل تغییر عبارت شرطی حلقه برای این‌که فقط به مقدار متناهی حلقه اجرا شود نیز وجود دارد. البته باید آن راهبردها را از نظر شفافیت با روش مورد استفاده در این‌جا بررسی کرد. ضمناً باید در نظر داشت که ابتدا حلقه‌های تودرتو[[91]](#footnote-91) و سپس حلقه بیرونی تحلیل می‌شوند. زیرا تاثیر رفتار نسخه بعد از بازنویسی آن‌ها ممکن است با نسخه قبل از بازنویسی متفاوت باشد. به همین منظور، الگوریتم مطرح‌شده ارتفاع گره‌های بیانگر حلقه در درختی که با حذف یال‌های وابستگی داده‌ای از گراف وابستگی برنامه به دست‌ آمده است را ملاک عمل قرار می‌دهد. در شکل شماره 15، نمونه کد برنامه تبدیل شده توسط این الگوریتم برای کد مبدأ برنامه شکل شماره 13 را مشاهده می‌شود.

برای اثبات سلامت و شفافیت الگوریتم‌های استفاده‌شده در این پروژه، می‌توانید به مقاله اصلی پروژه [2] بخش ششم مراجعه کنید.

program;

inH h1;

inL l1;

if l1 <= h1 or l1 < 0 then

while h1 < l1 do

NOP;

h1 = h1 - l1

done

endif;

outL l1

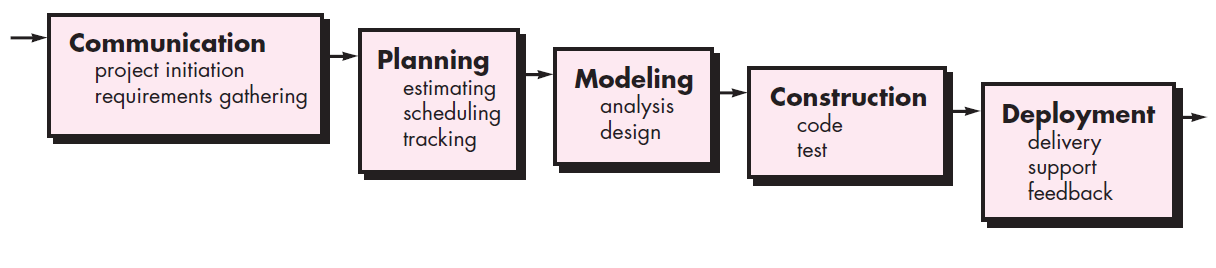
شکل 15 - کد مبدأ بازنویسی‌شده توسط الگوریتم حالت حساس به پیشرفت برای برنامه شکل شماره 13

# فصل ششم پیاده­سازی و ایجاد رابط کاربری

**پیاده‌سازی و ایجاد رابط کاربری**

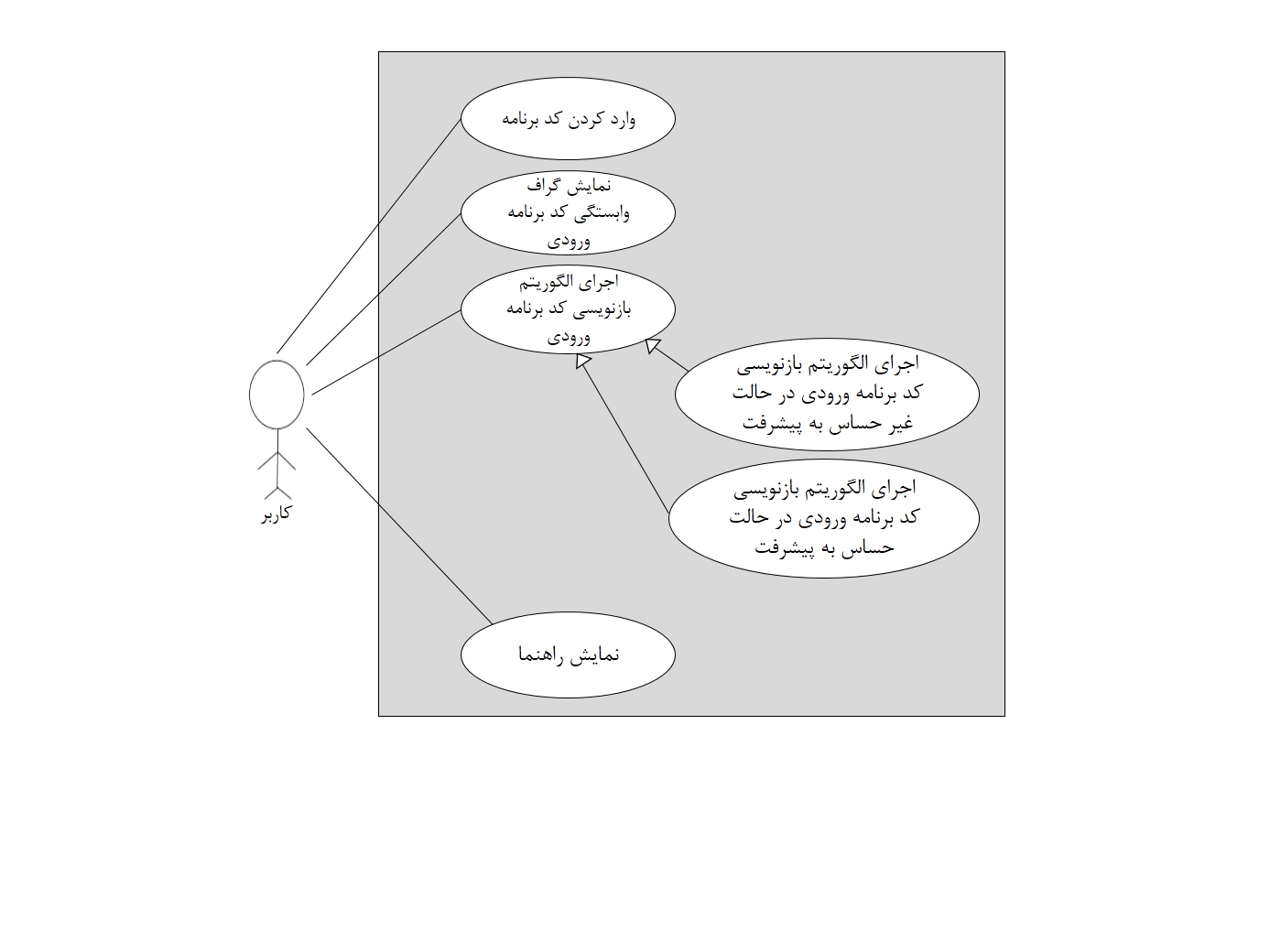
## تحلیل و طراحی نرم­افزار

با توجه به مشخص و ثابت بودن نیازهای این نرم­افزار در همان ابتدای تعریف پروژه، می‌توان از مدل فرآیندی آبشاری[[92]](#footnote-92) یا چرخه حیات کلاسیک[[93]](#footnote-93) استفاده کرد. هم‌چنین، روش تحلیل و طراحی این نرم‌افزار با رویکرد شی‌گرایی[[94]](#footnote-94) انجام شده است.

این مدل فرآیندی شامل پنج مرحله ارتباط[[95]](#footnote-95)، برنامه‌ریزی[[96]](#footnote-96)، مدل‌سازی[[97]](#footnote-97)، ساخت[[98]](#footnote-98) و استقرار[[99]](#footnote-99) است.

شکل 16 - مدل فرآیندی آبشاری

این مدل فرآیندی زمانی به کار بسته می‌شود که نیازمندی‌های پروژه کاملا خوش‌تعریف و پایدار باشند. مدل فرآیندی آبشاری یک روش ترتیبی و روش‌مند برای توسعه نرم‌افزار محسوب می‌شود که با مشخص کردن نیازمندی‌ها آغاز می‌شود و با گذر از مراحل برنامه‌ریزی، مدل‌سازی، ساخت و استقرار به پایان می‌رسد. گام اول به تعریف و جمع‌آوری نیازمندی‌های پروژه اختصاص پیدا می‌کند. پس از درک کامل آن‌ها، گام برنامه‌ریزی انجام می‌شود. در این مرحله، تخمین‌ها و برنامه‌ریزی‌های زمانی برآورد می‌شود. این تخمین‌ها و برنامه‌ریزی‌ها، شامل برآورد زمانی، هزینه‌، نیروی انسانی و سایر بخش‌هاست. قدم بعدی، مدل‌سازی یا همان تحلیل و طراحی نرم‌افزار خواهد بود. در این قسمت، با توجه به نیازمندی‌های پروژه، تحلیل‌های مربوط صورت می‌گیرد و مستندات و نمودارهای تحلیل و طراحی تولید می‌شوند. مهم‌ترین نمودار در مرحله تحلیل، نمودار موردکاربرد[[100]](#footnote-100) است که با توجه به نیازمندی‌ها و موردکاربردهای به دست آمده از تعریف پروژه ترسیم می‌شود. این نمودار مبنای تحلیل‌های بعدی خواهد بود. در ادامه نمودارهای موردکاربرد و فعالیت[[101]](#footnote-101) به عنوان بخشی از قسمت تحلیل آمده است.



شکل 17 - نمودار موردکاربرد پروژه

## شرح کلی مراحل پیاده­سازی و ابزارهای مورد استفاده

## ایجاد رابط کاربری گرافیکی[[102]](#footnote-102)

## راستی­آزمایی و آزمون

# منابع و مراجع

|  |  |
| --- | --- |
| Aggarwal, Charu C., and Philip S. Yu. *Finding generalized projected clusters in high dimensional spaces*. Vol. 29. No. 2. ACM, 2000. | [1] |
| Cabrera, Juan Manuel, Hugo Jair Escalante, and Manuel Montes-y-Gómez. "Distributional term representations for short-text categorization." Computational Linguistics and Intelligent Text Processing. Springer Berlin Heidelberg, 2013. 335-346. | [2] |
| Song, Ge, et al. "Short text classification: A survey." *Journal of Multimedia* 9.5 (2014): 635-643. | [3] |
| Phan, Xuan-Hieu, Le-Minh Nguyen, and Susumu Horiguchi. "Learning to classify short and sparse text & web with hidden topics from large-scale data collections." Proceedings of the 17th international conference on World Wide Web. ACM, 2008. | [4] |
|  |  |
|  |  |
|  |  |
|  |  |



Amirkabir University of Technology  
(Tehran Polytechnic)

Computer and Information Technology Engineering Department

B.Sc. Thesis

Title

…

By

Seyed Mohammad Mehdi Ahmadpanah

Supervisor

Dr. Mehran S. Fallah

September 2015

1. Language-based Security [↑](#footnote-ref-1)
2. Semantics [↑](#footnote-ref-2)
3. Types [↑](#footnote-ref-3)
4. Verification [↑](#footnote-ref-4)
5. Static Analysis [↑](#footnote-ref-5)
6. Compile Time [↑](#footnote-ref-6)
7. Dynamic Analysis [↑](#footnote-ref-7)
8. Run-time [↑](#footnote-ref-8)
9. Noninterference [↑](#footnote-ref-9)
10. Program Rewriting [↑](#footnote-ref-10)
11. Program Dependence Graph [↑](#footnote-ref-11)
12. Source Code [↑](#footnote-ref-12)
13. Policy [↑](#footnote-ref-13)
14. State [↑](#footnote-ref-14)
15. Property [↑](#footnote-ref-15)
16. Noninterference Security Policy [↑](#footnote-ref-16)
17. Observer [↑](#footnote-ref-17)
18. Progress-Sensitive [↑](#footnote-ref-18)
19. Progress-Insensitive [↑](#footnote-ref-19)
20. Progress Status [↑](#footnote-ref-20)
21. Divergence [↑](#footnote-ref-21)
22. Terminate [↑](#footnote-ref-22)
23. Information Flow [↑](#footnote-ref-23)
24. Explicit [↑](#footnote-ref-24)
25. Implicit [↑](#footnote-ref-25)
26. Assign [↑](#footnote-ref-26)
27. Attacker [↑](#footnote-ref-27)
28. Assignment [↑](#footnote-ref-28)
29. Monitor [↑](#footnote-ref-29)
30. Perl [↑](#footnote-ref-30)
31. JavaScript [↑](#footnote-ref-31)
32. Termination Channels [↑](#footnote-ref-32)
33. Syntactic [↑](#footnote-ref-33)
34. Execution Monitor [↑](#footnote-ref-34)
35. Abstract Events [↑](#footnote-ref-35)
36. Undecidable [↑](#footnote-ref-36)
37. Type System [↑](#footnote-ref-37)
38. Conservative [↑](#footnote-ref-38)
39. Co-recursively Enumerable [↑](#footnote-ref-39)
40. Platform [↑](#footnote-ref-40)
41. Characterization [↑](#footnote-ref-41)
42. Sound [↑](#footnote-ref-42)
43. Transparent [↑](#footnote-ref-43)
44. Path Conditions [↑](#footnote-ref-44)
45. Corrective Enforcement [↑](#footnote-ref-45)
46. Preorder [↑](#footnote-ref-46)
47. Abstract Syntax [↑](#footnote-ref-47)
48. Boolean [↑](#footnote-ref-48)
49. No Operation [↑](#footnote-ref-49)
50. Low [↑](#footnote-ref-50)
51. High [↑](#footnote-ref-51)
52. Trace [↑](#footnote-ref-52)
53. Event [↑](#footnote-ref-53)
54. Silent Divergence [↑](#footnote-ref-54)
55. Configuration [↑](#footnote-ref-55)
56. Terminal Configuration [↑](#footnote-ref-56)
57. Intermediate Configuration [↑](#footnote-ref-57)
58. Small-step Transition Rule [↑](#footnote-ref-58)
59. Judgment [↑](#footnote-ref-59)
60. Program Dependence Graph [↑](#footnote-ref-60)
61. Statement [↑](#footnote-ref-61)
62. Control Flow Graph [↑](#footnote-ref-62)
63. Path [↑](#footnote-ref-63)
64. Basic Block [↑](#footnote-ref-64)
65. Dummy [↑](#footnote-ref-65)
66. Parse Tree [↑](#footnote-ref-66)
67. Control Dependence Graph [↑](#footnote-ref-67)
68. Forward Dominance Tree [↑](#footnote-ref-68)
69. Post Dominance Tree [↑](#footnote-ref-69)
70. Dominance Tree [↑](#footnote-ref-70)
71. Strictly Dominate [↑](#footnote-ref-71)
72. Dominator [↑](#footnote-ref-72)
73. Immediate dominator [↑](#footnote-ref-73)
74. Post-Dominance Tree [↑](#footnote-ref-74)
75. Post-Dominance Frontier [↑](#footnote-ref-75)
76. Successor [↑](#footnote-ref-76)
77. Diverging points [↑](#footnote-ref-77)
78. Data Dependence Graph [↑](#footnote-ref-78)
79. Progress-Insensitive Non-Interference [↑](#footnote-ref-79)
80. Progress-Sensitive Non-Interference [↑](#footnote-ref-80)
81. Affect Function [↑](#footnote-ref-81)
82. Path Conditions [↑](#footnote-ref-82)
83. Incoming Edge [↑](#footnote-ref-83)
84. Execution Conditions [↑](#footnote-ref-84)
85. Backtracking [↑](#footnote-ref-85)
86. Static Single Assignment [↑](#footnote-ref-86)
87. Loop-carried Data Dependency [↑](#footnote-ref-87)
88. Termination Oracle [↑](#footnote-ref-88)
89. Loop Analyzer [↑](#footnote-ref-89)
90. Loop Guards [↑](#footnote-ref-90)
91. Nested Loops [↑](#footnote-ref-91)
92. Waterfall Process Model [↑](#footnote-ref-92)
93. Classic Life Cycle [↑](#footnote-ref-93)
94. Object-Oriented [↑](#footnote-ref-94)
95. Communication [↑](#footnote-ref-95)
96. Planning [↑](#footnote-ref-96)
97. Modeling [↑](#footnote-ref-97)
98. Construction [↑](#footnote-ref-98)
99. Deployment [↑](#footnote-ref-99)
100. Use Case Diagram [↑](#footnote-ref-100)
101. Activity Diagram [↑](#footnote-ref-101)
102. Graphical User Interface [↑](#footnote-ref-102)