دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده 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 ۱۳۹۴

به نام خدا

تعهدنامه اصالت اثر

تاریخ:

اینجانب سید محمدمهدی احمدپناه متعهد می‌شوم که مطالب مندرج در این پایان نامه حاصل کار پژوهشی اینجانب تحت نظارت و راهنمایی اساتید دانشگاه صنعتی امیرکبیر بوده و به دستاوردهای دیگران که در این پژوهش از آنها استفاده شده است، مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذکر گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم‌سطح یا بالاتر ارائه نگردیده است.

در صورت اثبات تخلف در هر زمان، مدرک تحصیلی صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پیگیری قانونی خواهد داشت.

کلیه نتایج و حقوق حاصل از این پایان نامه متعلق به دانشگاه صنعتی امیرکبیر می‌باشد. هرگونه استفاده از نتایج علمی و عملی، واگذاری اطلاعات به دیگران یا چاپ و تکثیر، نسخه‌برداری، ترجمه و اقتباس از این پایان نامه بدون موافقت کتبی دانشگاه صنعتی امیرکبیر ممنوع است. نقل مطالب با ذکر 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 پاک 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؛

از جناب آقای دکتر افشین لامعی که از آغاز پروژه در درک بهتر مفاهیم و بخش‌های مختلف مقاله مرا راهنمایی کردند؛

و در پایان، از حمایت‌ها و کمک‌های دوستان عزیزم، آقایان محمد پزشکی، بهنام ستارزاده، علی قنبری، مسعود غفاری‌نیا، احسان عدالت، حمیدرضا رمضانی و پرهام الوانی که در طول انجام پروژه از نظرات و راهنمایی‌هایشان استفاده کردم؛

کمال تشکر و قدردانی را دارم.

چکیده

خط مشی امنیتی امن بودن یک سیستم یا برنامه را تعریف می‌کند. در واقع خط مشی با ذکر قیود و محدودیت‌هایی، منظور ما از امنیت را بیان می‌کند. در این پروژه، خط مشی امنیتی عدم تداخل به عنوان مقصود ما از امنیت مطرح می‌شود. به زبان ساده این خط مشی بیان می‌کند که یک مشاهده‌گر سطح پایین که فقط به برنامه و مقادیر عمومی زمان‌اجرا دسترسی دارد، نتواند درکی نسبت به ورودی‌های سطح بالا یا خصوصی پیدا کند. به عبارت دیگر، در هر جفت اجراهای برنامه که ورودی‌های عمومی یکسان دارند، مستقل از ورودی‌های خصوصی متفاوت، باید خروجی‌های عمومی یکی باشند. نکته مهم این است که خط مشی عدم تداخل، یک خاصیت نیست. این مسئله باعث ایجاد محدودیت‌هایی برای اعمال این خط مشی در برنامه‌ها می‌شود. در این پروژه، با تقسیم‌بندی این خط مشی به دو حالت غیرحساس و حساس به پیشرفت، سعی در بیان دقیق‌تر این خط مشی داریم.

برای اعمال خط مشی‌ها، روش‌ها و راهکارهای گوناگونی وجود دارد که بسته به دسته خط مشی مورد نظر، می‌توان آن‌ها را به کار بست که خود یکی از چالشی‌ترین مسائل موجود در این حوزه به شمار می‌رود. هدف از انجام این پروژه، طراحی و پیاده‌سازی ابزاری است که بتواند با استفاده از روش بازنویسی برنامه، خط مشی عدم تداخل را اعمال کند تا برنامه‌های تبدیل شده، حتماً این خط مشی را برآورده سازند. در این روش، گراف وابستگی برنامه در کنار کد مبدأ، ورودی‌های الگوریتم بازنویس خواهند بود و نتیجه آن، تغییر و جایگزینی دستوراتی است که با پیمایش در گراف وابستگی برنامه، تهدیدی برای جریان اطلاعات سطح بالا به مشاهده‌گران سطح پایین تلقی می‌شوند. اساساً روش بازنویسی برنامه، برخلاف مکانیزم‌های دیگر اعمال خط مشی‌ها، برنامه مغایر با خط مشی را چه قبل یا چه در زمان‌اجرا، رد نمی‌کند؛ بلکه آن‌ها بازنویسی می‌شوند و از برنامه‌های ناامن به برنامه‌های امن تبدیل می‌شوند. سلامت و شفافیت روش مورد استفاده، که مهم‌ترین عوامل مقایسه روش‌های بازنویسی به شمار می‌روند، اثبات شده است.

واژه‌های کلیدی:

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

صفحه

فهرست عناوین

فهرست اشکال

صفحه

۱

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] می‌باشد. امنیت زبان‌مبنا را می‌توان مجموعه‌ای از تکنیک‌های مبتنی بر نظریه زبان‌های برنامه‌سازی و پیاده‌سازی آن‌ها، شامل معناشناخت[2]، نوع‌ها[3]، بهینه‌سازی و راستی‌آزمایی[4]، برای به کار گیری در مسائل امنیتی تعریف کرد. [۱] 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] یا زمان‌کامپایل[6] و تحلیل پویا[7] یا زمان‌اجرا[8] دسته‌بندی کرد. هر کدام از این روش‌ها نقاط قوت و ضعف

---

[1] Language-based Security

[2] Semantics

[3] Types

[4] Verification

[5] Static Analysis

[6] Compile Time

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] 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]، 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] و کاربرد آن در پروژه بحث خواهد شد. فصل پنجم به توضیح الگوریتم مورد نظر برای بازنویسی کد مبدأ[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.

---

[7] Dynamic Analysis

[8] Run-time

[9] Noninterference

[10] Program Rewriting

[11] Program Dependence Graph

[12] Source Code

۲

فصل دوم

خط مشیِ امنیتیِ عدم تداخل و اِعمال آن

خط مشی امنیتی عدم تداخل و اعمال آن

به طور کلی، خط مشیِ[۱۳] 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[۱۴]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*[۱۵] 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. [۲] برخی از نیازمندی‌های مهم امنیتی، خاصیت نیستند. یک نمونه مهم از این‌گونه نیازمندی‌ها، خط مشیِ امنیتی عدم تداخل[۱۶] است. عدم تداخل گوگن-مسگر [۳]، عدم تداخل تعمیم یافته [۴] و قطعیت مبتنی بر مشاهده [۵] 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[۱۷] 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

---

[13] Policy

[14] State

[15] Property

[16] Noninterference Security Policy

[17] Observer

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. [۲] 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*[۱۸] و *غیر حساس به پیشرفت*[۱۹] 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*[۲۰] برنامه نیز دسترسی دارد. این باعث می‌شود تا بتواند تفاوت بین واگرایی[۲۱] برنامه با موقعیتی که برنامه خاتمه[۲۲] می‌یابد یا در حال محاسبه مقادیر قابل‌مشاهده بعدی است، را تمیز دهد. [۲]

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[۲۳] از سطح بالا به پایین ممکن است صریح[۲۴] یا ضمنی[۲۵] باشد. انتساب[۲۶] 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

[18] Progress-Sensitive

[19] Progress-Insensitive

[20] Progress Status

[21] Divergence

[22] Terminate

[23] Information Flow

[24] Explicit

[25] Implicit

[26] Assign

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[۲۷] 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 [۶] یک روش تبدیل برنامه‌ی ترکیبی برای اِعمال عدم تداخل ارائه داده‌اند. برنامه تغییر داده شده، سطوح امنیتی انتساب[۲۸] 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.

مَگزینیوس و همکارانش [۷] یک چارچوب برای ناظر[۲۹]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[۳۰] و جاوااسکریپت[۳۱] 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 [۸] 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

[27] Attacker

[28] Assignment

[29] Monitor

[30] Perl

[31] JavaScript

شود. همچنین این روش، اجازه وقوع مجراهای خاتمه[۳۲] را نمی‌دهد. سانتوس و رِزک [۹] 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. [۱۰] این موضوع باعث می‌شود که پروژه‌هایی که محدودیت‌های نَحوی[۳۳] بر روی کد دارند، از اطلاعات ایستا در ناظری بر اجراهای چندگانه‌ی برنامه‌ها استفاده کنند.

بِلو و بونِلی [۱۱] یک ناظرِ اجرایی[۳۴] 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 [۱۲]یک ماشین طراحی کرده‌اند که رخدادهای انتزاعی[۳۵] 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[۳۶] است. پس در حالت کلی، عدم تداخل توسط روش‌های ایستا قابل اعمال نیست؛ به همین دلیل است که نوع‌سامانه[۳۷]های ارائه‌شده برای این مسئله، محافظه‌کارانه[۳۸] هستند و ممکن است بعضی برنامه‌های امن را نیز نپذیرند. از طرفی، این مسئله هم‌بازگشتی شمارش‌پذیر[۳۹] نیز

---

[32] Termination Channels

[33] Syntactic

[34] Execution Monitor

[35] Abstract Events

[36] Undecidable

[37] Type System

[38] Conservative

[39] Co-recursively Enumerable

نیست. بنابراین، توسط ناظرهای اجرایی که نقضِ خط مشی عدم تداخل در یک برنامه‌ی در حال اجرا را بررسی می‌کنند، قابل اعمال نیست. [۲]

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[۴۰]های سخت‌افزاری، دستگاه‌ها و بهینه‌سازی کارایی استفاده می‌شد. [۱۳] این روش اخیراً به عنوان وسیله‌ای برای اعمال خط مشی‌های امنیتی پیشنهاد شده است. [۴۶] 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[۴۱] 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. [۱۵] 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[۴۲] و شفاف[۴۳] باشد. سالم بودن به این معنا که کد تولید شده توسط آن، خط مشی را برآورده کند و شفاف بودن به معنای این‌که

---

[40] Platform

[41] Characterization

[42] Sound

[43] Transparent

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. [۲] 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 [۱۶] استفاده می‌کند که قبلاً اثبات شده است که در تشخیص جریان‌های اطلاعاتی احتمالی، دست‌کم به قدرت نوع‌سامانه‌های امنیتی است. [۱۷] 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[۴۴] 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*[۴۵] برای خط مشی عدم تداخلِ مطرح شده، به شکل دقیق و صوری و با بهره‌گیری از رابطه پیش‌ترتیبی[۴۶] 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.

[44] Path Conditions

[45] Corrective Enforcement

[46] Preorder

اثبات شده است که روش مورد استفاده در این پروژه سالم است، پس قطعاً برنامه‌های امن را تولید خواهد کرد. [۲]

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

۳

فصل سوم

توصیف زبان برنامه‌نویسی WL

توصیف زبان برنامه‌نویسی WL

در مقاله اصلی مورد استفاده در این پروژه [۲]، 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 [۱۸] و bison [۱۹] 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 ۱ - ساختار نحوی زبان برنامه‌نویسی WL

در شکل ۱، ساختار نحوی تجریدی[۴۷] زبانِ WL آمده است. یک عبارت، یا یک عدد صحیح ثابت یا مقدار منطقی بولی[۴۸]، 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*[۴۹]، 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[۵۰] یا بالا[۵۱] 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 [۲] 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[۵۲] دنباله‌ای از حالت‌هاست که انتقال به یک حالت ممکن است همراه با وقوع یک رویداد[۵۳] 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[۵۴] را نیز در شرایطی که توسط مشاهده‌گر سطح

[47] Abstract Syntax

[48] Boolean

[49] No Operation

[50] Low

[51] High

[52] Trace

[53] Event

[54] Silent Divergence

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[۵۵] استفاده می‌شود و معناشناخت بر اساس آن بیان می‌شود. یک پیکربندی در یکی از دو دسته زیر قرار می‌گیرد:

- پیکربندی پایانی[۵۶] مانند $(\epsilon, \sigma, \check{S}, \hat{S})$ که در آن، ϵ به معنای یک رشته خالی، σ بیانگر حالتِ پیکربندی، $\check{S}$ نشانه دنباله ورودی‌ها و $\hat{S}$ برای دنباله خروجی‌ها می‌باشد.

- پیکربندی میانی[۵۷] مانند $(c, \sigma, \check{S}, \hat{S})$ 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 کوچک‌گامی[۵۸] '$\rightarrow$' روی C تعریف می‌شود که در آن عبارت‌ها به عبارت‌های ریاضی یا منطق بولی تعبیر می‌شود. حکمِ[۵۹] $(c, \sigma, \check{S}, \hat{S}) \rightarrow (c', \sigma', \check{S}', \hat{S}')$ به این معناست که اجرای دستور c در حالت σ و با دنباله ورودی‌ها و خروجی‌های $\check{S}$ و $\hat{S}$، پیکربندی $(c', \sigma', \check{S}', \hat{S}')$ را نتیجه می‌دهد. البته ممکن است یک انتقال هیچ رویدادی را تولید نکند؛ یعنی $\check{S}' = \check{S}$ و $\hat{S}' = \hat{S}$باشد. در صورتی‌که یک انتقال باعث تولید یک رویداد شود، $\check{S}' = \check{S} \,.\, \langle e \rangle$ یا $\hat{S}' = \hat{S} \,.\, \langle e \rangle$ 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 $\sigma : X \rightarrow N \cup \{null\}$ تعریف می‌شود که در آن، N مجموعه همه اعداد صحیح و 'null' مقدار متغیرها قبل از انتساب به ورودی‌های گرفته شده از محیط باشد. $\sigma(x)$، مقدار x در حالت σ خواهد بود و

---

[55] Configuration

[56] Terminal Configuration

[57] Intermediate Configuration

[58] Small-step Transition Rule

[59] Judgment

σ [x ↦ v] حالت حاصل از σ توسط به‌روزرسانی مقدار x به v است. مقدار عبارت exp در σ نیز توسط σ(exp) نمایش داده می‌شود. منظور از ’Γ ↓ v‘ این است که ورودی بعدی که توسط محیط فراهم خواهد شد v با سطح امنیتی Γ می‌باشد. در شکل ۲، 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.

$$\frac{}{(\mathbf{Nop}, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S})}\ \textsc{Nop1}$$

$$\frac{}{(x = exp, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma[x \mapsto \sigma(exp)], \check{S}, \hat{S})}\ \textsc{Assign}$$

$$\frac{\Gamma \downarrow v}{(in_\Gamma\ x, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma[x \mapsto v], \check{S} \cdot \langle \check{v}_\Gamma \rangle, \hat{S})}\ \textsc{InputVar}$$

$$\frac{\Gamma \downarrow v}{((in_\Gamma\ x,\ varlist), \sigma, \check{S}, \hat{S}) \longrightarrow (in_\Gamma\ varlist, \sigma[x \mapsto v], \check{S} \cdot \langle \check{v}_\Gamma \rangle, \hat{S})}\ \textsc{InputVarList}$$

$$\frac{\sigma(x) = v}{(out_\Gamma\ x, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S} \cdot \langle \hat{v}_\Gamma \rangle)}\ \textsc{OutputVar}$$

$$\frac{}{(out_\Gamma\ \bot, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S} \cdot \langle \hat{\bot}_\Gamma \rangle)}\ \textsc{Output}\bot$$

$$\frac{(c_1, \sigma, \check{S}, \hat{S}) \longrightarrow (c_1', \sigma', \check{S}', \hat{S}')}{(c_1; c_2, \sigma, \check{S}, \hat{S}) \longrightarrow (c_1'; c_2, \sigma', \check{S}', \hat{S}')}\ \textsc{Seq1}$$

$$\frac{}{(\epsilon; c_2, \sigma, \check{S}, \hat{S}) \longrightarrow (c_2, \sigma, \check{S}, \hat{S})}\ \textsc{Seq2}$$

$$\frac{\sigma(exp) = \text{true}}{(\mathbf{if}\ exp\ \mathbf{then}\ c\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (c, \sigma, \check{S}, \hat{S})}\ \textsc{If-True}$$

$$\frac{\sigma(exp) = \text{false}}{(\mathbf{if}\ exp\ \mathbf{then}\ c\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (\epsilon, \sigma, \check{S}, \hat{S})}\ \textsc{If-False}$$

$$\frac{\sigma(exp) = \text{true}}{(\mathbf{if}\ exp\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (c_1, \sigma, \check{S}, \hat{S})}\ \textsc{If-Else-True}$$

$$\frac{\sigma(exp) = \text{false}}{(\mathbf{if}\ exp\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{endif}, \sigma, \check{S}, \hat{S}) \longrightarrow (c_2, \sigma, \check{S}, \hat{S})}\ \textsc{If-Else-False}$$

$$\frac{}{(\mathbf{while}\ exp\ \mathbf{do}\ c, \sigma, \check{S}, \hat{S}) \longrightarrow (\mathbf{if}\ exp\ \mathbf{then}\ (c; \mathbf{while}\ exp\ \mathbf{do}\ c)\ \mathbf{else}\ \mathbf{Nop}, \sigma, \check{S}, \hat{S})}\ \textsc{While}$$

شکل ۲ - معناشناخت کوچک‌گامی برای زبان WL

۴

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]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]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] 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 $X \xrightarrow{d} Y$ 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 $X \xrightarrow{c} Y$ نمایش داده می‌شود، به معنای این است که اجرای گزاره Y، توسط مقدار محاسبه‌شده در گزاره X کنترل می‌شود. ضمناً می‌توان یک مسیر[63] از

[60] Program Dependence Graph

[61] Statement

[62] Control Flow Graph

[63] Path

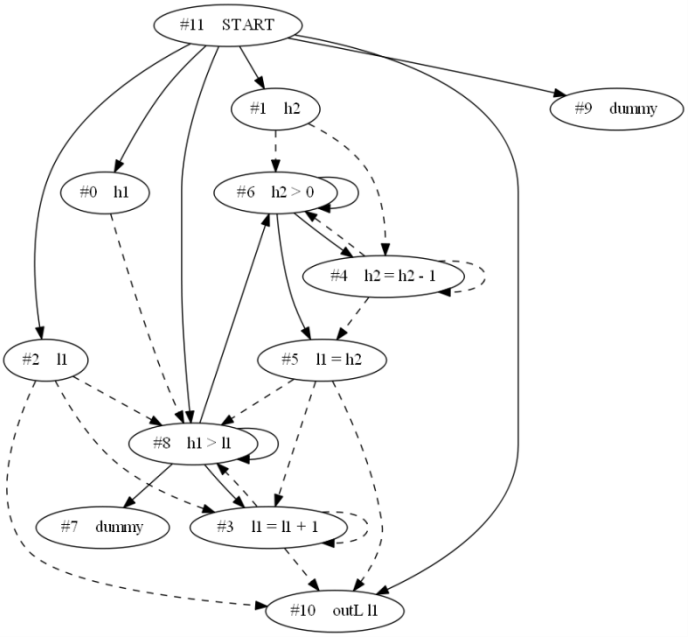
X به Y در گراف وابستگی برنامه را به شکل $X \rightsquigarrow Y$ علامت‌گذاری کرد. هر مسیر مشخص‌کننده یک وابستگی کنترلی یا داده‌ای است که بستگی به نوعِ آخرین یال آن مسیر دارد. در ضمن، مسیر ساخته‌شده به واسطه اضافه کردن یال $X \rightarrow Y$ -که به این معناست که نوع وابستگی آن اهمیتی ندارد- به مسیر $Y \rightsquigarrow Z$، به شکل $X \rightarrow Y \rightsquigarrow Z$ نمایش داده می‌شود. مسیری مانند $X \rightsquigarrow Y$ 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 $X \rightsquigarrow Y$ 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 $X \rightsquigarrow Y$ وابسته باشد و اجرای آن گزاره، توسط مقدار محاسبه‌شده در X کنترل شود.

پس با تعاریف فوق می‌توان چنین گفت که مسیر $X \rightsquigarrow Y$ 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
```

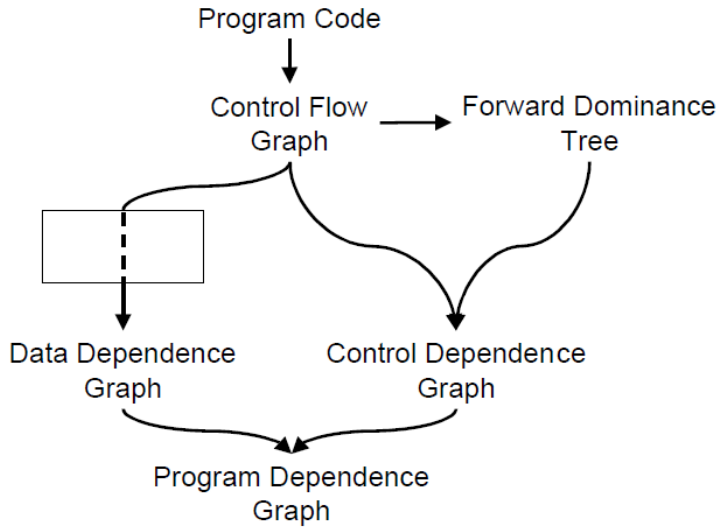


شکل ۱ - نمونه کد مبدأ به زبان WL و گراف وابستگی برنامه مربوط به آن

همان‌طور که در شکل شماره ۳ مشاهده می‌شود، یال‌های با خطوط ساده نماد وابستگی‌های کنترلی و یال‌های خط‌چین نمایانگر وابستگی‌های داده‌ای هستند.

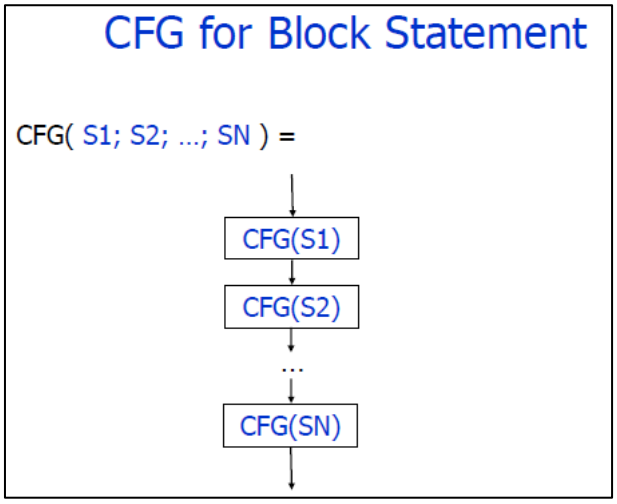
در ادامه این فصل به توضیح نحوه ساختن گراف وابستگی برنامه از روی کد مبدأ برای زبان برنامه‌نویسی WL می‌پردازیم و در فصل بعدی، نحوه استفاده از این گراف در الگوریتم بازنویسی برنامه را به تفصیل شرح خواهیم داد.

برای ساخت گراف وابستگی برنامه، مطابق با شکل شماره ۴، گراف‌های مورد نیاز برای تحلیل ساخته می‌شود.



شکل ۲ - نمودار کلی نحوه تولید گراف وابستگی برنامه از روی کد مبدأ برنامه [۲۰]

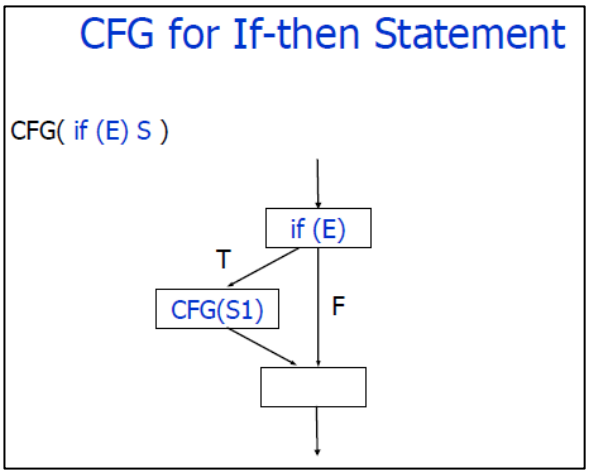
ابتدا از روی کد مبدأ، گراف جریان کنترل یا CFG به دست می‌آید. نحوه تولید این گراف به این نحو است که گزاره‌ها و عبارت‌های برنامه، به عنوان یک گره در گراف در نظر گرفته می‌شود. در این گراف، مفهومی به نام بلوک پایه[۶۴] مطرح می‌شود. هر بلوک پایه، شامل تعدادی گره است که تنها یک گره ورودی و یک گره خروجی در آن 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 ۵، 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.



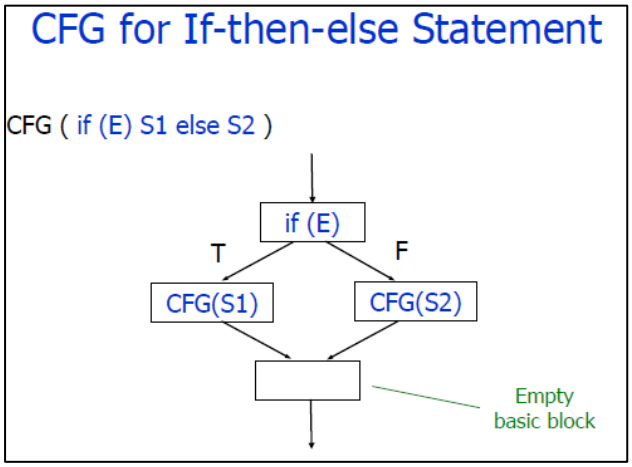
شکل ۳ - نحوه تولید زیرگراف بلوک پایه و اتصال به یکدیگر [۲۰]

[64] Basic Block

برای گزاره‌های شرطی، دو حالت ممکن زیر وجود دارد:



شکل ۴ - نحوه تولید زیرگراف گزاره‌های شرطی – حالت اول [۲۰]

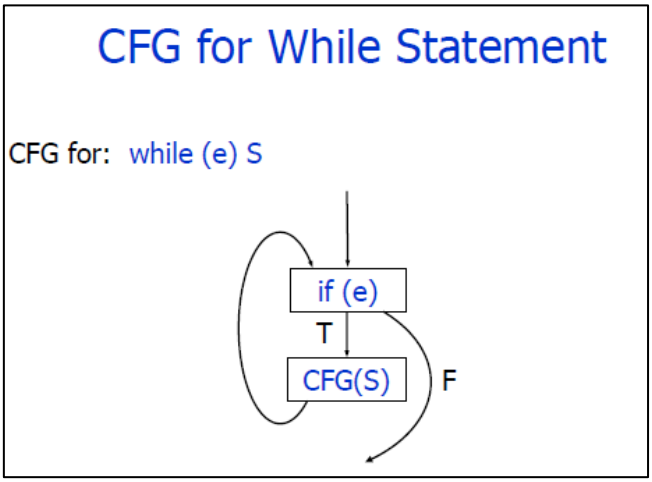


شکل ۵ - نحوه تولید زیرگراف گزاره‌های شرطی - حالت دوم [۲۰]

مطابق با شکل‌های شماره ۶ و ۷، 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[۶۵] 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 ۸ عمل می‌شود.



شکل ۶ - نحوه تولید زیرگراف گزاره‌های حلقه while [۲۰]

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[۶۶] ساخته می‌شوند و با اتصال آن‌ها به یکدیگر، گراف کنترل جریان برنامه به دست خواهد آمد.

مطابق با شکل شماره ۴، برای تولید گراف وابستگی کنترل یا به اختصار CDG[۶۷]، به درخت غلبه رو به جلو[۶۸] یا درخت پس‌غلبه[۶۹] نیاز خواهد بود. برای تولید این درخت، الگوریتم ساخت درخت

---

[65] Dummy

[66] Parse Tree

[67] Control Dependence Graph

[68] Forward Dominance Tree

[69] Post Dominance Tree

غلبه[۷۰] 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[۷۱] 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[۷۲] داشته باشد، اما برای تولید درخت غلبه، نزدیک‌ترین غلبه‌کننده یا غلبه‌کننده بی‌درنگ[۷۳] 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 ϕ≠ ) {
                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 ۷ - محاسبه گره‌های غلبه‌کننده برای هر گره [۲۱]

[70] Dominance Tree

[71] Strictly Dominate

[72] Dominator

[73] Immediate dominator

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[۷۴] 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[۷۵] یا اختصاراً PDF، مورد نیاز است. مرز پس‌غلبه گره X، مجموعه گره‌هایی هستند که توسط X اکیداً پس‌غلبه نمی‌شوند اما گره‌های مابَعدی[۷۶] دارند که توسط X پس‌غلبه می‌شوند. تعریف ریاضی این گره‌ها بدین شرح است:

$PDF(X) = \{y \mid (\exists z \in Succ(y)$ such that x post-dominates z) and x does not strictly post-dominate y$\}$

که این مجموعه بیانگر نزدیک‌ترین نقاط انشعابی[۷۷] 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.

---

[74] Post-Dominance Tree

[75] Post-Dominance Frontier

[76] Successor

[77] Diverging points

```
For each x in the bottom-up traversal of the postdominator tree do
    PDF(X) = ϕ
    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
```

شکل ۸- محاسبه گره‌های پس‌غلبه‌کننده مرزی برای هر گره

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 DDG[۷۸] 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.

[78] Data Dependence Graph

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.

۵

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[۷۹] و حالت حساس به پیشرفت یا PSNI[۸۰] خواهیم پرداخت.

با داشتن گراف وابستگی برنامه کد مبدأ ورودی، می‌توان وابستگی‌های موجود بین گزاره‌های مختلف برنامه را بررسی کرد. می‌توان تابع متأثر کردن[۸۱] 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. [۲]

foreach statement X producing a high input event h_{in} **do**
    foreach statement Y producing a low observable event e_L **do**
        if $Y \in affect(X)$ **then**
            transform Y into Y′ such that $Y' \notin affect(X)$ in the new program
        end
    end
end

شکل ۹ – الگوریتم کلی بازنویسی برای اعمال خط مشی عدم تداخل [۲]

[79] Progress-Insensitive Non-Interference

[80] Progress-Sensitive Non-Interference

[81] Affect Function

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 ۹ باید شرط‌هایی که تعیین‌کننده وقوع جریان غیرمجاز احتمالی هستند را لحاظ کرد.

۱.۵ 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[۸۲] 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 $X \rightsquigarrow Y$ 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[۸۳] 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 $inH\ h \rightsquigarrow$ outL l 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،

[82] Path Conditions

[83] Incoming Edge

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[۸۴] گره‌ها به دست می‌آیند. به طور کلی، شرط اجرا برای گره X با پیمایش معکوس[۸۵] 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 $X \rightsquigarrow Y$، 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[۸۶] 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[۸۷] در مسیرهای از ورودی‌های سطح بالا به خروجی‌های سطح پایین برنامه وجود ندارد.

در الگوریتم شکل شماره ۱۰، 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

[84] Execution Conditions

[85] Backtracking

[86] Static Single Assignment

[87] Loop-carried Data Dependency

مبدأ M′ 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 ۱۱ آورده شده است.

RW_{PINI} (M, G):

Initialize F to the set of all paths $Start \hookrightarrow P \rightsquigarrow P'$ in the PDG G of M where P is the node representing a high input and P' is the node representing outL l for some l;

if $F = \emptyset$ then

    return M;

end

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

determine the type of flow indicated by each path $f \in F$;

foreach $f \in F$ do:

    Generate the path condition of f as the conjunction of the execution conditions of node N satisfying $f = Start \rightsquigarrow X \xrightarrow{d} N \rightsquigarrow P'$ 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 $f' \in F$ which terminate at n;

    if all paths $f' \in F$ 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”;

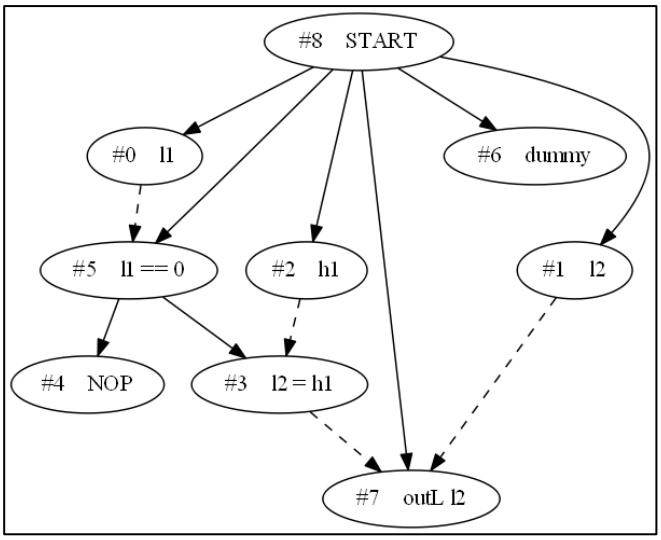
    end

end

return M′;

شکل ۱۰ - الگوریتم بازنویسی عدم تداخل حالت غیرحساس به پیشرفت که برنامه M و گراف وابستگی برنامه مربوط به آنِ G را می‌گیرد [۲]

```
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
```

شکل ۱۱ – (الف) نمونه کد به زبان WL؛ (ب) گراف وابستگی برنامه الف؛ (ج) برنامه بازنویسی‌شده برنامه الف در حالت غیرحساس به پیشرفت

۲.۵ 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 ۱۲، 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
```

(الف)

```
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
```

(ب)

شکل ۱۲ - (الف) نمونه برنامه به زبان WL؛ (ب) برنامه بازنویسی‌شده الف برای حالت غیرحساس به پیشرفت، که حالت حساس به پیشرفت را برآورده نمی‌کند

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 [۲۲] یک نوع‌سامانه به همراه یک مکانیزم زمان‌اجرا به نام پیش‌گویی خاتمه[۸۸] 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[۸۹] 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. [۲]

[88] Termination Oracle

[89] Loop Analyzer

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 ۱۳، 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 ۱۳ – 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. [۲]

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[۹۰] 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.

الگوریتم بازنویسی مطرح‌شده در شکل شماره ۱۴، کد مبدأ برنامه M و گراف وابستگی برنامه متناظر آن را می‌گیرد و کد M′ 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 $Start \hookrightarrow h \hookrightarrow E^+$ هستند، که در آن h یک ورودی سطح بالا و E^+ 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 M′ متناظر با گره n در گراف وابستگی برنامه را برمی‌گرداند.

[90] Loop Guards

RW_{PSNI}(M, G):

Initialize D to the set of all paths $Start \hookrightarrow P \hookrightarrow E^+$ in G where E^+ is a path terminating at a loop guard and P is the node representing a high input;

$M' = RW_{PINI}$(M, G);

if $D = \emptyset$ then

return M′;

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 M′ as follows:

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

foreach node n with *height(n)* = h representing a loop on some path $f \in D$ **do**

r = LoopAnalyzer(loop(n));

if r = False **then**

if $X \xrightarrow{c} n$ appears on at least one path $f \in D$ **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";

شکل ۱۴ - الگوریتم بازنویسی عدم تداخل حالت حساس به پیشرفت که برنامه M و گراف وابستگی برنامه مربوط به آنِ G را می‌گیرد [۲]

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] 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 ۱۵، نمونه کد برنامه تبدیل شده توسط این الگوریتم برای کد مبدأ برنامه شکل شماره ۱۳ را مشاهده می‌شود.

برای اثبات سلامت و شفافیت الگوریتم‌های استفاده‌شده در این پروژه، می‌توانید به مقاله اصلی پروژه [۲] 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 ۱۵ - کد مبدأ بازنویسی‌شده توسط الگوریتم حالت حساس به پیشرفت برای برنامه شکل شماره ۱۳

[91] Nested Loops

۶

فصل ششم

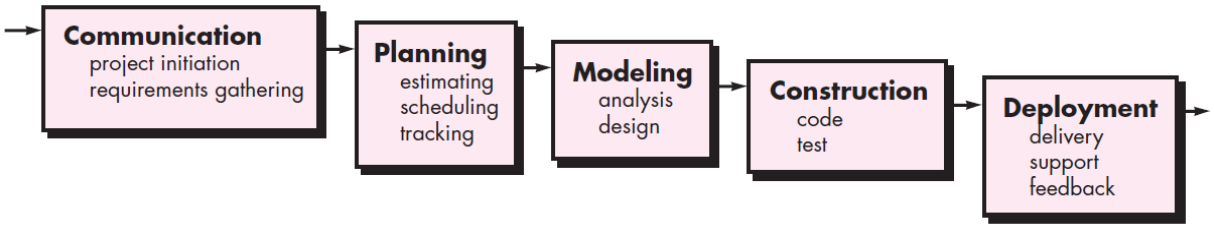
پیاده‌سازی و ایجاد رابط کاربری

پیاده‌سازی و ایجاد رابط کاربری

۱.۶ تحلیل و طراحی نرم‌افزار

با توجه به مشخص و ثابت بودن نیازهای این نرم‌افزار در همان ابتدای تعریف پروژه، می‌توان از مدل فرآیندی آبشاری[۹۲] یا چرخه حیات کلاسیک[۹۳] استفاده کرد. هم‌چنین، روش تحلیل و طراحی این نرم‌افزار با رویکرد شی‌گرایی[۹۴] انجام شده است.

این مدل فرآیندی شامل پنج مرحله ارتباط[۹۵]، برنامه‌ریزی[۹۶]، مدل‌سازی[۹۷]، ساخت[۹۸] و استقرار[۹۹] است.



شکل ۱۶ - 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

---

[92] Waterfall Process Model

[93] Classic Life Cycle

[94] Object-Oriented

[95] Communication

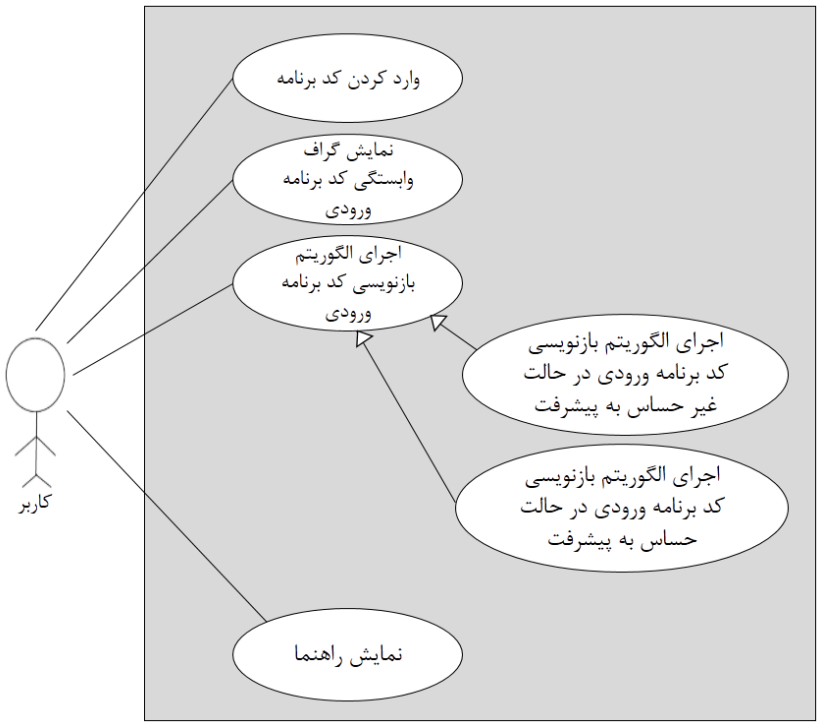
[96] Planning

[97] Modeling

[98] Construction

[99] Deployment

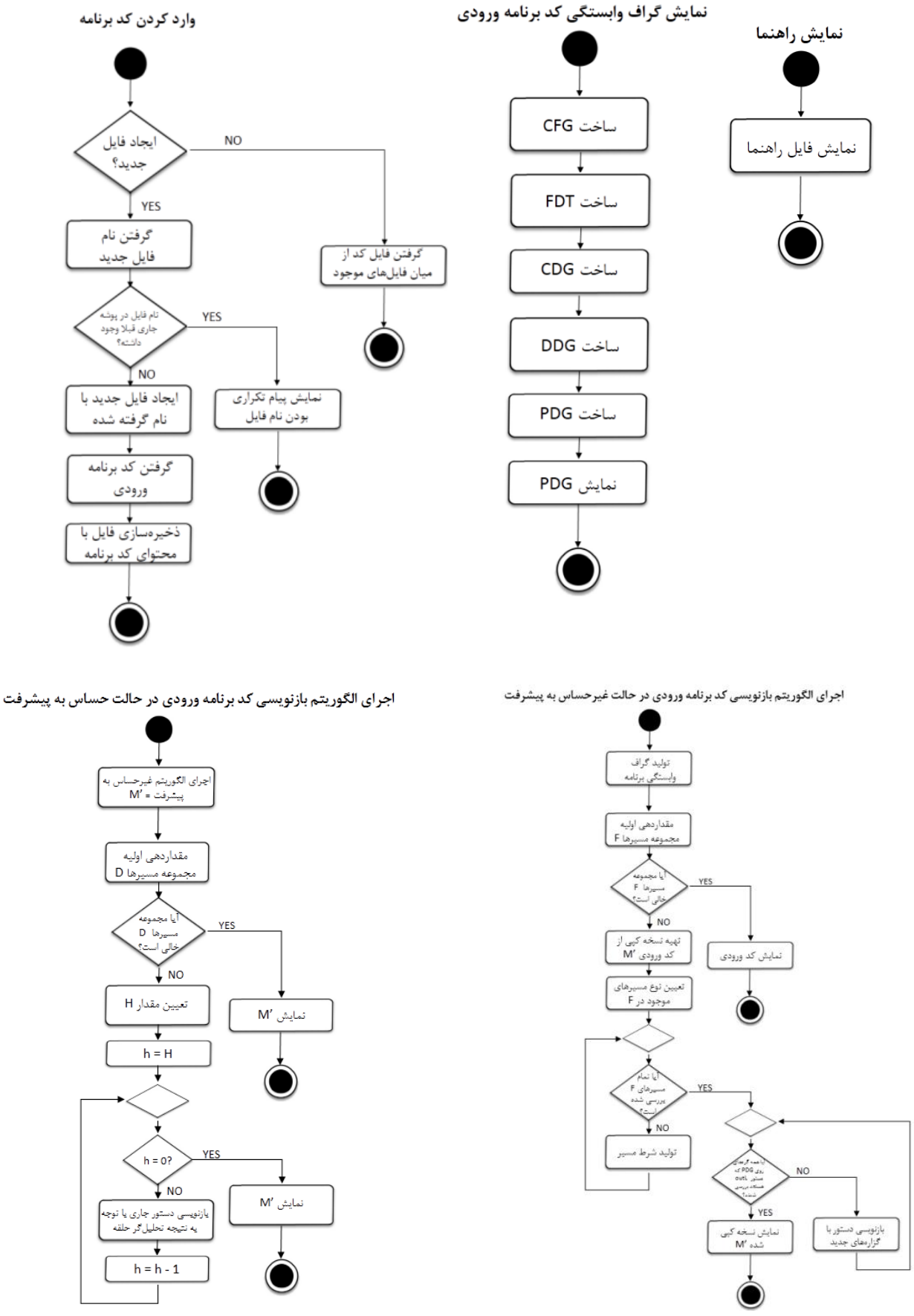
پایان می‌رسد. گام اول به تعریف و جمع‌آوری نیازمندی‌های پروژه اختصاص پیدا می‌کند. پس از درک کامل آن‌ها، گام برنامه‌ریزی انجام می‌شود. در این مرحله، تخمین‌ها و برنامه‌ریزی‌های زمانی برآورد می‌شود. این تخمین‌ها و برنامه‌ریزی‌ها، شامل برآورد زمانی، هزینه، نیروی انسانی و سایر بخش‌هاست. قدم بعدی، مدل‌سازی 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] 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] به عنوان بخشی از قسمت تحلیل آمده است.



شکل ۱۷ - نمودار موردکاربرد نرم‌افزار پروژه

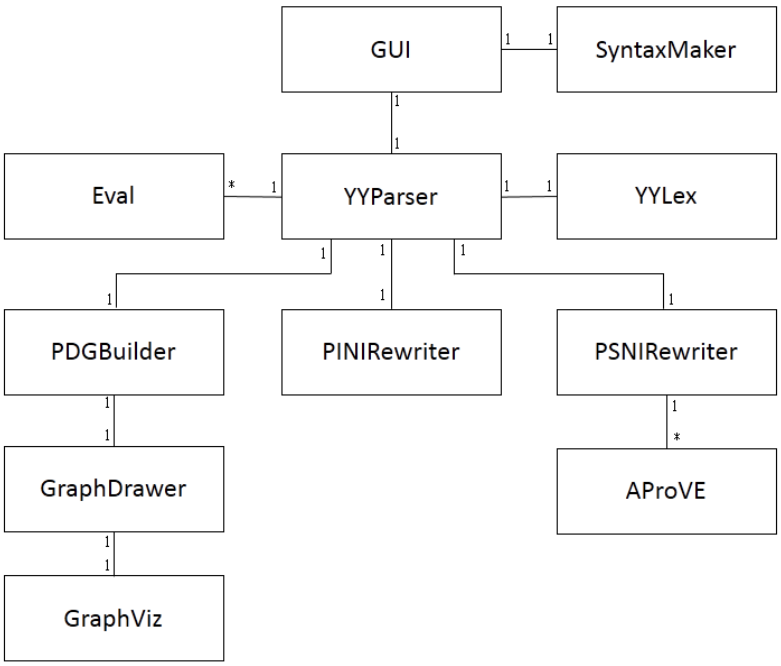
[100] Use Case Diagram

[101] Activity Diagram



شکل ۱۸ – نمودارهای فعالیت نرم‌افزار پروژه

در مرحله طراحی، مهم‌ترین نمودار که آینه تمام‌نمای معماری نرم‌افزار نیز به شمار می‌رود، نمودار کلاس[۱۰۲] است. این نمودار کلاس‌های مورد استفاده در نرم‌افزار و نحوه ارتباط بین آن‌ها را مشخص می‌کند. در واقع این نمودار، مرز بین تحلیل و طراحی است و از این نمودار به عنوان مبنای نمودارها و طراحی‌های نرم‌افزار می‌توان نام برد. در شکل شماره ۱۹، نمودار کلاس نرم‌افزار این پروژه را مشاهده می‌کنید. در این شکل، جزئیات فیلدها و متدهای هر کلاس آورده نشده و تنها به نام کلاس‌ها و ارتباط بین آن‌ها اکتفا شده است.



شکل ۱۹ – نمودار کلاس نرم‌افزار پروژه (بدون ذکر فیلدها و متدها)

پس از این مراحل، گام بعدی پیاده‌سازی و آزمون نرم‌افزار خواهد بود که در قسمت‌های بعدی به تفصیل به آن‌ها پرداخته می‌شود.

دلایل انتخاب این مدل فرآیندی، علاوه بر ثابت و مشخص بودن نیازهای پروژه در ابتدای امر عبارتند از:

- فهم این مدل نسبت به مدل‌های فرآیندی دیگر ساده‌تر است.

[102] Class Diagram

- از حیث تولید مستندات، شرایط بهتر و آسان‌تری دارد.
- مراحل به سادگی قابل بررسی و کنترل هستند.

۲.۶ شرح کلی مراحل پیاده‌سازی و ابزارهای مورد استفاده

در این مرحله، با توجه به طراحی انجام شده، نرم‌افزار پروژه پیاده‌سازی می‌شود. برای این منظور، ابتدا پس از رفع ابهام و بازنویسی گرامر زبان WL مطرح‌شده در مقاله اصلی پروژه، با استفاده از ابزارهای jflex و bison، دو بخش اصلی کامپایلر این زبان؛ یعنی lexer و parser، فراهم شدند. در lexer، تمامی کلمات کلیدی و عناصر مختلف زبان به صورت نشانه[۱۰۳]هایی در نظر گرفته شدند. سپس این نشانه‌ها، به parser داده می‌شود و با توجه به قواعد مختلف زبان، رفتار مربوط به هر قاعده در ذیل آن نوشته می‌شود. به این ترتیب اجزای زبان و قواعد گرامر آن پیاده‌سازی می‌شود. در این مرحله، خطاهای نحوی[۱۰۴] تشخیص داده می‌شود و در صورت بروز آن‌ها، به کاربر گزارش داده می‌شود. البته لازم به یادآوری است که بنا به نیازهای پیاده‌سازی، قسمت‌هایی از parser پس از تولید توسط ابزار bison، به صورت دستی تغییر پیدا کرده است، که این امر نیازمند تسلط کافی به جزئیات این کلاس است. در صورت نیاز، نحوه اجرای کدها و تولید آن‌ها توسط ابزارهای ذکر شده، در فایلی به نام README-GuidToRun.txt در پوشه پروژه آمده است. در کد نوشته شده برای parser، در زمان تشکیل درخت تجزیه و بررسی برنامه داده شده به آن، به طور همزمان گراف جریان کنترل برنامه نیز تولید می‌شود. گراف‌های مورد استفاده در این پروژه، همگی از نوع لیست پیوندی[۱۰۵] می‌باشند. دلیل استفاده از این ساختمان داده، سهولت در پیمایش، عدم نیاز به دسترسی تصادفی و رعایت حفظ ترتیب گره‌های فرزند و پدر است. در هر گره، اطلاعات مورد نیاز ذخیره می‌شود.

[103] Token

[104] Syntax Errors

[105] Linked List

تا این‌جا، کد برنامه داده شده به برنامه از نظر نحوی بررسی و گراف جریان کنترل ساخته شده است. اکنون با توجه به نوع درخواست کاربر؛ یعنی تولید گراف وابستگی برنامه، بازنویسی در حالت غیرحساس به پیشرفت یا در حالت حساس به پیشرفت، عملیات مربوط به هر کدام اجرا می‌شود.

برای نمایش گراف وابستگی برنامه، از ابزار قدرت‌مند GraphViz [۲۳] استفاده شده است. به این شکل که کد مربوط به این ابزار به پروژه اضافه شده است و با انجام تنظیمات اولیه، با تولید گراف به زبان dot که توسط این ابزار شناخته شده است، گراف مورد نظر در قالب یک تصویر با فرمت png. تولید می‌شود. در هنگام تولید گراف وابستگی برنامه، علاوه بر نمایش گرافیکی آن، گراف‌های وابستگی کنترلی و داده‌ای نیز به طور مجزا ذخیره می‌شوند.

اما علاوه بر ابزارهای ذکر شده در بالا، برای تابع تحلیل‌گر حلقه که در الگوریتم بازنویسی حالت حساس به پیشرفت نقش تأثیرگذاری را ایفا می‌کرد، ابزارهای مختلفی بررسی شد. گرچه هیچ‌کدام از ابزارهای بررسی‌شده، تحلیل مورد نیاز ما برای این تابع را ارائه نکردند، اما جستجو برای یافتن ابزار مناسب و نزدیک به خواسته ما، کار ساده‌ای نبود. برای این قسمت، مقالات مختلفی مطالعه شد و ابزارهای گوناگونی نظیر DRR، T2، Cooperative T2، Polyrank، Frama-C، Clang، Kittle، rankFinder، PAG، LoopFrog و AProVE [۲۴] نصب و بررسی شدند. در پایان این مرحله و با در نظر گرفتن معیارهای دقت و سرعت بالاتر، راحتی استفاده و نوع چاپ خروجی و میزان شباهت در تحلیل مورد نیاز این نرم‌افزار، از ابزار AProVE استفاده شده است. به این منوال که کد حلقه‌ای که به عنوان ورودی به تابع تحلیل‌گر حلقه داده می‌شود، به زبان C تبدیل می‌شود و سپس، برنامه تبدیل شده به زبان C به عنوان ورودی به ابزار تحلیل حلقه AProVE داده می‌شود. این ابزار با توجه به کد ورودی، یکی از سه جواب ممکن Proven، Disproven و Maybe را به سوال درباره خاتمه آن برنامه می‌دهد. Proven به معنای اثبات خاتمه برنامه و Disproven به معنای اثبات عدم خاتمه برنامه تحت هر شرایطی است. این در حالیست که نتیجه تحلیل Maybe به معنای ناتوانی این ابزار در تحلیل برنامه داده شده تلقی می‌شود. لذاست که نتیجه تحلیل Proven معادل با عبارت همواره درست خواهد بود، اما در صورتی‌که پاسخ یکی از حالت‌های Disproven یا Maybe باشد، بهتر است با اجرای الگوریتم‌هایی سعی در تحلیل حلقه داشته باشیم.

شایان ذکر است که به طور کلی، کدهای نوشته شده به زبان WL به زبان‌های سطح بالا و رایج‌تری مثل C قابل تبدیل است. در این نرم‌افزار نیز می‌توان کد ورودی و کدهای بازنویسی‌شده در هر حالت را در قالب برنامه‌های به زبان C نیز مشاهده کرد.

توضیحات جزئیات پیاده‌سازی کلاس‌های مختلف نرم‌افزار و گزارش روند انجام کار، در فایل‌های جداگانه‌ای در پوشه پروژه موجود است.

۳.۶ ایجاد رابط کاربری گرافیکی[۱۰۶]

اهمیت ظاهر برنامه و صفحاتی که کاربر توسط آن‌ها با سیستم در تعامل است، بر کسی پوشیده نیست. این اهمیت درباره نرم‌افزارهای مورد استفاده توسط کاربران حرفه‌ای رایانه یا همان برنامه‌نویسان که کاربران اصلی این نرم‌افزار هستند، دوچندان می‌شود. چرا که طراح باید پیچیدگی‌ها را در رابط کاربری به حداقل برساند، به نحوی که قابلیت‌های برنامه کاهش نیابد. از این رو، برای طراحی رابط کاربری گرافیکی این برنامه زمان زیادی صرف شده است. ابتدا با مشورت از یکی از متخصصان زیبایی‌شناسی، طراحی کلی صفحه برنامه انجام شد. پس از آن، طرح‌های مختلفی ارائه شد و به عنوان آزمایش، در اختیار تعدادی از برنامه‌نویسان قرار گرفت تا بازخورد آن‌ها نسبت به رابط کاربری این برنامه سنجیده شود. در پایان، با انجام اصلاحات، رابط کاربری گرافیکی برنامه نهایی شد.

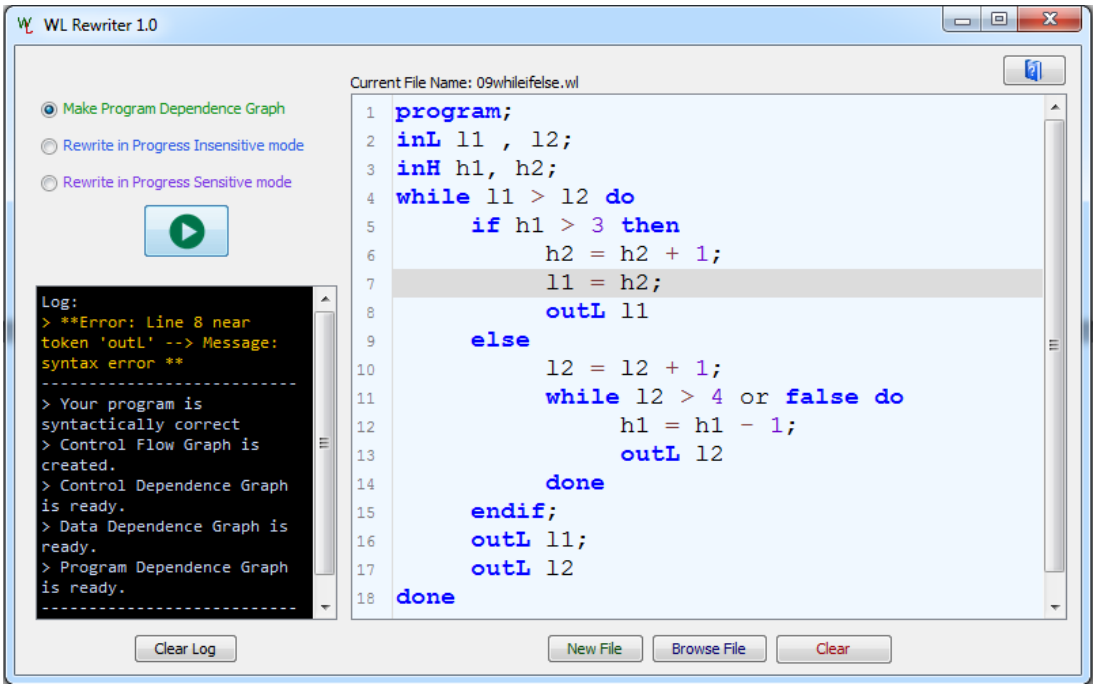
نکات زیر برای طراحی رابط کاربری این برنامه مورد استفاده قرار گرفته شده است:

- گزینه‌ها و دکمه‌های موجود در صفحه باید همگون و با سبک یکسان باشند.
- در هنگام تغییر وضعیت برنامه، باید ظاهر نیز متناسب با آن تغییر یابد. یعنی برنامه متناسب با هر فعالیت، بازخورد مناسبی داشته باشد.
- هر گزینه باید کاملاً واضح و دارای معنای خاص باشد.
- برای همگی فعالیت‌ها، حالت‌های پیش‌فرض در نظر گرفته شود.
- کاربر نیازی به آموزش برای یادگیری کار با رابط کاربری نداشته باشد یا حداقل باشد.

[106] Graphical User Interface

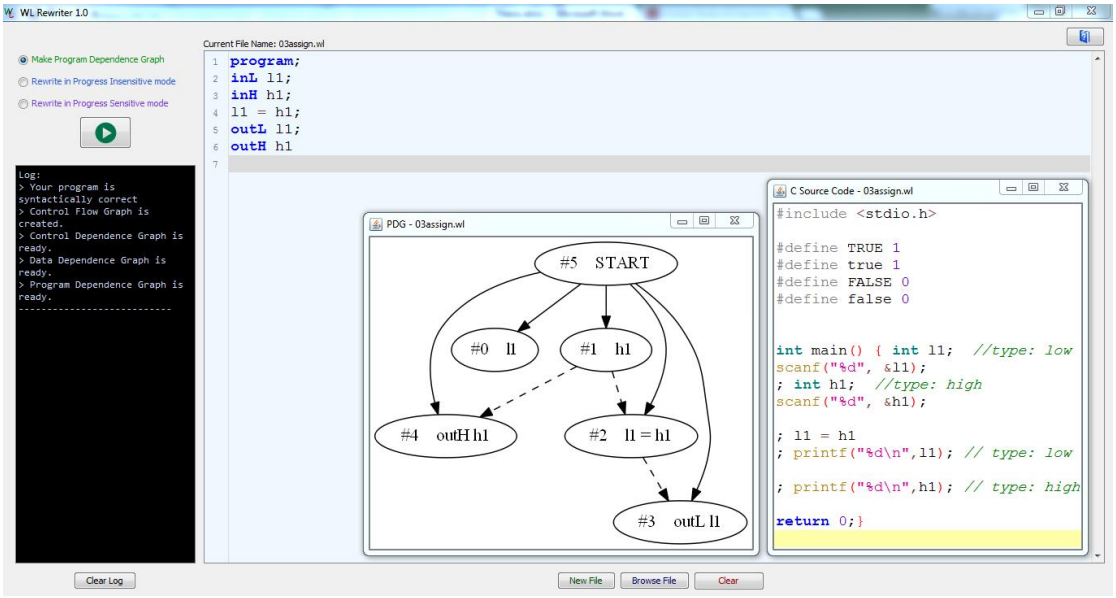
- اجزائی که با یکدیگر مرتبط هستند، در یک گروه‌بندی خاص باشند.
- از رنگ‌ها و سبک‌ها به درستی و با توجه به گروه‌بندی‌ها و معانی رنگ‌ها در ذهن کاربر با توجه به سابقه قبلی آن‌ها استفاده شود.
- برای گزینه‌ها، از میان‌برها و یادمان[۱۰۷]ها استفاده شود.
- برای حذف یا پاک کردن اطلاعات مهم، تأیید مجدد کاربر دریافت شود.
- برای نمایش پیغام‌ها از رنگ‌های متناسب استفاده شود.
- به دلیل استفاده طولانی مدت کاربر از این نرم‌افزار، بهتر است از رنگ‌ها و چینشی استفاده شود که آلودگی بصری برای کاربر را به دنبال نداشته باشد.
- امکان تغییر ابعاد صفحه برای کاربر وجود داشته باشد و ضمناً با تغییر ابعاد پنجره برنامه، چینش اجزا در صفحه منظم باقی بماند.

در این پروژه سعی شده است تا موارد بالا تا حد امکان رعایت شوند و تجربه خوب و لذت‌بخشی را برای کاربر به ارمغان بیاورد. به طور مثال، یک ویرایشگر کد سفارشی‌شده به اجزای زبان WL به رابط کاربری اضافه شده است. در ادامه تصاویری از رابط کاربری برنامه را مشاهده می‌کنید.



شکل ۲۰ - نمای کلی رابط کاربری گرافیکی نرم‌افزار

[107] Mnemonic



شکل ۲۱ – نمونه‌ای از اجرای برنامه در رابط کاربری گرافیکی نرم‌افزار

۴.۶ راستی‌آزمایی و آزمون

همان‌طور که قبلاً ذکر شد، روش مطرح شده و الگوریتم‌های بازنویسی با توجه به بیان صوری انجام شده در مقاله اصلی پروژه [۲]، از نظر سلامت و شفافیت قابل اثبات است. اما راستی‌آزمایی و آزمون برنامه پیاده‌سازی شده نیز اهمیت دارد. برای این کار، با بهره‌گیری از آزمون دامنه[۱۰۸] تعدادی موردآزمون[۱۰۹] برای بررسی صحت اجرای برنامه پیاده‌سازی شده طراحی شد. روش آزمون دامنه یکی از روش‌های پرکاربرد در آزمون نرم‌افزار به شمار می‌رود. در این روش، تعداد محدودی موردآزمون که هر یک به عنوان نماینده‌ای از دسته موردآزمون‌های مشابه هستند، به عنوان ورودی به نرم‌افزار داده می‌شود و خروجی حاصل از پردازش نرم‌افزار بر روی داده ورودی بررسی و راستی‌آزمایی می‌شود. در این پروژه نیز با همین روش، تعداد نزدیک به سی موردآزمون بررسی شد که هر یک شامل ساختار متفاوتی از

[108] Domain Testing

[109] Test Case

عناصر موجود در زبان WL می‌باشند. با توجه به این‌که در حوزه زبان‌های برنامه‌سازی، استقرا از مرسوم‌ترین روش‌های اثبات به شمار می‌رود، سعی شد تا با کمترین تعداد استفاده از عناصر زبان در هر برنامه، نرم‌افزار مورد آزمون و بررسی قرار گیرد و برنامه‌های مشابه یا دارای ساختار مشابه با برنامه‌های موردآزمون، به استقرا آزمون‌شده بگیریم. از طرفی، در طراحی مواردآزمون سعی شد تا انواع مختلف جریان‌های صریح و ضمنی مدنظر در خط مشی عدم‌تداخل در هر دو حالت حساس و غیرحساس به پیشرفت مورد بررسی قرار بگیرد.

در شکل شماره ۲۲، به عنوان نمونه، یکی از مواردآزمون آورده شده است.

```
program;
inL l1 , l2;
inH h1, h2;
while l1 > l2 do
        if h1 > 3 then
                h2 = h2 + 1;
                l1 = h2;
                outL l1
        else
                l2 = l2 + 1;
                while l2 > 4 or false
do
                        h1 = h1 - 1;
                        outL l2
                done
        endif;
        outL l1;
        outL l2
done
```

شکل ۲۲- نمونه‌ای از موردآزمون‌های بررسی‌شده

۷

فصل هفتم

جمع‌بندی و کارهای آینده

جمع‌بندی و کارهای آینده

در طول فصول گذشته، ابتدا درباره امنیت و خط مشی امنیتی صحبت شد. سپس خط مشی امنیتی عدم تداخل را به عنوان یکی از خطوط مشی که خاصیت نیستند، معرفی کردیم و اشاره‌ای به محدودیت‌هایی که این‌گونه خط مشی‌ها برای اعمال دارند، شد. در ادامه انواع مختلف عدم تداخل؛ یعنی حالت غیرحساس به پیشرفت و حساس به پیشرفت را مطرح کردیم و مشاهده شد که برای اعمال این خط مشی، راه‌های مختلفی وجود دارد که یکی از بهترین مکانیزم‌ها، روش بازنویسی برنامه است. در حالت غیرحساس به پیشرفت، مسیرهایی اهمیت داشت که از مقادیر ورودی سطح بالا آغاز و به دستورات خروجی مقادیر سطح پایین ختم می‌شدند. در حالت حساس به پیشرفت، وضعیت پیشرفت برنامه نیز ممکن بود اطلاعات سطح بالایی را به مشاهده‌گر سطح پایین منتقل کند. از این رو، نگاه ویژه‌ای به ساختار ایجاد واگرایی در برنامه‌ها داشتیم. سپس زبان برنامه‌نویسیِ مدل ارائه‌شده در مقاله اصلی پروژه یا همان WL شرح داده شد. این زبان شامل ساختارهای مختلف و مرسوم زبان‌های برنامه‌نویسی بود که عنصر ایجاد حلقه در آن، ساختار while است. با توجه به تعاریف ارائه شده، الگوریتم‌های بازنویسی برای حالت‌های غیرحساس و حساس به پیشرفت بیان شد و همان‌طور که قبلاً اشاره شده بود، بیان صوری خط مشی‌ها و اثبات سلامت و شفافیت الگوریتم‌های بازنویسی، در مقاله اصلی پروژه آمده است. در ادامه به نحوه پیاده‌سازی و ابزارهای مورد استفاده پرداخته شد و با ارائه مورد-آزمون‌های کاربردی، صحت برنامه پیاده‌سازی شده را نشان دادیم.

اما ایده این پروژه یکی از گام‌های ابتدایی و رو به جلویی برای اعمال خط مشی‌های امنیتی با حفظ شفافیت است. می‌توان فکر اصلی استفاده شده در این پروژه را برای زبان‌های برنامه‌نویسی پیشرفته‌تر و رایج‌تر که ساختارهای زبانی پیچیده‌تری دارند به کار بست. زبان‌هایی که از ساختارهای کلاس، شی، چندنخی و سایر ویژگی‌های نوین زبان‌های برنامه‌نویسی امروزی پشتیبانی می‌کنند، می‌توانند به عنوان آینده این پروژه قلمداد کرد. در پیاده‌سازی نیز می‌توان با بهینه‌سازی کد برنامه، به سرعت و استفاده کمتر از حافظه کمک کرد. تابع تحلیل‌گر حلقه نیز به تنهایی می‌تواند موضوع پژوهش جذابی برای علاقمندان این حوزه باشد. از حیث پژوهش‌های نظری نیز می‌توان به دسته‌بندی خط مشی‌های قابل اعمال توسط روش بازنویسی به عنوان یکی از چالشی‌ترین مسائل روز نام برد.

منابع و مراجع

Amirkabir University of Technology
(Tehran Polytechnic)

Computer and Information Technology Engineering Department

B.Sc. Thesis

Title

Design and Implementation of a Tool for Rewriting-Based Enforcement of Noninterference in Programs

By
Seyed Mohammad Mehdi Ahmadpanah

Supervisor
Dr. Mehran S. Fallah

September 2015