باسمه تعالی

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

فرم تعریف **پروژه**
فارغ التحصیلی دوره کارشناسی

دانشکده مهندسی کامپیوتر
و فناوری اطلاعات

تاریخ: .....
شماره: .....

عنوان پروژه: طراحی و پیاده‌سازی ابزاری به منظور اِعمال خط مشی امنیتیِ عدم تداخل مبتنی بر روش بازنویسی برنامه
استاد راهنمای پروژه: دکتر مهران سلیمان‌فلاح **امضاء:**
مشخصات دانشجو: **نام و نام خانوادگی: سید محمدمهدی احمدپناه**[1] **گرایش: نرم‌افزار** **شماره دانشجویی: ۹۰۳۱۸۰۶** **ترم ثبت نام پروژه: دوم ۹۳-۹۴**
داوران پروژه: **۱-** **امضاء داور:** **۲-** **امضاء داور:**
شرح پروژه (در صورت مشترک بودن بخشی از کار که بعهده دانشجو می باشد مشخص شود): **به پیوست آمده است.**
وسائل مورد نیاز: **- امکان دسترسی به مقالات مرتبط** **- یک دستگاه کامپیوتر دارای دسترسی به اینترنت**
محل انجام پروژه: دانشکده مهندسی کامپیوتر و فناوری اطلاعات دانشگاه صنعتی امیرکبیر **تاریخ شروع: اردیبهشت ۱۳۹۴**

این قسمت توسط دانشکده تکمیل می‌گردد:

تاریخ تصویب در گروه: **اسم و امضاء:** **تاریخ تصویب در دانشکده:** **اسم و امضاء:** **اصلاحات لازم در تعریف پروژه:**

توجه: پروژه حداکثر یک‌ماه و نیم پس از شروع ترمی که در آن در درس پروژه ثبت نام به عمل آمده است باید به تصویب برسد.

نسخه۱- دانشکده	نسخه ۲- استاد راهنما	نسخه ۳- دانشجو

[1] Email: smahmadpanah@aut.ac.ir

تعریف مسئله:

امروزه و با گسترش سیستم‌ها و نرم‌افزارها، امنیت انتقال اطلاعات در برنامه‌های نوشته شده به زبان‌های برنامه‌نویسی گوناگون، بیش از پیش اهمیت پیدا کرده است. از این رو، می‌توان با افزودن ابزارهایی به یک زبان برنامه‌نویسی، باعث تولید نرم‌افزارهای مقاوم‌تری در برابر حملات و نفوذها شد. لذا باید مفهوم امن بودنِ یک سیستم یا برنامه به طور دقیق تعریف و مشخص شود که همین مسئله، چالشی برای متخصصان این حوزه است. در مرحله بعدی، نحوه و رویکرد اِعمال آن مفهوم از امنیت مهم خواهد بود که وابستگی زیادی به تعریف ارائه شده دارد.

به طور کلی، خط مشی[۲] 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.

یک خط مشی امنیتی را می‌توان به عنوان یک زیرمجموعه از مجموعه‌ی توانی همه اجراها، که هر اجرا یک دنباله دلخواه از حالت[۳]ها است، تعریف کرد. ضمناً می‌توان آن را به عنوان مجموعه‌ی برنامه‌هایی که آن خط مشی را برآورده می‌کنند، در نظر گرفت. بعضی از خط مشی‌های امنیتی، *خاصیت*[۴] هستند؛ به‌خاطر این‌که قابل دسته‌بندی و تشخیص توسط مجموعه اجراهای جداگانه می‌باشند. برخی از نیازمندی‌های مهم امنیتی، خاصیت نیستند. یک نمونه مهم از این‌گونه نیازمندی‌ها، خطِ مشیِ عدم تداخل[۵] است. نکته حائز اهمیت این است که روش اِعمال خاصیت‌ها با نحوه اعمال خط مشی‌هایی که خاصیت نیستند، متفاوت است.

به زبان ساده، خط مشی عدم تداخل بیان می‌کند که یک مشاهده‌گرِ[۶] 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.

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. 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*[۷] و *غیر حساس به پیشرفت*[۸] 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*[۹] برنامه نیز دسترسی دارد. این باعث می‌شود تا بتواند تفاوت بین واگرایی[۱۰] برنامه با موقعیتی که برنامه خاتمه می‌یابد یا در حال محاسبه مقادیر قابل‌مشاهده‌ی بعدی است، را تمیز دهد.

نکته مهم دیگر، بررسی جریان‌های غیرمجاز صریح و ضمنی است که می‌تواند مسئله‌ی داده شده را پیچیده کند.

---

[2] Policy
[3] State
[4] Property
[5] Noninterference
[6] Observer
[7] Progress-Sensitive
[8] Progress-Insensitive
[9] Progress Status
[10] Divergence

در این پروژه، هدف این است که با بهره‌گیری از روش بازنویسی برنامه[11]، برنامه‌های نوشته‌شده توسط یک زبان برنامه‌نویسیِ مدل به نام WL به نحوی تغییر داده شوند که خط مشی امنیتیِ عدم تداخل را برآورده سازند.

راه حل‌های فعلی و مشکلات آن‌ها:

وِنکَتَکریشنَن و دیگران یک روش تبدیل برنامه‌ی ترکیبی برای اِعمال عدم تداخل ارائه داده‌اند. برنامه تغییر داده شده، سطوح امنیتی انتساب[12] را دنبال می‌کند و زمانی‌که یک جریان غیرمجاز در حال وقوع باشد، خاتمه می‌یابد. این روش، تنها در فرمول‌بندی‌هایی از عدم تداخل که بدون توجه به رفتار خاتمه‌ی برنامه‌ها مطرح می‌شود، قابل استفاده است.

مَگزینیوس و دیگران یک چارچوب برای ناظر[13]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 Perl و JavaScript 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 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[14] را نمی‌دهد. سانتوس و رِزک 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. این موضوع باعث می‌شود که پروژه‌هایی که محدودیت‌های نَحوی[15] بر روی کد دارند، از اطلاعات ایستا در ناظری بر اجراهای چندگانه‌ی برنامه‌ها استفاده کنند.

بِلو و بونِلی یک ناظرِ اجرایی[16] 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 دیگران یک ماشین طراحی کرده‌اند که رخدادهای انتزاعی[17] 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.

به طور کلی، مسئله‌ی تشخیص برنامه‌هایی که عدم تداخل را برآورده می‌کنند، تصمیم‌ناپذیر[18] است. پس در حالت کلی، عدم تداخل توسط روش‌های ایستا قابل اعمال نیست؛ به همین دلیل است که نوع‌سامانه[19]های ارائه‌شده برای این مسئله، محافظه‌کارانه[20] هستند و ممکن است بعضی برنامه‌های امن را نیز رد کنند. از طرفی، این مسئله هم‌بازگشتی شمارش‌پذیر[21] نیز نیست. بنابراین، قابل اِعمال توسط ناظرهای اجرایی که نقضِ خط مشی عدم تداخل در یک برنامه‌ی در حال اجرا را بررسی می‌کنند، نیست.

---

[11] Program Rewriting
[12] Assignment
[13] Monitor
[14] Termination Channels
[15] Syntactic
[16] Execution Monitor
[17] Abstract Events
[18] Undecidable
[19] Type System
[20] Conservative
[21] Co-recursively Enumerable

با توجه به مشکلات اخیر مطرح‌شده در روش‌های قبلی، روش بازنویسیِ برنامه تغییراتی را به ذاتِ عدم تداخل وارد نمی‌کند؛ بلکه به جای آن، یک برنامه جدید با حداقل تغییرات ممکن نسبت به برنامه اصلی، که عدم تداخل را برآورده می‌کند، تولید می‌کند. در واقع، می‌توان روش بازنویسی برنامه را روشی بین روش‌های ایستا و روش‌های پویا دانست.

راه حل پیشنهادی:

راه حل مورد توجه در این پروژه، از گراف‌های وابستگی برنامه[۲۲] استفاده می‌کند. گراف وابستگی برنامه، ابزاری است که توسط آن، یک برنامه با یک گراف جهت‌دار که گره‌های آن گزاره‌ها یا عبارت‌های برنامه است و یال‌های آن، وابستگی‌های کنترلی یا داده‌ای بین گره‌ها را نشان می‌دهد، بیان می‌شود. گراف وابستگی یک برنامه، همه وابستگی‌های بین گزاره‌های آن برنامه را منعکس می‌کند. البته ممکن است مسیری بین گره‌ها در این گراف وجود داشته باشد ولی جریانی بین آن دو برقرار نباشد.

برخلاف سایر روش‌های امنیتی مطرح‌شده، روش این پروژه، یک برنامه مغایر با خط مشی موردنظر را، چه قبل یا چه در طول زمان اجرا، رد نمی‌کند؛ بلکه آن‌ها بازنویسی می‌شوند و از برنامه‌های ناامن به برنامه‌های امن تبدیل می‌شوند. در این روش، هر دو نوع عدم تداخل- حساس به پیشرفت و غیر حساس به پیشرفت- در برنامه‌ها به همراه مقادیر قابل‌مشاهده میانی، مورد توجه هستند. ضمناً قابل اثبات است که بازنویس[۲۳]های مورد نظر، در اعمال عدم تداخل، سالم[۲۴] و شفاف[۲۵] هستند. شفاف بودن یک روش بدین معناست که تا حد ممکن، مجموعه اجراهای ممکنِ برنامه‌ی تبدیل‌شده مشابه برنامه‌ی ورودی باشد، خواه امن باشد یا نباشد.

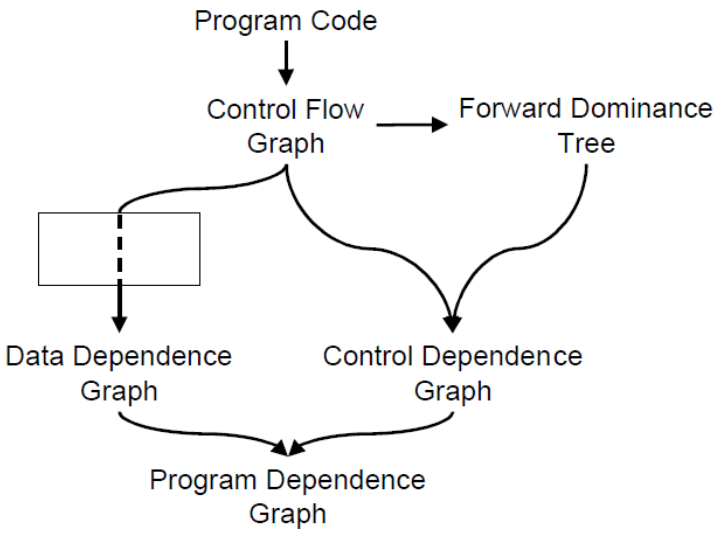
توصیف نرم‌افزار:

ورودی: کد برنامه (که ممکن است خط مشی عدم تداخل را برآورده نکند)

خروجی: کد برنامه تغییر داده شده (که خط مشی مورد نظر را برآورده می‌کند)

پردازش: از روی کد برنامه ورودی، گراف وابستگی برنامه ساخته می‌شود. سپس با توجه به این‌که خط مشی مورد نظر حساس به پیشرفت یا غیر حساس به پیشرفت است، پردازش مربوط به هر کدام انجام می‌شود.

برای ساخت گراف وابستگی برنامه، باید گراف‌های وابستگی کنترل و وابستگی داده را از روی کد برنامه تشکیل داد. گراف وابستگی برنامه از گراف جریان کنترل به دست می‌آید. با تشخیص وابستگی‌های کنترل و داده، گراف جریان کنترل به گراف وابستگی برنامه تبدیل می‌شود.



شکل ۱- نمودار کلی نحوه به دست آوردن گراف وابستگی برنامه

[22] Program Dependence Graph (PDG)
[23] Rewriter
[24] Sound
[25] Transparent

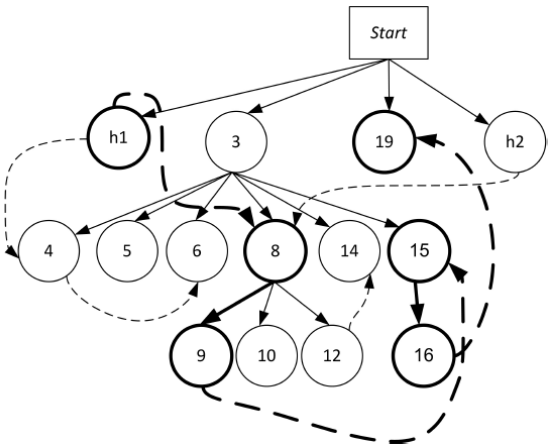
گراف جریان کنترل گرافی است که گره‌های آن، دستورهای برنامه و جریان‌های کنترلی بین گره‌ها نمایش داده می‌شود. دو گره شروع و پایان نیز نقاط ورود و خروج برنامه را مشخص می‌کند.

وابستگی داده بین گره الف به ب یعنی گره ب شامل متغیری است که در الف منتسب شده است. وابستگی کنترل بین گره الف به ب یعنی اجرای ب توسط مقداری که در الف محاسبه شده است، کنترل می‌شود. بنابراین، هر مسیر در گراف وابستگی برنامه ممکن است جریانی بین دو گره باشد. در گراف وابستگی برنامه، می‌گوییم جریانی از الف به ب وجود دارد، اگر مقدار محاسبه شده در ب یا صرف اجرای ب، وابستگی به مقدار محاسبه شده در الف داشته باشد.

با استفاده از درخت غلبه رو به جلو[۲۶]، وابستگی کنترلی ب به الف این گونه تعریف می‌شود که یک مسیر در گراف جریان کنترل از الف به ب وجود دارد که حاوی غلبه‌کننده رو به جلوی بلافصل[۲۷] نیست.

تعاریف و الگوریتم‌های مبتنی بر گراف جریان کنترل نیازمند وجود یک گراف هم‌بند است که در این زبان مدل، به دلیل عدم وجود تابع یا رویه، مشکلی ایجاد نخواهد کرد.

```
1.  in_H h1, h2;
2.  in_L l1, l2, l3;
3.  if (l1 ≤ 0) then
4.    l2 = h1;
5.    out_L l1;
6.    out_L l2;
7.  else
8.    if(h2 == h1) then
9.      l3 = 0;
10.     out_L l1;
11.   else
12.     l1 = 1;
13.   endif;
14.  out_L l1;
15.  if(l3 == 0) then
16.    l3 = 1;
17.  endif;
18. endif;
19.  out_L l3;
```



شکل ۲- نمونه کدی به زبان WL و گراف وابستگی برنامه آن

در شکل شماره ۲، نمونه کدی به زبان مدل مورد نظر و همچنین، گراف وابستگی برنامه متناظر با آن وجود دارد. در این گراف، یال‌های با خطوط ممتد، بیانگر وابستگی‌های کنترل و یال‌های خط‌چین، بیانگر وابستگی‌های داده در برنامه هستند. البته به دلیل بررسی خط مشی عدم تداخل، وابستگی‌های داده‌ای که از ورودی‌های سطح پایین ناشی می‌شوند، در شکل نیامده است. به عنوان مثال، یال‌های پررنگ شده نشان دهنده مسیر از ورودی سطح بالای h_1 تا دستور خروجی متغیر l_3 در خط پایانی برنامه است.

در عدم تداخل غیر حساس به پیشرفت، از روی گراف وابستگی برنامه، مسیرهای حاوی ورودی سطح بالا که در ادامه آن‌ها خروجی سطح پایینی نمایش داده خواهد شد، مورد بررسی قرار می‌گیرند. با استفاده از شرط‌های اجرا[۲۸] و شرط‌های مسیر[۲۹]، بسته به وجود جریان صریح یا ضمنی، دستور نمایش یک مقدار سطح پایین، به عبارتی شرطی تبدیل می‌شود که در صورت برقراری آن شرط، مقدار خروجی نشان داده می‌شود و در غیر این صورت، آن مقدار قابل مشاهده برای مشاهده‌گر سطح پایین نخواهد بود.

[26] Forward Dominance Tree
[27] Immediate Forward Dominator
[28] Execution Conditions
[29] Path Conditions

در عدم تداخل حساس به پیشرفت، به دلیل این‌که مشاهده‌گر سطح پایین دسترسی‌های بیشتری دارد، لذا رفتار برنامه نیز اهمیت بیشتری پیدا می‌کند؛ یعنی باید بازنویس به گونه‌ای کد برنامه را تغییر دهد که وضعیت پیشرفت در برنامه‌ی تغییر پیدا کرده، به مقادیر سطح بالا وابستگی نداشته باشد. به همین دلیل، واگرایی یا خاتمه برنامه از دیدِ مشاهده‌گر سطحِ پایین نیز حاوی اطلاعاتی است که باعث نقضِ خط مشی عدم تداخل در این حالت می‌شود. در این حالت، روش پردازش و بازنویسی برنامه، به یک تحلیل‌گرِ حلقه[30] وابسته خواهد شد. در صورتی که بتوان تحلیل‌گر قوی و مناسبی برای این کار پیدا شود یا پیاده‌سازی شود، می‌توان الگوریتم در حالت غیرحساس به پیشرفت را پیاده‌سازی کرد.

این تحلیل‌گر تابعی است که با وارسی نحوی متن برنامه حاوی حلقه، مقدار منطقی درست برمی‌گرداند، اگر حلقه همواره خاتمه یابد، مقدار منطقی غلط برمی‌گرداند، اگر حلقه همواره واگرا باشد، و یک عبارت بولی برمی‌گرداند که آن عبارت در حالتی برآورده می‌شود که حلقه مورد نظر حتما خاتمه یابد. برای ساخت یک تحلیل‌گر حلقه، الگوهای رایج در برنامه‌ها را که مشاهده آن‌ها در یک برنامه، به ما این اطمینان را می‌دهد که یک حلقه همواره خاتمه می‌یابد، همواره واگراست و یا می‌توانیم برای آن یک عبارت بولی ساخته که ویژگی‌هایی که پیشتر بیان کردیم را دارا باشد.

در ابتدا، مراحل حالت غیر حساس به پیشرفت بر روی کد برنامه ورودی انجام می‌شود و در صورتی که برنامه دارای مسیرهایی باشد که در آن قبل از رسیدن به یک حلقه، مقدار ورودی سطح بالایی وجود داشته باشد، مطابق با نتیجه‌ی تحلیل‌گر حلقه در هر کدام از حلقه‌ها، کدهای مربوط به آن بازنویسی و اصلاح می‌شوند. این‌گونه کد برنامه خروجی نیز خط مشی عدم تداخل حساس به پیشرفت را برآورده می‌سازد.

ضمناً می‌توان با استفاده از ابزارهای موجود مانند lex و yacc، کدهای به زبان WL را به کد میانیِ به زبان C تبدیل کرد.

همچنین برای آشنایی بیشتر کاربر با نرم‌افزار و الگوریتم‌ها، بخش راهنمای استفاده از برنامه تهیه خواهد شد.

روش راستی‌آزمایی: با استفاده از گراف جریان کنترل برنامه بازنویس، موارد آزمونی برای برنامه خود تولید می‌کنیم. موارد آزمون برای برنامه ما، برنامه‌هایی در زبان مدل WL است. همچنین، تعدادی موارد آزمون ثابت برای راستی‌آزمایی برنامه‌های بازنویسی شده تهیه خواهیم کرد. گراف جریان کنترل نشان دهنده جریان کنترل بین دستورالعمل‌های یک برنامه است اما از آن‌جایی که پیش‌بینی می‌شود برنامه بازنویسی کننده برنامه بزرگی خواهد بود، استخراج گراف جریان کنترل مقرون به صرفه نخواهد بود. برای این منظور، دنباله‌ای از دستورالعمل‌های برنامه خود را به عنوان درشت‌دستورالعمل و آن را به عنوان گره‌ای از گراف جریان کنترل برنامه در نظر می‌گیریم. در این گراف، گره الف به گره ب با یک یال جهت‌دار متصل می‌شود، اگر و تنها اگر آخرین دستورالعمل بلوک دستورات متناظر با گره الف ممکن است قبل از اولین دستورالعمل بلوک دستورات متناظر با گره ب اجرا شود.

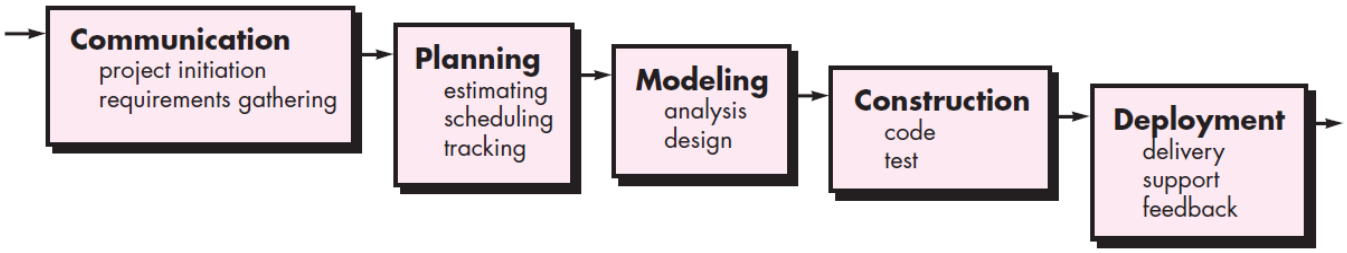
بعد از به دست آوردن گراف جریان کنترل برای برنامه خود، با محاسبه پیچیدگی حلقوی[31]، درصدی از مسیرهای مستقل را شناسایی نموده و برای آن‌ها موارد آزمونی تهیه می‌کنیم. سپس به ازای هر مورد آزمون، موارد آزمون ثابت خود را بر روی برنامه‌های بازنویسی شده (به وسیله ردیابی برنامه‌ها) می‌آزماییم.

موارد آزمون ثابت بهتر است به گونه‌ای طراحی شوند که ادعای برآورده‌سازی عدم تداخل در برنامه‌های بازنویسی شده را به چالش بکشد.

[30] Loop Analyzer
[31] Cyclomatic Complexity

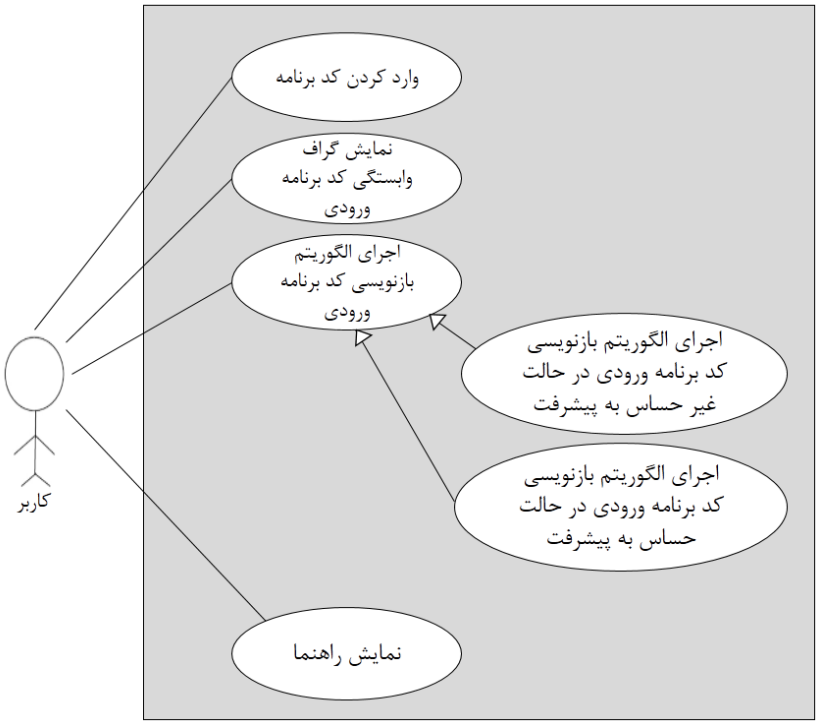
تحلیل نرم‌افزار:

با توجه به مشخص و ثابت بودن نیازهای این نرم‌افزار در ابتدای تعریف پروژه، می‌توان از مدل فرآیندی آبشاری[۳۲] یا چرخه حیات کلاسیک[۳۳] استفاده کرد. این مدل فرآیندی شامل پنج مرحله ارتباط[۳۴]، برنامه‌ریزی[۳۵]، مدل‌سازی[۳۶]، ساخت[۳۷] و استقرار[۳۸] است.



شکل ۳ - مدل فرآیندی آبشاری

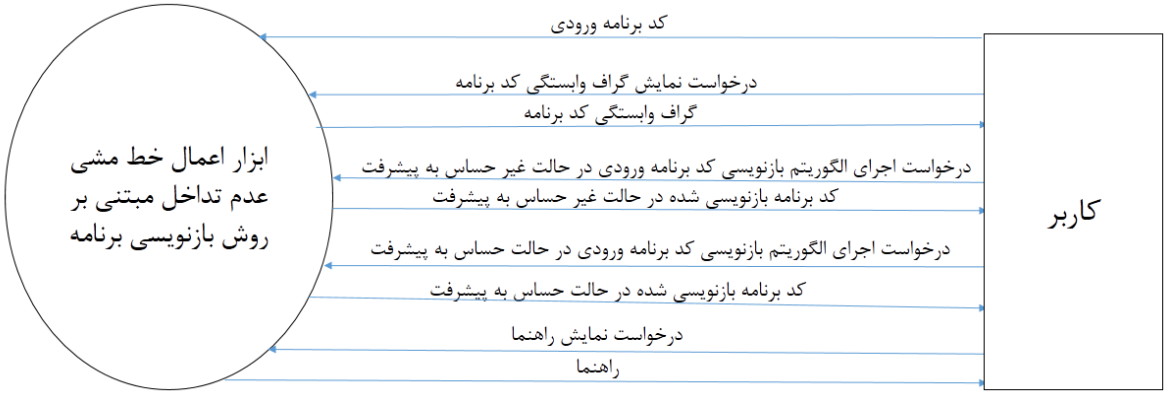
نمودار Use Case:



شکل ۴ - نمودار Use Case

[32] Waterfall
[33] Classic Life Cycle
[34] Communication
[35] Planning
[36] Modeling
[37] Construction
[38] Deployment

نمودار Context:



شکل ۵ - نمودار Context

برای پیاده‌سازی این ابزار می‌توان از زبان‌های سطح بالای برنامه‌نویسی نظیر جاوا و محیط‌های توسعه مانند نت‌بینز[۳۹] بهره برد.

مراجع:

مقالات مورد استفاده در این پروژه:

[1] A. Lamei and M.S. Fallah, “Rewriting-based Enforcement of Noninterference in Programs with Observable Intermediate Values”, To appear in the Journal of Universal Computer Science, Vol. XX, No. XX, Month 20XX, 1-24.

[2] D. Wasserrab, D.Lohner, and G. Snelting, “On PDG-based noninterference and its modular proof”, in Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS ’09, ACM, 2009, pp. 31-44.

[3] C. Hammer and G. Snelting, “Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs”, the International Journal of Information Security, Volume 8, Issue 6, 2009, pp 399-422

[4] P. Anderson, T. Reps, and T. Teitelbaum, “Design and Implementation of a Fine-Grained Software Inspection Tool”, IEEE Transactions on Software Engineering, Vol. 29, No. 8, 2003, pp. 721-733.

[5] J. Ferrante, K.J. Ottenstein, and J.D. Warren, “The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9, 1987, pp. 319-349.

[6] F. Nielson, H. R. Nielson, and C. Hankin, “Principles of Program Analysis”, 2nd Ed., Springer Science & Business Media, 2005.

[39] NetBeans IDE