
بسم الله الرحمن الرحيم

دانشگاه صنعتی اصفهان

دانشکده مهندسی برق و کامپیوتر

تبدیل خودکار کد منبع سیستم عامل از زبان C به زبان مدلسازی Promela جهت استفاده برای تست و ارزیابی خودکار مدل

پروژه کارشناسی دانشکده کامپیوتر – نرم افزار

مرضیه علیدادی

استاد راهنما

دکتر الهام محمودزاده

۱۴۰۰

فهرست مطالب

فصل اول

مقدمه

امروزه اینترنت اشیاء[1] و سیستم‌های رایافیزیکی[2] در مسائل روزمره بسیار کاربرد دارند. از جمله کاربردهای این سیستم‌ها در زندگی روزمره، هوشمندسازی لوازم خانگی و ساعت‌های هوشمند است. بنابراین، باتوجه به اینکه این سیستم‌ها کارهای حساسی را انجام می‌دهند، اطمینان از درستی عملکرد آن‌ها بسیار مهم است. همچنین، اطمینان از درستی عملکرد مواردی مانند چگونگی مدیریت وقفه‌ها[3] ، محدودیت حافظه و سبک وزن بودن عملیات، بسیار مهم است؛ چرا که مسائلی مانند کمبود حافظه و منابع در دستگاه‌های کم مصرف، جنبه‌های قابل اطمینان در سیستم‌عامل‌های این دستگاه ها را دچار چالش می‌کند. بنابراین، لازم است عملکرد آن ها به صورت رسمی مورد تست و ارزیابی قرار گیرد[۷]. یکی از فرایند های حائز اهمیت در این تست و وارسی رسمی، تبدیل و مدل‌سازی کد منبع سیستم عامل از زبان C به Promela ، به عنوان یک زبان مدل‌سازی؛ و بالعکس، تبدیل از زبان مدل‌سازی، به کد منبع سیستم عامل است. با توجه به اینکه این تبدیل، عملیاتی طاقت فرساست، به رویکردی خودکار برای انجام آن نیاز است.

با توجه به اهمیت بیان شده برای مدل‌سازی و وارسی رسمی که با استفاده از زبان مدل‌سازی Promela صورت می‌گیرد، در این پژوهش سعی شده است تا با بررسی قوانین استخراج شده از ساختار موجود در دو زبان Promela و C ، ساختار و نحوه ی تبدیل خودکار کد منبع سیستم عامل از زبان C به Promela ، به عنوان یک زبان مدل‌سازی ابداع شود.

۱ - ۱ پیشینه تحقیق

در ابتدا بررسی هایی در جهت شناسایی ابزار های[4] موجود احتمالی که عملیات تولید مدل را به صورت خودکار انجام می‌دادند، صورت گرفت. طبق بررسی های صورت گرفته، ابزار هایی که به این منظور و برای تولید خودکار

[1] IOT
[2] Cyber-Physical systems
[3] interrupts
[4] tools

مدل در دسترس است، دارای نقاط ضعف قابل توجهی هستند. درنتیجه طی این پژوهش به بررسی این نقاط ضغف و امکان رفع آن ها پرداخته‌شد.

۱-۲ اهداف و دستاوردهای تحقیق

با توجه به بررسی های صورت گرفته، مشخص شد که کار بر روی ابزار های موجود و رفع نقاط ضعف و کاستی های آن ها، به صرفه تر از تولید ابزار جدید خواهدبود. بنابراین، در ادامه، به مقایسه ی عملکرد این ابزار ها و استخراج نقاط ضعف تولید کد در هر یک از آن ها و ارائه ی راه حل برای رفع کاستی های آن ها پرداخته‌شد.

۱-۳ ساختار گزارش

پیش از بیان شرح اصلی این پژوهش، شناخت برخی مفاهیم و ابزار ها ضرورت دارد. بنابراین، در فصل دوم به بیان برخی مفاهیم پایه، جهت کسب آگاهی نسبت به آن‌ها پرداخته‌می‌شود. در فصل سوم، برخی ابزارهای مشابه و مرتبط با این پژوهش بررسی می‌شود، و ویژگی‌ها و کاستی‌های آن‌ها بیان می‌شود. در فصل چهارم، به شرح روش پیشنهادی برای اصلاح ترتیب اجرای رویه‌ها در مدل Promela تولید شده توسط Modex ، پرداخته شده‌است. در فصل پنجم، نتیجه‌گیری از پژوهش انجام شده آورده‌شده‌است.

فصل دوم

مفاهیم پایه

۲-۱ سیستم‌های هم‌زمان

همزمانی[1] به معنای انجام چند عملیات در یک زمان واحد است. هنگامی که در یک سیستم، چند نخ[2] اجرایی به صورت موازی[3] اجرا شوند، همزمانی رخ می‌دهد. هنگامی که همزمانی رخ می‌دهد، ممکن است برای مثال چند نخ اجرایی به صورت همزمان به یک منبع داده در سیستم دسترسی پیدا کنند. در این صورت، امکان بروز یک سری مشکلات احتمالی وجود دارد. به این دلیل است که به سیستم‌های هم‌زمان[4] توجه ویژه‌ای می‌شود.

۲-۲ سیستم‌های زمان‌ـواقعی

سیستم‌های زمان‌ـواقعی[5] ، سیستم‌هایی هستند که انجام عملیات و پردازش‌ها توسط آن‌ها باید در کسری از ثانیه رخ‌دهد. این سیستم‌ها محدودیت زمانی مشخصی دارند و باید آن را تضمین کنند. به این ترتیب، یک سیستم زمان‌ـواقعی یا یک عملیات را در آن زمان معین انجام می‌دهد، یا با شکست[6] مواجه می‌شود.

۲-۳ Spin

Spin [۱] محبوب ترین ابزار در جهان برای تشخیص نقص های نرم افزاری در طراحی سیستم های همزمان است. با این حال، کد های Promela را به عنوان ورودی در یافت می‌کند و نمی‌تواند برنامه های C را مستقیماً بررسی کند. بنابراین همواره سعی می‌شود تا روش‌هایی واسطه ای برای تبدیل کد C به کد Promela ابداع شود؛ تا امکان توصیف و بررسی کدهای C برای یافتن مشکلات احتمالی در برنامه های همزمان و سیستم های موازی با استفاده از Spin میسر شود.

---
[1] concurrency
[2] thread
[3] parallel
[4] concurrent systems
[5] real-time systems
[6] failure

این ابزار که به صورت متن‌باز[1] و رایگان دردسترس همه‌ی افراد است؛ قابلیت اجرا برروی سیستم‌عامل‌های Unix ، Linux ، Mac ، Solaris و بسیاری از نسخه‌های Windows را دارد. همچنین، علاوه بر اینکه با کمک خط فرمان[2] قابل استفاده‌است، دارای یک رابط کاربری گرافیکی کاربرپسند[3] است، که کار با آن را راحت‌تر می‌کند. در این پژوهش، در چندین مورد، از این ابزار قوی برای وارسی و تایید[4] مدل‌های Promela استفاده شده‌است. به علاوه، یک راهنمای بسیار کاربردی درباره‌ی نحوه‌ی نصب و اجرای قابلیت‌های مختلف آن دردسترس است.

۴-۲ زبان Promela

Promela [۴] یک زبان مدل‌سازی فرایند است، که از آن برای تست و وارسی منطق سیستم های موازی استفاده می‌شود. برای وارسی و تایید صحت مدل‌های نوشته شده در این زبان، از ابزار Spin استفاده می‌شود.
این زبان از نظر قواعد نحوی[5] ، به زبان C شباهت دارد. به همین دلیل است که می‌توان بسیاری از ساختمان‌داده‌های[6] معمول و ساده‌ی موجود در زبان C را به طور مستقیم به همان نوع ساختمان‌داده‌ها در Promela ترجمه کرد. از جمله تفاوت‌های موجود در بین این دو زبان، که در این پژوهش نیز مورد بررسی قرار گرفته‌است، عدم وجود توابع [7] (با همان رفتار مشابه توابع در زبان C) در زبان Promela است. بنابراین، توابع موجود در زبان C باید به رویه [8] ها در زبان Promela ترجمه شوند و با استفاده از امکانات دیگری که در این زبان وجود دارد، عملکرد توابع زبان C به خوبی شبیه‌سازی شود. در بخش های آتی، دقیق‌تر به این موضوع پرداخته‌خواهدشد.

۵-۲ زبان SDL (Specification and Description Language)

SDL یک زبان مدل‌سازی است که برای توصیف سیستم‌های زمان_واقعی استفاده می‌شود. نمودار SDL فرایند مدل‌سازی را نشان می‌دهد. این زبان می‌تواند به طور گسترده در سیستم‌های خودرو، هوانوردی، ارتباطات، پزشکی و مخابرات استفاده شود.

---
[1] open source
[2] command line
[3] user-friendly GUI
[4] verification
[5] syntactic rules
[6] data structures
[7] functions
[8] proctype

فصل سوم

سیستم های مشابه

۳-۱ ابزار Modex

Modex [۲] ابزاری است که حدوداً بیست سال پیش توسط جرارد هالزمن [1] توسعه یافته‌است. این ابزار با استفاده از زبان C نوشته شده‌است و هدف آن استخراج خودکار مدل به زبان Promela از کد C است. این ابزار برنامه‌های معمولی C را به شیوه‌ی معقولی به زبان Promela ترجمه می‌کند. اما هنوز محدودیت های آشکاری دارد؛ یعنی فقط از Promela به‌صورت محدودی استفاده می‌کند. به عنوان مثال، اشاره‌گر[2]ها را اداره نمی کند و هیچ فراخوانی رویه[3] ای ندارد. از آنجا که Promela به اندازه‌ی زبان C قدرتمند نیست، بسیاری از انواع داده های انعطاف پذیر و عملکردهای مبتکرانه ی موجود در زبان C را نمی‌توان مستقیماً به Promela ترجمه کرد. بنابراین، Modex این کدهای ترجمه ناپذیر را به صورت مستقیم در کد Promela تعبیه می‌کند و به این صورت به این نوع امکانات زبان C رسیدگی می‌کند. قطعات کد تعبیه‌شده[4] را نمی‌توان با SPIN بررسی کرد؛ نه در مرحله تجزیه[5] و نه در مرحله تأیید[6]. بنابراین، به آن‌ها به صورت کورکورانه اعتماد می‌شود، و از متن مدل، در کدی که Spin ایجاد می‌کند، کپی می‌شوند. به طور خاص، اگر یک قطعه کد C تعبیه‌شده شامل یک عملیات غیرمجاز باشد؛ مانند عملیات تقسیم بر صفر یا یک اشاره‌گر به فضای خالی[7]؛ نتیجه می‌تواند در حالی که مدل بررسی می‌شود، مخرب باشد. برای ترجمه‌ی این بخش‌های کد C به Promela ، برای اینکه بتوان آن‌ها را با Spin مورد وارسی قرارداد، باید از روش‌های خلاقانه تری نسبت به Modex استفاده کرد. یکی از مهم‌ترین محدودیت های Modex ، ترجمه‌ی اشاره‌گرهاست. Modex در این حد اشاره‌گرها را کنترل می‌کند، که هنگام استفاده از آن‌ها، تاکید[8]هایی را ایجاد می‌کند؛ اما فقط برای اطمینان از این هستند که اشاره‌گرها به فضای خالی از حافظه ارجاع داده‌نشده‌باشند. بنابراین، سعی نمی‌کند جزئیات را بررسی کند و مشکلات احتمالی

[1] Gerard Holzmann
[2] pointer
[3] procedure call
[4] embedded
[5] parsing phase
[6] verification phase
[7] NULL pointer
[8] assertion

هر اشاره‌گر را پیداکند، و فقط به بررسی عدم اشاره به فضای خالی از حافظه کفایت می‌کند.

۲-۳ ابزار JPF(Java PathFinder)

JPF [۳] در ابتدا در سال ۱۹۹۹ به عنوان مترجمی از زیرمجموعه‌ای از Java - شامل ایجاد شی پویا[۱]، وراثت[۲]، استثنائات[۳] و عملیات مربوط به نخ های اجرا[۴] - به Promela توسعه داده‌شد و از Spin برای بررسی مدل ترجمه شده استفاده‌کرد. این ابزار سعی می‌کرد کدهای Java را به کدهای با عملکردهای مشابه در Promela ترجمه کند. چند سال بعد، به عنوان یک ماشین مجازی Java [۵] توسعه داده‌شد، که به طور خودکار تمام مسیرهای اجرای احتمالی یک برنامه را برای یافتن نقض ویژگی ها [۶] مانند بن بست [۷] ها یا موارد استثنایی کنترل نشده [۸] مورد بررسی قرارمی‌دهد.

۳-۳ ابزار SDL2PML

SDL2PML [۵] ابزاری برای تولید خودکار مدل Promela از SDL است. کاملاً مستقل است و به هیچ ابزار خارجی متکی نیست. به صورت خودکار مدل هایی را تولید می‌کند که می‌توان با Spin آن‌ها را ارزیابی کرد. مدل‌هایی که تولید می‌کند، می‌توانند شامل کد C تعبیه‌شده‌باشند؛ یعنی از این مزیت برخوردار است که بخش‌هایی از کد را که نمی‌تواند ترجمه کند، به صورت کدهای تعبیه‌شده ی C در کد Promela قرار دهد. این ابزار با موفقیت در مواردی برای تولید مدل در کاربردهای دنیای واقعی مورد استفاده قرارگرفته‌است. کاستی قابل‌توجهی که در این ابزار وجود دارد، عدم پشتیبانی برنامه‌های با اجرای چندخطی[۹] است.

۴-۳ KE THESIS

Ke Jiang مقاله‌ای تحت عنوان Model Checking C Programs by Translating C to Promela [۶] ارائه کرده‌است؛ که در آن روش‌هایی موثر برای استخراج کد Promela از کد C معرفی شده‌است. این روش‌ها با تمرکز بر روی ضعف‌های سایر ابزارهایی که از پیش وجود داشتند و برای رفع آن کاستی‌ها ارائه شده‌اند.

---

[1] dynamic object creation
[2] inheritance
[3] exceptions
[4] thread operations
[5] JVM
[6] violations of properties
[7] deadlock
[8] unhandled exceptions
[9] multi-threaded execution

۳-۵ نتیجه‌گیری

با توجه به بررسی‌های صورت‌گرفته بر روی سیستم‌های مشابه، ابزار Modex بهترین ابزار دردسترس برای تبدیل کد C به Promela تشخیص‌داده‌شد. بنابراین، تصمیم بر بهبود برخی عملکردهای آن گرفته‌شد. می‌توان برای این کار، از روش‌های موجود در مقاله‌ی معرفی شده در بخش ۴.۳ الهام گرفت.

فصل چهارم

روش پیشنهادی

۴-۱ شرح مسئله

همانطورکه پیش‌تر بیان‌شد، یکی از نقاط ضعف Modex مربوط به عدم فراخوانی هیچ گونه رویه ای است. این موجب می‌شود تا تبدیل توابع کد C به رویه ها در کد Promela ، در برخی کدهای پیچیده‌تر با مشکل جدی مواجه شود. برای مثال، در برنامه‌های با فراخوانی‌های پیچیده‌تر توابع در زبان C ، ترتیب اجرای رویه های کد Promela تولید شده، مطابق با ترتیب اجرای صحیح توابع کد C اولیه نخواهدبود. برای درک بهتر مسئله، در ادامه با بیان یک مثال، این مشکل شرح داده‌خواهدشد و راه حلی با تمرکز بر روی همین مثال ارائه خواهدشد. شکل ۴-۱ که کد به زبان C را نشان می‌دهد، در نظر بگیرید.

۴-۲ مدل سازی با Modex

می خواهیم این کد را به Promela ترجمه کنیم،

۴-۲-۱ حالت اول

اگر فایل target.prx نداشته باشیم(یا این فایل خالی باشد)، و یا اینکه محتوای آن مانند کد نشان داده شده در شکل ۴-۲ باشد، کد Promela تولید شده توسط Modex ، به صورت کد نشان داده شده در دو شکل ۴-۳ و ۴-۴ خواهدبود.

حال برای بررسی این مدل و مسیرهای اجرایی ممکن در آن، از Spin استفاده می‌کنیم.

به طور کلی ۵ مسیر اجرایی ممکن به شرح زیر برای مدل تولید شده، محتمل است:

۱. به ترتیب از راست به چپ main ، f۱ ، f۲ و درنهایت f۳ اجرا می‌شود. و برنامه به اتمام می‌رسد.

۲. به ترتیب از راست به چپ main ، f۱ ، main ، f۱ ، f۲ و درنهایت f۳ اجرا می‌شود. و برنامه به اتمام می‌رسد.

```c
// target.c
int main()
{
   while(1)
   f1();
   return 0;
}
int f1()
{
   int i;
   for(i = 1; i < 10; i++)
   f2();
   return 0;
}
int f2()
{
   f3();
   return 0;
}
int f3()
{
   f1();
   return 0;
}
```

شکل ۴-۱: کد به زبان C

```
// target.prx
%x -xe
```

شکل ۴-۲: محتوای فایل prx.

```
// model.pml
int res_p_f3;
bool lck_p_f3_ret;
bool lck_p_f3;
int res_p_f2;
bool lck_p_f2_ret;
bool lck_p_f2;
int res_p_f1;
bool lck_p_f1_ret;
bool lck_p_f1;
int res_p_main;
bool lck_p_main_ret;
bool lck_p_main;
chan ret_p_f3 = [1] of { pid };
chan exc_cll_p_f3 = [0] of { pid };
chan req_cll_p_f3 = [1] of { pid };
chan ret_p_f2 = [1] of { pid };
chan exc_cll_p_f2 = [0] of { pid };
chan req_cll_p_f2 = [1] of { pid };
chan ret_p_f1 = [1] of { pid };
chan exc_cll_p_f1 = [0] of { pid };
chan req_cll_p_f1 = [1] of { pid };
chan ret_p_main = [1] of { pid };
chan exc_cll_p_main = [0] of { pid };
chan req_cll_p_main = [1] of { pid };
active proctype p_main()
{
  pid lck_id;
  L_0:
  do
  :: true;
  atomic {
    lck_p_f1 == 0 && empty(req_cll_p_f1) -> req_cll_p_f1!_pid;
    exc_cll_p_f1!_pid;
  }
  ret_p_f1?eval(_pid);
  c_code { ; now.lck_p_f1_ret = 0; };
  goto L_0;
  :: c_expr { !1 }; -> break
  od;
  atomic { !lck_p_main_ret -> lck_p_main_ret = 1 };
  c_code { now.res_p_main = (int ) 0; }; goto Return;
  Return: skip;
}
active proctype p_f1()
{
  int i;
  pid lck_id;
  endRestart:
  atomic {
    nempty(req_cll_p_f1) && !lck_p_f1 -> lck_p_f1 = 1;
    req_cll_p_f1?lck_id; exc_cll_p_f1?eval(lck_id);
    lck_p_f1 = 0;
  };
  c_code { Pp_f1->i=1; };
  L_1:
  do
  :: c_expr { (Pp_f1->i<10) };
```

شکل ۴-۳: فایل نتیجه Modex

```
    atomic {
      lck_p_f2 == 0 && empty(req_cll_p_f2) -> req_cll_p_f2!_pid;
      exc_cll_p_f2!_pid;
    }
    ret_p_f2?eval(_pid);
    c_code { ; now.lck_p_f2_ret = 0; };
    c_code { Pp_f1->i++; };
    goto L_1;
    c_code { Pp_f1->i++; };
    :: c_expr { !(Pp_f1->i<10) }; -> break
    od;
    atomic { !lck_p_f1_ret -> lck_p_f1_ret = 1 };
    c_code { now.res_p_f1 = (int ) 0; }; goto Return;
    Return: skip;
    ret_p_f1!lck_id;
    goto endRestart
}
active proctype p_f2()
{
    pid lck_id;
    endRestart:
    atomic {
      nempty(req_cll_p_f2) && !lck_p_f2 -> lck_p_f2 = 1;
      req_cll_p_f2?lck_id; exc_cll_p_f2?eval(lck_id);
      lck_p_f2 = 0;
    };
    atomic {
      lck_p_f3 == 0 && empty(req_cll_p_f3) -> req_cll_p_f3!_pid;
      exc_cll_p_f3!_pid;
    }
    ret_p_f3?eval(_pid);
    c_code { ; now.lck_p_f3_ret = 0; };
    atomic { !lck_p_f2_ret -> lck_p_f2_ret = 1 };
    c_code { now.res_p_f2 = (int ) 0; }; goto Return;
    Return: skip;
    ret_p_f2!lck_id;
    goto endRestart
}
active proctype p_f3()
{
    pid lck_id;
    endRestart:
    atomic {
      nempty(req_cll_p_f3) && !lck_p_f3 -> lck_p_f3 = 1;
      req_cll_p_f3?lck_id; exc_cll_p_f3?eval(lck_id);
      lck_p_f3 = 0;
    };
    atomic {
      lck_p_f1 == 0 && empty(req_cll_p_f1) -> req_cll_p_f1!_pid;
      exc_cll_p_f1!_pid;
    }
    ret_p_f1?eval(_pid);
    c_code { ; now.lck_p_f1_ret = 0; };
    atomic { !lck_p_f3_ret -> lck_p_f3_ret = 1 };
    c_code { now.res_p_f3 = (int ) 0; }; goto Return;
    Return: skip;
    ret_p_f3!lck_id;
    goto endRestart
}
```

شکل ۴-۴: ادامه فایل نتیجه Modex

۳. به ترتیب از راست به چپ main ، f۱ ، main ، و درنهایت f۱ اجرا می‌شود. و برنامه به اتمام می‌رسد.

۴. به ترتیب از راست به چپ main ، f۱ ، و درنهایت main اجرا می‌شود. و برنامه به اتمام می‌رسد.

۵. فقط main اجرا می‌شود. و برنامه به اتمام می‌رسد.

برای مثال، مسیر اجرایی چهارم به صورت شکل ۴-۵ است:

اگر به برنامه C موردنظر توجه کنید، متوجه می‌شوید که این برنامه در تابع اصلی (main) دارای یک حلقه‌ی بی‌نهایت است و منطقاً نباید هیچ‌گاه فراخوانی توابع به پایان برسد. ولی در مدل Promela تولید شده، در همه‌ی حالت‌ها فراخوانی توابع تمام می‌شود و مدل‌سازی به پایان می‌رسد.

حالا اگر به مدل Promela تولید شده توجه کنید، متوجه می‌شوید که این اتمام زودهنگام فراخوانی‌ها به همان دلیل عدم امکان فراخوانی رویه‌ها در کد تولیدی Modex ، که پیش‌تر عنوان شد، است. تمام رویه‌های تولید شده در این کد، به صورت فعال[1] استفاده شده‌اند و با استفاده از کانال‌های هم‌گام‌ساز [2] سعی شده که تا حدی ابتدایی ترتیب اجرای آن‌ها رعایت شود. ولی با توجه به اینکه رویه‌های از ابتدا فعال، فقط به صورت یک نمونه[3] از آن رویه عمل می‌کنند، این ساختار تولید مدل نمی‌تواند در یک شرایط پیچیده از فراخوانی رویه ها، پاسخگوی اجرای درست رویه‌ها باشد.

۴-۲-۲ حالت دوم

اگر محتوای فایل target.prx آن مانند کد نشان داده شده در شکل ۴-۶ باشد، کد Promela تولید شده توسط Modex ، به صورت کد نشان داده شده در شکل ۴-۷ خواهدبود.

در این حالت با توجه به اینکه رویه‌ها از ابتدا فعال هستند، ولی این‌بار با استفاده از کانال‌های هم‌گام‌ساز سعی در کنترل ترتیب اجرای آن‌ها نشده؛ تعداد مسیرهای اجرایی نسبت به حالت قبل هم بیشتر است و همچنان شامل مسیر درست نیست. بنابراین، این حالت نسبت به حالت اول هم عدم قطعیت[4] بیشتری دارد.

۴-۳ اصلاح مدل خروجی Modex

هدف این پژوهش اصلاح همین نقص Modex که در بخش ۲.۴ توصیف شده؛ یعنی عدم توانایی در مدل‌سازی صحیح مدل‌های پیچیده از لحاظ توالی فراخوانی توابع است.

در ادامه سعی می‌شود با ایجاد تغییراتی در کد مدل تولیدی Modex در بخش ۲.۲.۴ ، ترتیب فراخوانی رویه‌های

[1] active proctype
[2] synchronization channels
[3] instance
[4] non-determinism

```
Selected: 4
1: proc 0 (p_main:1) model-noprx.pml:33 (state 1) [(1)]
Selected: 4
2: proc 0 (p_main:1) model-noprx.pml:35 (state 2)
    [(((lck_p_f1==0)&&empty(req_cll_p_f1)))]
3: proc 0 (p_main:1) model-noprx.pml:35 (state 3) [req_cll_p_f1!_pid]
Selected: 3
4: proc 1 (p_f1:1) model-noprx.pml:53 (state 1) [((nempty(req_cll_p_f1)&&!(lck_p_f1)))]
5: proc 1 (p_f1:1) model-noprx.pml:53 (state 2) [lck_p_f1 = 1]
6: proc 1 (p_f1:1) model-noprx.pml:54 (state 3) [req_cll_p_f1?lck_id]
Selected: 4
7: proc 0 (p_main:1) model-noprx.pml:36 (state 4) [exc_cll_p_f1!_pid]
7: proc 1 (p_f1:1) model-noprx.pml:54 (state 4) [exc_cll_p_f1?eval(lck_id)]
8: proc 1 (p_f1:1) model-noprx.pml:55 (state 5) [lck_p_f1 = 0]
Selected: 3
c_code4: {  /* line 57 model-noprx.pml */
  Pp_f1->i=1; }
9: proc 1 (p_f1:1) model-noprx.pml:57 (state 7) [{c_code4}]
c_code5:  /* line 60 model-noprx.pml */
(Pp_f1->i<10)
c_code9:  /* line 70 model-noprx.pml */
!(Pp_f1->i<10)
Selected: 4
c_code9:  /* line 70 model-noprx.pml */
!(Pp_f1->i<10)
10:  proc 1 (p_f1:1) model-noprx.pml:70 (state 18) [({c_code9})]
Selected: 3
11:  proc 1 (p_f1:1) model-noprx.pml:59 (state 22) [break]
Selected: 3
12:  proc 1 (p_f1:1) model-noprx.pml:72 (state 23) [(!(lck_p_f1_ret))]
13:  proc 1 (p_f1:1) model-noprx.pml:72 (state 24) [lck_p_f1_ret = 1]
Selected: 3
c_code10: {  /* line 73 model-noprx.pml */
  14:  proc 1 (p_f1:1) model-noprx.pml:73 (state 26) [{c_code10}]
  Selected: 3
  15:  proc 1 (p_f1:1) model-noprx.pml:74 (state 28) [(1)]
  Selected: 3
  16:  proc 1 (p_f1:1) model-noprx.pml:75 (state 29) [ret_p_f1!lck_id]
  Selected: 4
  17:  proc 0 (p_main:1) model-noprx.pml:38 (state 6) [ret_p_f1?eval(_pid)]
  Selected: 4
  c_code1: {  /* line 39 model-noprx.pml */
    18:  proc 0 (p_main:1) model-noprx.pml:39 (state 7) [{c_code1}]
    c_code2:  /* line 41 model-noprx.pml */
    !1
    Selected: 5
    c_code2:  /* line 41 model-noprx.pml */
    !1
    19:  proc 0 (p_main:1) model-noprx.pml:41 (state 9) [({c_code2})]
    Selected: 4
    20:  proc 0 (p_main:1) model-noprx.pml:32 (state 13) [break]
    Selected: 4
    21:  proc 0 (p_main:1) model-noprx.pml:43 (state 14) [(!(lck_p_main_ret))]
    22:  proc 0 (p_main:1) model-noprx.pml:43 (state 15) [lck_p_main_ret = 1]
    Selected: 4
    c_code3: {  /* line 44 model-noprx.pml */
      23:  proc 0 (p_main:1) model-noprx.pml:44 (state 17) [{c_code3}]
      Selected: 4
      24:  proc 0 (p_main:1) model-noprx.pml:45 (state 19) [(1)]
```

شکل ۴-۵: مسیر اجرایی

```
// target.prx
%x -x
```

شکل ۴-۶: محتوای فایل prx.

مدل Promela را منطبق بر توالی اجرای تابع‌ها در کد C کرد.

۴-۳-۱ گام اول

کد حاصل از این گام، به صورت کد نشان داده شده در شکل ۴-۸ است.

تغییرات ایجاد شده، شامل موارد زیر است:

۱. قطعه کدهای ساده‌ی مشترک بین دو زبان که در بلاک‌های تعبیه شده‌ی کد C [۱] قرار داشت، از این بلاک‌ها خارج شده و به صورت کد Promela در مدل قرار گرفت. به این ترتیب، قابلیت وارسی این بخش از مدل هم توسط Spin فراهم شد.

۲. فقط تابع اصلی برنامه به صورت فعال تعریف شده‌است. بقیه‌ی توابع، در هنگام فراخوانی، با کلیداژه‌ی run اجرا می‌شوند. (برای اصلاح مشکل فراخوانی توابع، نمی‌توان از توابع inline بهره گرفت؛ چرا که در این توابع، ترتیب تعریف بدنه‌ی توابع برای امکان فراخوانی آن‌ها توسط یک‌دیگراهمیت دارد، و این موجب ایجاد محدودیت در تنوع فراخوانی های توابع می‌شود.)

حال برای بررسی این مدل و مسیرهای اجرایی ممکن در آن، از Spin استفاده می‌کنیم.

در این حالت، مشاهده می‌شود که همچنان تعداد مسیرهای اجرایی ممکن، بیش از یک مسیر است. با این حال، بهبودی که در این حالت دیده می‌شود، این است که ترتیب اجرای رویه‌ها در یکی از مسیرهای اجرایی، منطبق بر ترتیب اجرای توابع در کد C متناظر است. یعنی مدل Promela تهیه شده، علاوه بر رفتار مدل اصلی C ، رفتارهای دیگری هم از خود نشان می‌دهد. در این حالت اگر به این کد Promla به عنوان نماینده‌ی کد C مورد نظر و برای بررسی آن با استفاده از Spin تکیه کنیم، نتیجه می‌گیریم که مدل ما همواره رفتار موردنظر ما را ندارد. این نتیجه‌گیری نادرست، به دلیل وجود همان مسیرهای غیر از مسیر اصلی و درست کد C است. در این حالت اصطلاحاً گفته می‌شود که این نتیجه، منفی کاذب [۲] است.

این مشکل به دلیل این است که وقتی یک رویه، رویه‌ی دیگری را فراخوانی می‌کند، هم خود رویه‌ی اول و هم رویه‌ی فراخوانی شده، امکان اجرا دارند. در این حالت، اطمینانی وجود ندارد که بدنه‌ی کدام‌یک از این دو

[1] C-expression
[2] false negative

```
// model.pml
active proctype p_main()
{
   L_0:
   do
   :: true;
   c_code { f1(); };
   goto L_0;
   :: c_expr { !1 }; -> break
   od;
   goto Return;
   Return: skip;
}
active proctype p_f1()
{
   int i;
   c_code { Pp_f1->i=1; };
   L_1:
   do
   :: c_expr { (Pp_f1->i<10) };
   c_code { f2(); };
   c_code { Pp_f1->i++; };
   goto L_1;
   c_code { Pp_f1->i++; };
   :: c_expr { !(Pp_f1->i<10) }; -> break
   od;
   goto Return;
   Return: skip;
}
active proctype p_f2()
{
   c_code { f3(); };
   goto Return;
   Return: skip;
}
active proctype p_f3()
{
   c_code { f1(); };
   goto Return;
   Return: skip;
}
```

شکل ۴-۷: فایل نتیجه Modex

```
// model.pml
active proctype p_main()
{
   L_0:
   do
   :: true;
   run p_f1();
   goto L_0;
   :: !1 -> break
   od;
   goto Return;
   Return: skip;
}
proctype p_f1()
{
   int i;
   i=1;
   L_1:
   do
   :: i<10;
   run p_f2();
   i++;
   goto L_1;
   i++;
   :: !(i<10) -> break
   od;
   goto Return;
   Return: skip;
}
proctype p_f2()
{
   run p_f3();
   goto Return;
   Return: skip;
}
proctype p_f3()
{
   run p_f1();
   goto Return;
   Return: skip;
}
```

شکل ۴-۸: کد حاصل از گام اول

```
0: proc - (:root:) creates proc 0 (p_main)
1: proc 0 (p_main:1) model-prefer.pml:8 (state 1) [(1)]
Starting p_f1 with pid 1
2: proc 0 (p_main:1) creates proc 1 (p_f1)
2: proc 0 (p_main:1) model-prefer.pml:9 (state 2) [(run p_f1())]
Selected: 1
3: proc 1 (p_f1:1) model-prefer.pml:19 (state 1) [i = 1]
Selected: 1
4: proc 1 (p_f1:1) model-prefer.pml:22 (state 2) [((i<10))]
Selected: 1
Starting p_f2 with pid 2
5: proc 1 (p_f1:1) creates proc 2 (p_f2)
5: proc 1 (p_f1:1) model-prefer.pml:23 (state 3) [(run p_f2())]
Selected: 1
Starting p_f3 with pid 3
6: proc 2 (p_f2:1) creates proc 3 (p_f3)
6: proc 2 (p_f2:1) model-prefer.pml:34 (state 1) [(run p_f3())]
Selected: 1
Starting p_f1 with pid 4
7: proc 3 (p_f3:1) creates proc 4 (p_f1)
7: proc 3 (p_f3:1) model-prefer.pml:40 (state 1) [(run p_f1())]
Selected: 1
8: proc 4 (p_f1:1) model-prefer.pml:19 (state 1) [i = 1]
.
.
.
```

شکل ۴-۹: مسیر اجرایی مربوط به کد حاصل از گام اول

تابع اجرا خواهدشد. و این قضیه موجب ایجاد عدم قطعیت در رفتار مدل می‌شود. پس باید مسیر اجرای یکی از این دو رویه بلاک شود، تا مسیر مورد انتظار اجرا شود.

مسیر موردانتظار که در مسیرهای اجرایی این مدل وجود دارد، به صورت شکل ۴-۹ است.

۴-۳-۲ گام دوم

کد حاصل از این گام، به صورت کد نشان داده شده در دو شکل ۴-۱۰ و ۴-۱۱ است.

دراین گام سعی شد تا با تعریف دو متغیر عمومی[1]، در هنگامی که بدنه‌ی چند رویه به صورت هم‌زمان امکان اجرا داشتند، رویه‌هایی که موجب مسیر غیرمنتظره می‌شوند بلاک شود. یکی از متغیرها به منظور نشان دادن این است که رویه‌ای که باید اجرا شود، کدام است[2]. متغیر دیگر این را نشان می‌دهد که اگر رویه‌ای که در حال حاضر در آن قرار داریم به اتمام رسید، کدام رویه باید اجرا شود(با توجه به اینکه رویه‌ی حال حاضر توسط چه رویه‌ای فراخوانی شده است، تنظیم می‌شود) [3].

در این حالت، همچنان دو مشکل وجود دارد:

۱. دراین حالت، با استفاده از یک حلقه سعی شده در هر رویه چک شود که آیا می‌تواند به اجرای بدنه‌ی

[1] global variables
[2] function turn
[3] function prev

```
// model.pml
int function_turn;
int function_prev;
active proctype p_main()
{
   function_turn = 0;
   function_prev = -1;
   L_0:
   do
   :: true;
   function_prev = 0;
   function_turn = 1;
   run p_f1();
   L_turn_0:
   do
   :: !(function_turn == 0);
   goto L_turn_0;
   :: function_turn == 0 -> break
   od;
   goto L_0;
   :: !1 -> break
   od;
   goto Return;
   Return: skip;
   function_turn = function_prev;
}
proctype p_f1()
{
   int i;
   i=1;
   L_1:
   do
   :: i<10;
   function_prev = 1;
   function_turn = 2;
   run p_f2();
   L_turn_1:
   do
   :: !(function_turn == 1);
   goto L_turn_1;
   :: function_turn == 1 -> break
   od;
   i++;
   goto L_1;
```

شکل ۴ - ۱۰ : کد حاصل از گام دوم

```
    i++;
    :: !(i<10) -> break
    od;
    goto Return;
    Return: skip;
    function_turn = function_prev;
}
proctype p_f2()
{
    function_prev = 2;
    function_turn = 3;
    run p_f3();
    L_turn_2:
    do
    :: !(function_turn == 2);
    goto L_turn_2;
    :: function_turn == 2 -> break
    od;
    goto Return;
    Return: skip;
    function_turn = function_prev;
}
proctype p_f3()
{
    function_prev = 3;
    function_turn = 1;
    run p_f1();
    L_turn_3:
    do
    :: !(function_turn == 3);
    goto L_turn_3;
    :: function_turn == 3 -> break
    od;
    goto Return;
    Return: skip;
    function_turn = function_prev;
}
```

شکل ۴-۱۱: ادامه کد حاصل از گام دوم

خود ادامه دهد، یا بلاک شود. برای مثال وقتی که یک رویه، رویه‌ی دیگری را فراخوانی می‌کند، پس از آن باید وارد حلقه شود و به طور مرتب چک کند که آیا می‌تواند به اجرای بدنه‌ی خود ادامه بدهد یا خیر. اینجا مشکلی که پیش می‌آید، این است که ممکن است تمام cpu به رویه‌ی اول اختصاص پیدا کند و رویه‌ی دوم نتواند هیچ‌وقت اجرا شود. بنابراین، همچنان قطعیتی روی مسیر اجرای برنامه وجود ندارد، و مشخص نیست بدنه‌ی کدام رویه پس از آن اجرا می‌شود.

۲. مشکل دیگر، مربوط به کنترل مسیر اجرا با استفاده از متغیرهای عمومی به کارگرفته شده‌است. در این حالت، هر رویه اگر بیش از یک بار به روش های مختلف قصد اجرا داشته باشد، این متغیرها کارایی خود را از دست می‌دهند و به درستی ترتیب اجرای رویه‌ها را نمی‌توانند کنترل کنند. برای مثال اگر در تابع ۱، تابع ۲ فراحوانی شد، بدنه‌ی تابع ۱ روی خط بعد از فراخوانی تابع ۲ باید بلاک شود. حالا اگر دوباره از طریق تابع ۲، تابع ۱ فراخوانی شود، تابع ۱ هم از ابتدا می‌تواند اجرا شود و هم از خطی که در میانه ی بدنه‌اش بلاک شده بود؛ درحالی که ما انتظار داشتیم تابع ۱ فقط از ابتدا اجرا شود و نمونه ی اولیه‌ی آن همچنان در بدنه‌ی خود بلاک بماند تا آن نمونه از تابع ۱ که اجرا کرده‌بود، به اتمام برسد.

۴-۳-۳ گام سوم

کد حاصل از این گام، به صورت کد نشان داده شده در شکل ۴-۱۲ است.

در این حالت، برای کنترل ترتیب اجرای رویه‌ها، به جای استفاده از متغیرهای عمومی، از کانال‌های هم‌گام‌ساز محلی[1] استفاده می‌شود. به این ترتیب، هر دو مشکلی که در گام قبل با آن‌ها مواجه بودیم، برطرف می‌شود. روش استفاده از این کانال‌ها به این صورت است که در هر فراخوانی رویه، یک کانال هم‌گام‌ساز بین دو رویه فراخوانی‌کننده و فراخوانی‌شونده تنظیم می‌شود، تا اجرای این دو رویه را کنترل کند. به این ترتیب، با توجه به اینکه در هر فراخوانی رویه، یک کانال جداگانه بین نمونه‌های رویه‌ها تنظیم می‌شود، مشکل دوم گام قبل حل می‌شود. همچنین، در بدنه‌ی رویه فراخوانی‌کننده، در خط پس از خط فراخوانی، منتظر دریافت مقدار از طریق کانال می‌ماند و در همان خط بلاک می‌شود و نیاز به اختصاص مرتب cpu نخواهدداشت و عدم قطعیتی در مسیر اجرا ایجاد نمی‌کند.

تنها مسیر اجرای حاصل از مدل در این گام، که همان مسیر درست است، در شکل ۴-۱۳ نشان داده‌شده‌است.

[1] local synchronization channels

```
// model.pml
active proctype p_main()
{
   L_0:
   do
   :: true;
   int flag;
   chan out_p_f1 = [0] of {int};
   run p_f1(out_p_f1);
   out_p_f1 ? flag;
   goto L_0;
   :: !1 -> break
   od;
   goto Return;
   Return: skip;
}
proctype p_f1(chan out_p_f1)
{
   int i;
   i=1;
   L_1:
   do
   :: i<10;
   int flag;
   chan out_p_f2 = [0] of {int};
   run p_f2(out_p_f2);
   out_p_f2 ? flag;
   i++;
   goto L_1;
   i++;
   :: !(i<10) -> break
   od;
   goto Return;
   Return: skip;
   out_p_f1 ! 1;
}
proctype p_f2(chan out_p_f2)
{
   int flag;
   chan out_p_f3 = [0] of {int};
   run p_f3(out_p_f3);
   out_p_f3 ? flag;
   goto Return;
   Return: skip;
   out_p_f2 ! 1;
}
proctype p_f3(chan out_p_f3)
{
   int flag;
   chan out_p_f1 = [0] of {int};
   run p_f1(out_p_f1);
   out_p_f1 ? flag;
   goto Return;
   Return: skip;
   out_p_f3 ! 1;
}
```

شکل ۴-۱۲: کد حاصل از گام سوم

```
0: proc - (:root:) creates proc 0 (p_main)
1: proc 0 (p_main:1) model-prefer.pml:8 (state 1) [(1)]
2: proc 0 (p_main:1) model-prefer.pml:10 (state 2) [flag = 0]
Starting p_f1 with pid 1
3: proc 0 (p_main:1) creates proc 1 (p_f1)
3: proc 0 (p_main:1) model-prefer.pml:11 (state 3) [(run p_f1(out_p_f1))]
Selected: 1
4: proc 1 (p_f1:1) model-prefer.pml:22 (state 1) [i = 1]
Selected: 1
5: proc 1 (p_f1:1) model-prefer.pml:25 (state 2) [((i<10))]
Selected: 1
6: proc 1 (p_f1:1) model-prefer.pml:27 (state 3) [flag = 0]
Selected: 1
Starting p_f2 with pid 2
7: proc 1 (p_f1:1) creates proc 2 (p_f2)
7: proc 1 (p_f1:1) model-prefer.pml:28 (state 4) [(run p_f2(out_p_f2))]
Selected: 1
Starting p_f3 with pid 3
8: proc 2 (p_f2:1) creates proc 3 (p_f3)
8: proc 2 (p_f2:1) model-prefer.pml:43 (state 1) [(run p_f3(out_p_f3))]
Selected: 1
Starting p_f1 with pid 4
9: proc 3 (p_f3:1) creates proc 4 (p_f1)
9: proc 3 (p_f3:1) model-prefer.pml:53 (state 1) [(run p_f1(out_p_f1))]
Selected: 1
10:  proc 4 (p_f1:1) model-prefer.pml:22 (state 1) [i = 1]
Selected: 1
11:  proc 4 (p_f1:1) model-prefer.pml:25 (state 2) [((i<10))]
Selected: 1
12:  proc 4 (p_f1:1) model-prefer.pml:27 (state 3) [flag = 0]
Selected: 1
Starting p_f2 with pid 5
13:  proc 4 (p_f1:1) creates proc 5 (p_f2)
13:  proc 4 (p_f1:1) model-prefer.pml:28 (state 4) [(run p_f2(out_p_f2))]
Selected: 1
Starting p_f3 with pid 6
14:  proc 5 (p_f2:1) creates proc 6 (p_f3)
14:  proc 5 (p_f2:1) model-prefer.pml:43 (state 1) [(run p_f3(out_p_f3))]
Selected: 1
Starting p_f1 with pid 7
15:  proc 6 (p_f3:1) creates proc 7 (p_f1)
15:  proc 6 (p_f3:1) model-prefer.pml:53 (state 1) [(run p_f1(out_p_f1))]
Selected: 1
16:  proc 7 (p_f1:1) model-prefer.pml:22 (state 1) [i = 1]
Selected: 1
17:  proc 7 (p_f1:1) model-prefer.pml:25 (state 2) [((i<10))]
Selected: 1
18:  proc 7 (p_f1:1) model-prefer.pml:27 (state 3) [flag = 0]
Selected: 1
Starting p_f2 with pid 8
19:  proc 7 (p_f1:1) creates proc 8 (p_f2)
19:  proc 7 (p_f1:1) model-prefer.pml:28 (state 4) [(run p_f2(out_p_f2))]
.
.
.
```

شکل ۴-۱۳: مسیر اجرایی مربوط به کد حاصل از گام سوم

فصل پنجم

نتیجه‌گیری

۵-۱ پژوهش حاضر

در این پژوهش سعی بر آن شد تا با تمرکز بر روی مدل‌های با فراخوانی‌های پیچیده‌ی توابع، بهبودی در عملکرد ابزار Modex برای تولید خودکار مدل Promela از کد C صورت گیرد. در این راستا، یک ماژول خودکار توسعه داده‌شد، که فایل Promela تولید شده توسط Modex را به عنوان ورودی دریافت می‌کند، و با اعمال اصلاحات بیان شده در فصل پیشین، مدلی بهبودیافته به عنوان خروجی تولید می‌کند.

۵-۲ کارهای آتی

با توجه به بررسی‌هایی که روی کاستی‌های Modex در تولید خودکار کد Promela انجام شد؛ در آینده تلاش برای تعمیم امکان فراخوانی توابع و بهبود مسیر اجرا در هنگام وجود توابع بازگشتی[1] و توابع دارای پارامتر ورودی، صورت خواهدگرفت. به علاوه، سعی می‌شود، مطالعات برای اصلاح تبدیل اشاره‌گرها ادامه‌یابد. همچنین ویژگی‌ها[2] و کارایی[3] مدل‌های حاصل از روش‌های پیشنهادی، بررسی خواهدشد.

[1] recursive functions
[2] properties
[3] performance

مراجع

[1] Spin reference, *https://spinroot.com/spin/Man/README.html*.

[2] Modex reference, *https://spinroot.com/modex/MANUAL.html*.

[3] K. Havelund and T. Pressburger, *Model checking JAVA programs using JAVA PathFinder*, Int. J. Softw. Tools Technol. Transf., vol. 2, no. 4, pp. 366–381, Apr. 2000.

[4] Promela reference, *http://spinroot.com/spin/Man/promela.html/*.

[5] B. Vlaovič, A. Vreže and Z. Brezočnik, *Applying Automated Model Extraction for Simulation and Verification of Real-Life SDL Specification With Spin*, in IEEE Access, vol. 5, pp. 5046-5058, 2017, doi: 10.1109/ACCESS.2017.2685238.

[6] K. Jiang, *Model Checking C Programs by Translating C to Promela*, Dissertation, 2009.

[7] H. Mousavi, E. Mahmoudzadeh and A. Ebnenasir, *A Promela Model for Contiki's Scheduler*, 2020 CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST), 2020, pp. 1-10, doi: 10.1109/RTEST49666.2020.9140094.