Generování testovacích stimulů

Ondřej Čekan
3. ročník, prezenční studium
Školitel: Zdeněk Kotásek

Fakulta informačních technologií, Vysoké učení technické v Brně
Božetěchova 2, 612 66 Brno, ČR
icekan@fit.vutbr.cz

***Abstrakt* — Tento článek popisuje jednotlivé kroky k dosažení stanovených cílů disertační práce a přehled výsledků dosažených za dobu studia. Hlavní část práce je věnována generátoru testovacích stimulů, který je založen na dvou vstupních strukturách popisujících formát a omezení generovaných stimulů. V rámci práce je ukázáno generování assemblerovských programů pro procesory, generování softwarově implementované odolnosti proti poruchám a generování bludiště. Tyto případy představují různé oblasti v použití tohoto generátoru a v práci jsou řádně zadokumentovány včetně ukázek. Závěrem práce je představena budoucí práce.**

Klíčová slova* — *verifikace; generování; stimul; program; odolnost proti poruchám; bludiště.

I. Úvod

Elektronické obvody se dostávají stále více do popředí našeho každodenního života a dalo by se říci, že od 21. století si bez nich svět již nedokážeme představit. Mikrokontroléry, které mají na starost řízení určitých zařízení, se nacházejí dnes i tam, kde bychom to ani nečekali. Jedná se o hračky, kuchyňské spotřebiče, chytré přívěsky nebo dokonce šperky. Takovéto použití představuje spíše high-end současné doby a obvody, které takovéto zařízení řídí, nejsou abnormálně spolehlivé. Na druhou stranu ale existují aplikace, u nichž jakýkoli nedostatek v návrhu nebo ve funkci systému může znamenat vážné riziko a v ohrožení se nacházejí především lidské životy. Takovéto aplikace se označují jako kritické z hlediska bezpečnosti (safety-critical) a představují je oblasti automobilového průmyslu, letectví, kosmonautiky či medicíny.

V případě, že cílíme na systémy, které neobsahují návrhové ani implementační chyby, které by mohly způsobit nekorektní chování, musí být tyto systémy důkladně otestovány. V úvahu se berou obvyklé, ale i neobvyklé kombinace vstupních hodnot, které mohou v daném systému nastat. Jelikož neustále roste složitost systému, tak roste i složitost spojená s důkladným ověřením jeho správné funkce [1]. Jednoduché systémy není složité otestovat manuálně. U komplexnějších systémů je manuální testování velmi časově náročné. Rovněž doposud vyvinuté formální techniky pro ověření rozsáhlých systémů selhávají. Z tohoto důvodu byla vytvořena technika zvaná funkční verifikace, která na základě vstupních a výstupních hodnot ze systémů ověřuje jejich korektnost. Hodnoty, které vstupují do systému, se typicky získávají pomocí generátoru stimulů (testů), který je předmětem této práce.

V současnosti existuje řada nástrojů, které jsou schopny generovat vstupní stimuly [2,3]. Tyto nástroje se ale zaměřují na jeden konkrétní systém, typicky procesor typu RISC. Systém pro generování stimulů, který by byl schopný uplatnit se v různorodých oblastech, neexistuje, proto je naším cílem takovýto systém vytvořit. Pomocí tohoto systému chceme definovat závěry v podobě obecných principů konstrukce testovacích stimulů pro různé systémy.

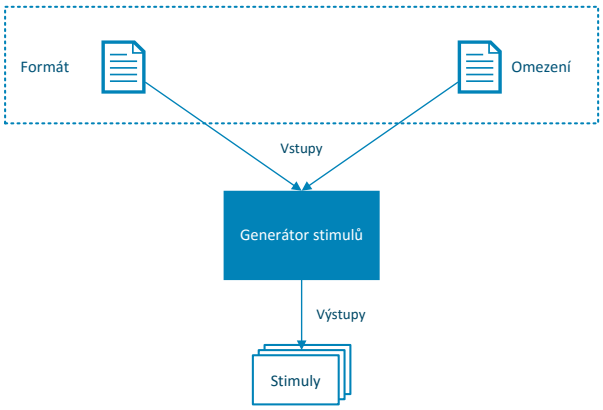
II. Cíle Práce

Hlavními cíli této práce jsou:

1. Navrhnout a vytvořit univerzální generátor testovacích stimulů založený na řešení problému s omezujícími podmínkami, který bude vhodný především pro použití ve funkční verifikaci.
2. Navrhnout obecný a jednotný popis stimulů různých systémů, pomocí něhož lze popsat veškeré podmínky a zákonitosti nutné k vygenerování platného testovacího stimulu. Výsledkem této činnosti bude vytvoření obecných principů konstrukce testovacích stimulů pro různé systémy.

III. Generátor Stimulů

Princip generování stimulů je založen na námi navržené architektuře [4], která je znázorněna na Obrázku 1. Tato architektura má za cíl zjednodušit a urychlit vytváření stimulů pro různé systémy.



Obr. 1. Architektura námi navrženého generátoru stimulů.

Základní idea je založena na dvou specifických vstupních strukturách, které definují formát generovaných dat (*Formát*) a omezující podmínky (*Omezení*) určující, jak má být s těmito formáty nakládáno při jejich generování. Tyto dva popisy představují vstup do *Generátoru stimulů*, který na základě jejich obsahu generuje validní stimuly na jeho výstup. Tyto stimuly pak již mohou být předány konkrétnímu systému jako vstupní data. Uživatel při vytváření vlastních stimulů neprogramuje žádný kód, ale pouze specifikuje požadovaný formát stimulů a omezení, které generátor na základě řešení problému s omezujícími podmínkami vyřeší. Důležitým předpokladem pro tento popsaný princip generování je široká množina vstupní abecedy, která musí obecně popisovat problémy a omezení různých stimulů. Tento princip generování stimulů je parametrizovatelný. Dokáže za běhu zpracovávat a měnit omezující podmínky změnou vstupních struktur, a tudíž je vhodný pro použití v procesu funkční verifikace.

A. *Formát stimulu*

Formát stimulu popisujeme pomocí tří základních části: *Nahrazení*, *Proměnné* a *Syntaxe*.

Část *Nahrazení* definuje identifikátor a všechny možné substituce, za které je možno daný identifikátor nahradit. Jedná se o obdobu výčtového datového typu. Nahrazuje se za konkrétní řetězec z definované množiny hodnot. V každém novém cyklu generování dochází k náhodnému zvolení určité substituce pro daný identifikátor.

Část *Proměnné* definuje proměnné v obecném slova smyslu. Pro každou proměnnou je přiřazena náhodná hodnota v závislosti na jejím datovém typu a to v každém cyklu generování.

Část *Syntaxe* syntakticky popisuje řetězce, jeden po druhém, které mají být náhodně generovány na výstup generátoru jako součást stimulu. V jednotlivých definovaných řetězcích se mohou nacházet identifikátory z částí *Nahrazení* a *Proměnné*. Tyto identifikátory jsou nahrazeny za konkrétní nebo náhodnou hodnotu v závislosti na typu identifikátoru. Část *Syntaxe* představuje statické hodnoty v generovaném řetězci, zatímco zbylé dvě části představují dynamické (měnící se) hodnoty v generovaném řetězci.

B. *Omezení*

Omezeními povolujeme generování pouze těch stimulů, které jsou platné pro zvolený systém. Jedná se především o omezení pro datové hodnoty (proměnná může nabývat pouze hodnot z určitého rozsahu) nebo závislostní omezení (některá kombinace hodnot nemůže nastat po aktuálně generované kombinaci). Omezení jsou unikátní pro každý systém, a tedy různá omezení jsou aplikována na různé systémy.

C. *Generátor stimulů*

Generátor stimulů kombinuje *Syntaxe*, *Nahrazení* a *Proměnné* tak, že všechna omezení jsou splněna - žádné není porušeno. Výstupem generátoru je posloupnost řádků, která odpovídá definovanému problému a která tvoří výsledný stimul.

IV. Generování Programů

Generování assemblerovského programu pro procesory je prvním příkladem použití tohoto generátoru. Vstupní struktury byly vytvořeny speciálně pro procesor Codix RISC [5], ale úpravou lze popsat program pro jiný procesor obdobného typu.

A. *Formát*

Část *Syntaxe* obsahuje instrukční sadu zvoleného procesoru. Každá definovaná instrukce v této části se skládá z identifikátoru a své assemblerovské reprezentace. Identifikátor slouží jako odkaz mezi instrukcí a omezeními. Připravené assemblerovské reprezentace instrukcí v sobě obsahují další identifikátory, které jsou definovány a nahrazovány z částí *Nahrazení* (ty představují především registry) a *Proměnné* (ty představují především náhodná čísla a řetězce). Příklad definice instrukce OR:

```
ori { "dst = or src1, imm" }
```

kde *ori* je identifikátor instrukce, řetězec ve složených závorkách je assemblerovská reprezentace instrukce, *dst* a *src1* jsou identifikátory z části *Nahrazení* a *imm* je jméno proměnné. Pro procesor Codix RISC bylo potřeba vytvořit 61 takovýchto definic instrukcí.

Část *Nahrazení* definuje množinu řetězců, za které může být nahrazena určitá část assemblerovské reprezentace instrukce. Tato část je použita především pro definici dostupných registrů procesoru. Příklad definice jednoho nahrazení:

```
dst { r0|r1|r2 }
```

kde *dst* je identifikátor nahrazení a *r0*, *r1*, a *r2* jsou možné řetězce, kterými je identifikátor nahrazován.

Část *Proměnné* definuje proměnné, do kterých je přiřazena náhodná hodnota v závislosti na jejich datovém typu. Tato část je použita pro definici a přiřazení přímých operandů anebo řetězců jako návěští skokových instrukcí. Příklad definice jedné proměnné:

```
VAR16 imm
```

kde datový typ *VAR16* určuje 16-bitové celé číslo a *imm* je jméno proměnné.

B. *Omezení*

Pro běžné instrukce, které nevyžadují zvláštní předzpracování, omezení hodnot či kooperaci s jinými instrukcemi, není zapotřebí definovat žádná omezení. V případě, že není žádné omezení pro danou instrukci definováno, znamená to, že se tato instrukce může nacházet kdekoliv v programu a to po jakékoliv instrukci. Některé instrukce ale omezení vyžadují, aby byla zajištěna jejich platná sekvence a jejich platné hodnoty operandů. Proto bylo vytvořeno několik omezení, které řeší typické problémy při generování assemblerovského kódu. Těchto omezení pro procesor Codix RISC je celkem 18. Příklad definice několika omezení:

```
end("nop//halt")     # pro ukončení programu
nouse(add(dst),1)    # pro latence
```

```
contain(label(name),jump(name)) # pro skoky
unique(label(name)) # pro skoky
mdiv(imm,4)         # pro zarovnání paměti
```

C. *Shrnutí*

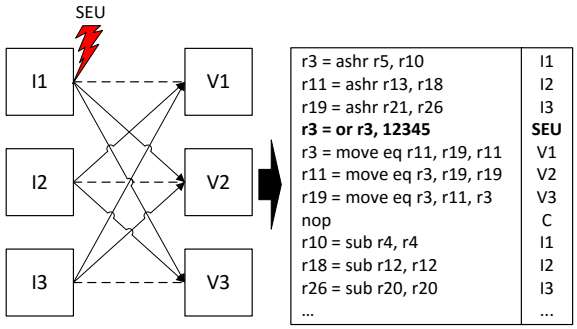
Princip popsaný v této kapitole byl využit v praxi v komerční společnosti a publikován v časopise [6].

V. Generování Softwarově Implementované Odolnosti Proti Poruchám

Základní idea je založena na informační redundanci, která je přidána do assemblerovského programu. Využili jsme princip *Triple Modular Redundancy* (TMR) v softwaru a definovali jsme proto obdobu *Triple Instructional Redundancy* (TIR). TIR pracuje na principu ztrojení jednotlivých instrukcí programu a vyhodnocení majority. Tím dojde k detekci a opravě chyby v registrech procesoru nebo paměti. Korektní funkčnost a odolnost samotného programu jsme ověřili pomocí softwarové injekce poruchy.

A. *Softwarová injekce poruchy*

Simulace poruchy v softwaru je provedena vložením nezabezpečené instrukce do zabezpečeného programu. Tímto jsme schopni přepsat hodnotu jednoho paměťového prvku a následně jednonásobnou poruchu opravit se 100% úspěšností. Tato operace je ekvivalentní se *single event upset* (SEU) poruchou. Pomocí této injekce jsme schopni rovněž vygenerovat více poruch na různá místa programu a tak simulovat více poruch. Příklad injekce do TIR programu je ukázán na Obrázku 2.

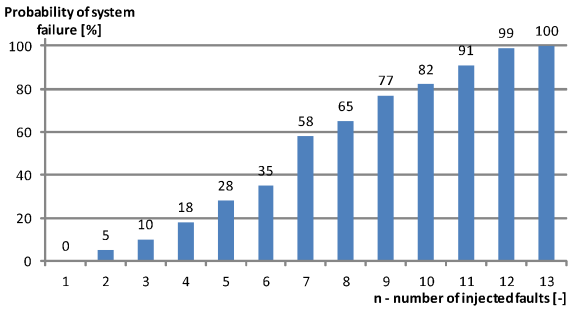


Obr. 2. Příklad softwarové injekce poruchy do zabezpečeného programu.

B. *Generování TIR programu*

Každá instrukce v části Syntaxe je ztrojena ve vstupní struktuře definující formát generovaného programu a je následována trojicí porovnávacích instrukcí *move*, které představují voliče pro vyhodnocení majority. Omezení obsahují speciální definice zajišťující správné rozdělení a používání registrového a paměťového prostoru, aby si jednotlivé TIR instrukce nepřepisovaly své výsledky. SEU porucha je generována instrukcí bez ztrojení. Pro ověření správnosti TIR programu jsme generovali jak zabezpečený program, tak i nezabezpečený.

V rámci experimentů jsme generovali až 13 poruch v rámci jednoho programu o 100 instrukcích. Poruchy byly umísťovány náhodně mezi TIR instrukce a voliče. Výsledek určující procentuální selhání programu při vícenásobné poruše ukazuje Obrázek 3.



Obr. 3. Pravděpodobnost selhání systému pokud je injektováno *n* poruch.

C. *Shrnutí*

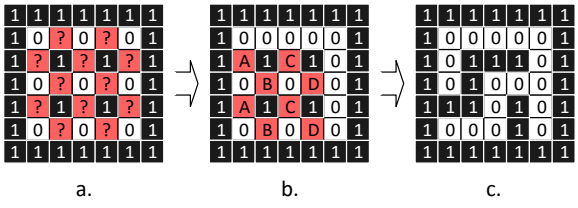
Tento přístup byl ověřen pro procesor Codix RISC a byl publikován na konferenci [7].

VI. Generování Bludiště

Generování bludiště je velmi známá a prozkoumaná oblast, pro kterou existuje značné množství algoritmů, jak generovat jednoduchá nebo sofistikovanější bludiště. Drtivá většina algoritmů pracuje ve dvourozměrném prostoru, uchovávají si aktuální stav a dokáží neustále měnit hodnoty buněk bludiště v čase. Takovéto algoritmy jsou pro námi navrženou architekturu univerzálního generátoru značně nevhodné, jelikož výstup generátoru (v našem případě se jedná o řádek bludiště) není možno určit v jednom kroku, ale je určen mnoha faktory a závislostmi mezi různými buňkami bludiště. Existuje ovšem algoritmus, který je založen na binárním stromu a konkrétní řádek bludiště lze určit jen a pouze z předcházejícího řádku. Takovýto princip je pro náš generátor zcela vyhovující a výstupní bludiště zcela dostačuje pro naše potřeby.

A. *Základní princip*

Základní princip algoritmu je ukázán na Obrázku 4. Vychází se ze základní matice bludiště (a), u které jsou některá pole pevně zadaná – buď chodba, nebo stěna. Chodby reprezentujeme pomocí nul, kdežto stěny pomocí jedniček. Pole označená otazníkem reprezentují oblasti, které mohou nabývat hodnotu 0 nebo 1, tedy chodba nebo stěna. Aby byla zachována průchodnost bludiště z jakékoliv chodby do jiné, je potřeba provést modifikaci základního bludiště tak, že vždy dvě přilehlé strany bludiště musí obsahovat chodbu přes celou jeho šíři (b). V našem případě jsme si zvolili tuto chodbu na severní a východní straně bludiště. Posledním nejkritičtějším úkolem je určit buňky A,B,C,D tak, aby bludiště bylo spojité (c).



Obr. 4. Ukázka převodu základního bludiště pro potřeby generátoru.

Originální popis algoritmu rozdělí buňky bludiště na daném řádku do skupin chodeb ohraničených zdmi. Pro každou takovouto skupinu algoritmus určí jeden vchod buď v severní, nebo východní části ohraničení skupiny. Tímto způsobem je zajištěn průchod ze severní části bludiště do jižní a to samé platí pro průchod ze západu na východ. Tento princip jsme převedli do podoby závislostí jednoho řádku bludiště na druhém a výsledkem je následující závislost. Pokud na Obrázku 4.b buňka A respektive C byla náhodně vybrána za chodbu, pak buňka B respektive D bude zdí a naopak.

B. *Aplikování na generátor*

Struktura popisující formát generování v tomto případě definuje množinu hodnot a požadovaný výstup – nuly a jedničky. Struktura s omezeními zahrnuje podmínky vycházející z předchozího odstavce, které jsou nutné pro spojité generování bludiště. Ukázky jednoduchosti jednotlivých popisů bez dalšího vysvětlení, které dostačují k vygenerování bludiště o rozměrech 7x7 buněk jsou níže:

```
----- Formát -----

substitute {
  A,C { "0"|"1" }
  B,D { "0" }
}

syntax {
  odd { "1A1C101" }
  even { "10B0D01" }
}
```

```
----- Omezení -----

constraints {
  nlines(7,7)
  ifthen(A("0"),B("1"))
  ifthen(C("0"),D("1"))
  start("1111111")
  start("1000001")
  useonly(odd)
  afterinsert(odd,even)
  end("1111111")
}
```

Drobnou modifikací se lze dostat na jakýkoliv požadovaný rozměr bludiště. Aby mohl být použit předpoklad základní matice bludiště, je nutné uvažovat liché rozměry bludiště.

C. *Shrnutí*

Generováním bludiště jsme ukázali další možnou oblast v generování pomocí námi navržené architektury generátoru. Vygenerovaná bludiště jsme použili jako vstup pro verifikaci řídicí jednotky robota, která hledá cestu v bludišti. V rámci práce byla řídicí jednotka ověřována na správnou funkcionalitu pro různé tvary a rozměry bludišť. Tento přístup byl opublikován na konferenci [8].

VII. Budoucí Práce

V rámci budoucí práce pracujeme na zobecnění v samotném generování stimulů, které bude možné aplikovat pro různé systémy. Tato práce zahrnuje modifikaci architektury a popsání stimulů pomocí existujícího aparátu. Jako vhodný aparát se jeví použití gramatických systémů, které principiálně již částečně využíváme v definici vstupní struktury generátoru. Určité omezující podmínky ale budou nadále nutné, aby byla zajištěna validnost vygenerovaného stimulu.

V současné funkcionalitě generátor podporuje několik omezení pro generování programů pro procesory typu RISC, VLIW, softwarově implementovanou odolnost proti poruchám a bludiště. Zobecněním vstupních popisů by proto práce nabyla jiného rozměru a našla by uplatnění v dalších systémech v řadě oblastí.

VIII. Závěr

V práci jsme ukázali architekturu námi navrženého generátoru stimulů, která je založena na dvou vstupních strukturách. Pomocí těchto struktur jsme byli schopni nadefinovat popisy pro generování assemblerovských programů pro procesor typu RISC. Takovýto program jsme byli rovněž schopni zabezpečit pomocí softwarově implementované odolnosti ztrojením instrukcí a přidáním voličů pro vyhodnocení majority v paměťových prvcích procesoru. V poslední části práce jsme se věnovali generováním bludišť pro řídicí jednotku robota.

Ukázali jsme široké použití generátoru stimulů v různých oblastech použití a nastínili naše kroky v další výzkumné činnosti. Naše výsledky jsou řádně zdokumentovány díky řadě publikací na konferencích ve světě. V další činnosti pracujeme na zobecnění popisu a samotného generování stimulů.

Poděkování

Tato práce byla podpořena Ministerstvem školství, mládeže a tělovýchovy z Národního programu udržitelnosti (NPU II); projekt IT4Innovations excellence in science (IT4I XS - LQ1602), ARTEMIS JU na základě grantové dohody č. 641439 (ALMARVI) a projektu Vysokého učení technického v Brně FIT-S-14-2297.

Reference

[1] Roy, S.; Ramesh, S.: Functional verification of system on chips - practices, issues and challenges. In Proceedings of ASP-DAC 2002, 2002, s. 11-13, doi:10.1109/ASPDAC.2002.994873.

[2] Belkin, V.; Sharshunov, S.: ISA Based Functional Test Generation with Application to Self-Test of RISC Processors. In Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE, duben 2006, s. 73-74, doi:10.1109/DDECS.2006.1649575.

[3] Hudec, J.: An effient technique for processor automatic functional test generation based on evolutionary strategies. In Proceedings of the ITI 2011, 33rd International Conference on Information Technology Interfaces, červen 2011, ISSN 1330-1012, s. 527-532.

[4] Podivínský, J.; Čekan, O.; Šimková, M.; Kotásek, Z.: The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications. In: 17th Euromicro Conference on Digital Systems Design. Verona: IEEE Computer Society, 2014, s. 312-319. ISBN 978-1-4799-5793-4.

[5] Codasip. (2013) Codasip - codix-risc. [Online]. Dostupné z: www.codasip.com/products/codix-risc/

[6] Podivínský, J.; Čekan, O.; Šimková, M.; Kotásek, Z.: The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications. Microprocessors and Microsystems. Amsterdam: Elsevier Science, 2015, roč. 39, č. 8, s. 1215-1230. ISSN 0141-9331.

[7] Čekan, O.; Podivínský, J.; Kotásek, Z.: Software Fault Tolerance: the Evaluation by Functional Verification. In: Proceedings of the 18th Euromicro Conference on Digital Systems Design. Funchal: IEEE Computer Society, 2015, s. 284-287. ISBN 978-1-4673-8035-5.

[8] Podivínský, J.; Čekan, O.; Lojda, J.; Kotásek, Z.: Verification of Robot Controller for Evaluating Impacts of Faults in Electro-mechanical Systems. In: Proceedings of the 19th Euromicro Conference on Digital Systems Design. IEEE Computer Society, 2016, přijato k publikaci.