IV113 Validace a verifikace

Organizace kurzu, motivace, přehled technik

Jiří Barnat

Validace versus verifikace

Proces validace

- Ověření, že systém nabízí chtěnou/požadovanou funkcionalitu.
- Ověření, že model reality je v požadovaných aspektech věrný.

Proces verifikace

- Ověření, že poskytované funkce systému pracují dle očekávání.

Anotace

Cílem předmětu je seznámit studenty s

- Základním rozdělením verifikačních metod.
- Technikami testování.
- Vybranými technikami formální verifikace.
- Využitím řešičů SAT a SMT v oblasti formální verifikace.

Organizace kurzu

Místo a čas konání (podzim 2016)

- středa 16:00-17:40, A318

Ukončení předmětu

- Ústní zkouška
- Hodnocení A navíc vyžaduje vypracování všech domácích úloh.

Zdroje

Testování

- http://www.testingeducation.org/
- Cem Kaner, James Bach, Bret Pettichord:
  Lessons Learned in Software Testing
- Cem Kaner, Jack Falk, Hung Quoc NGuyen:
  Testing Computer Software

Formální verifikace

- Edmund M. Clarke, Orna Grumberg, Doron Peled:
  Model checking
- D. A. Peled:
  Software Reliability Methods
- ... a další ...

Motivace V&V

Vývoj kvalitních systémů

Marketing

- Lepší kredit a důvěra u zákazníků.
- Osobní uspokojení a prestiž.

Ekonomické dopady nekvalitního produktu

- Náklady spojené s nápravou chyb v době vývoje produktu.
- Náklady spojené s podporou po dodání zákazníkovi.
- Náklady na soudní výlohy v případě právních dopadů.
- ...

Vývoj kvalitních produktů

- Chyby ve vyvíjeném produktu snižují jeho kvalitu.
- Procedury pro detekci a prokázání absence chyb.
- **Validace a Verifikace**

Therac-25 (1985)

- Přistroj pro léčbu ozařováním
- Therac-25 ozařoval ve 2 režimech
  1) několika vteřinové ozařování o malé intenzitě
  2) násobný výboj o velké intenzitě
- při rychlém zadávání příkazů na klávesnici došlo k race-condition
- 5 lidí zemřelo po léčbě, kdy byli ozáření po dobu několika vteřin velkou intenzitou záření

Nejdražší/nejzávažnější počítačové chyby

Ariane 5 (1996)

- nosná raketa ESA, byla zničena 40 vteřin po startu
- následník Ariane 4, používala software z Ariane 4
- A5 dosahovala při startu 5x vyššího zrychlení než A4
- hodnoty se dostaly mimo očekávaný rozsah a při konverzi 64-bitového desetinného čísla na 16-bitové celé došlo k aritmetickému přetečení
- rutina, která měla tuto výjimku ošetřit, byla z důvodů efektivity kódu vypnuta
- A5 se sebe-zničila
- `http://www.youtube.com/watch?v=kYUrqdUyEpI`

Mars Climate Orbiter (1999)

- měl obíhat Mars ve výšce 150km
- klesl do výšky 57 km, kde byl zničen
- důvod: 2 spolupracující jednotky navigačního podsystému pracovaly jedna v metrických jednotkách a druhá v imperiálních

Nejdražší/nejzávažnější počítačové chyby

Výpadek sítě AT&T (1990)

- přetížil se jeden uzel sítě následkem čehož se "rebootoval"
- před každým "rebootem" však uzel informoval sousední uzly
- zpráva však díky softwarové chybě způsobila "reboot" adresáta
- výsledek: celá síť AT&T se s frekvencí 6 vteřin kompletně restartovala
- fix: inženýři nahráli předchozí verzi SW

Výpadek proudu v Severní Americe (2003)

- ztráty 6 miliard USD
- 50 milionů lidí bez dodávky elektřiny
- důvod: race condition

Nejdražší/nejzávažnější počítačové chyby

Pentium FDIV bug (1994)

- nesprávné výsledky při dělení desetinných čísel
- způsobeno nevyplněnou tabulkou v matematickém koprocesoru
- celková škoda 500 milionů USD

2001: Vesmírná odysea

- průzkumná mise k Saturnu
- palubní počítač měl nekonzistentní specifikaci svých úkolů
  - nesdělit členům posádky pravý účel cesty
  - nezatajovat posádce žádné informace
- počítač dospěl k logickému řešení nekonzistence

Nejdražší/nejzávažnější počítačové chyby

Pentium FDIV bug (1994)

- nesprávné výsledky při dělení desetinných čísel
- způsobeno nevyplněnou tabulkou v matematickém koprocesoru
- celková škoda 500 milionů USD

2001: Vesmírná odysea

- průzkumná mise k Saturnu
- palubní počítač měl nekonzistentní specifikaci svých úkolů
  - nesdělit členům posádky pravý účel cesty
  - nezatajovat posádce žádné informace
- počítač dospěl k logickému řešení nekonzistence
- **vyvraždil posádku**

Souvislost s oborem “PDS” @ FI

Pozorování

- Netriviální množství skutečně zákeřných a drahých chyb v systému vzniká v důsledku nespecifikovaných či nejasně specifikovaných spojení jednotlivých částí systému a následně pak zpracováním neočekávaných vstupních dat.

Paralelní/distribuované počítačové systémy

- Distribuované webové aplikace
- Systémy vystavěné z komponent
- Vícevláknové aplikace
- Řídící a operační systémy
- Bezpečnostní a jiné komunikační protokoly
- Vestavné systémy (Embedded systems, HW-SW co-design)
- ...

Kontext IV113 na FI

Verifikace

- IV022 Návrh a verifikace algoritmů
- IA159 Formal Verification Methods
- IA040 Modální a temporální logiky procesů
- IV101 Seminář z verifikace

Formální analýza, Modelování, Simulace, ...

Vývoj SW

Výzkum na FI související s verifikací

- Laboratoř paralelních a distribuovaných systémů (ParaDiSe)
- Laboratoř formálních metod (ForMeLa)

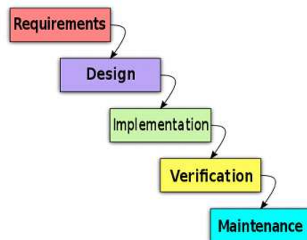
Sekce

V&V ve vývojových modelech

Fáze vývoje systému

Fáze

- Sběr požadavků (Requirements)
- Design
- Implementace
- Testování
- Běh u zákazníka a údržba systému



Vztahy a produkty jednotlivých fází

- Fáze logicky navazují a probíhají v daném pořadí.
- Fáze nemusí být ve vývojovém modelu konkrétního výrobce explicitně identifikovatelné.
- Nezávislé činnosti mohou probíhat "mimo pořadí", např. vývoj testovacího prostředí může předcházet samotné implementaci.

Poučka o ceně za nekvalitní produkt

Pozorování

- Čím delší je doba mezi okamžikem zanesení chyby do systému a jejím odhalením, tím je případná realizace nápravy chyby nákladnější.

Relativní cena nápravy chyby

Chyba zanesena	Chyba odhalena				
	Req.	Design	Impl.	Test.	U zákazníka
Requirements	1	3	5	20	100
Design		1	10	25	100
Implementace			1	4	50
Testování				1	10

Testování v modelech vývoje software

Vodopád/V-model

- Testování je poslední fáze.
- Ve V-Modelu je Testování hierarchicky členěno
  - Testování modulů (jednotek) systému
  - Integrační testování
  - Systémové testování
  - "Acceptance" testování

Agilní metody (iterativní metody vývoje)

- Ranné testování
- Vývoj testů často předchází samotné implementaci
- Dominuje testování nových modulů
- Integrační a systémové testování utlumeno
- "Acceptance testy" při vstupu do další iterace

Techniky pro zvyšování kvality

Procedurální postupy

Snížení rizika zanesení chyb

- Povinnost formalizace požadavků.
- Model Driven Development (automatické generování kódu).
- Programování ve dvou.
- "Štábní" kultura a disciplína.

Snížení rizika neodhalení zanesených chyb

- Předepsané verifikační kroky ve vývoji systému.
- Požadavky na výsledky verifikačních kroků.

Snížení rizika opomenutí nápravy odhalených chyb

- Nástroje pro sledování nalezených chyb (Bugzilla, Trac).

Metody zvyšování kvality produktu

Neformální metody

- Technická revize
- Ladění a desk-checking
- Simulace
- Runtime analýza
- Testování

Formální metody

- Deduktivní verifikace (Dokazování)
- Statická analýza a Abstraktní Interpretace
- Symbolická exekuce
- Model Checking

Technická revize

Technická revize

- Člověkem prováděná analýza daného artefaktu za určitým předem stanoveným cílem.
- Revize se (ideálně) účastní několik osob v různých rolích.
- Hlavní role jsou moderátor, sekretář a tým recenzentů.

Studované Artefakty

- Popis specifikace, část zdrojového kódu, výsledek provedeného testu, ...

Možné cíle revize

- Detekce chyb či jiných důvodů nízké kvality produktu.
- Nesoulad se standardem či se specifikací, ...

Průběh technické revize

Sezení

- Samotná revize probíhá na sezení.
- Jednotlivé námitky recenzentů jsou postupně prezentovány na sezení a diskutují se s autory analyzovaného artefaktu a ostatními recenzenty.
- Cílem revize není hledat řešení na nalezené problémy, pouze poukázat na jejich existenci. Po dobu sezení se analyzovaný artefakt nesmí měnit.
- Nalezené problémy jsou sekretářem zaznamenány ve zprávě o revizi.

Mimo sezení

- Před sezením by měli recenzenti mít dostatek času na analýzu revidovaného artefaktu.
- Po sezení je možné stanovit priority nalezených problémů a případně stanovit termín další revize.

Varianty technické revize

Walkthrough

- Autor revidovaného artefaktu je přítomen sezení.
- Autor moderuje sezení a prochází analyzovaný artefakt.

Inspekce

- Opakování walkthrough dokud není dosaženo přijatelně nízkého počtu nově objevených problémů.
- Každé sezení končí rozhodnutím, zda je třeba další iterace.

Audit

- Nezávislá revize provedená externí skupinou.
- Objektivní, úplná, systematická a dokumentovaná.
- Cílem auditu je sběr podkladů, na základě kterých lze argumentovat, že revidovaný artefakt splňuje stanovená kritéria auditu.

Výhody a nevýhody technické revize

Výhody revizí

- Aplikovatelné na artefakty, které nelze podrobit automatizované analýze.
- Lidská kreativita v identifikaci problémů.

Nevýhody revizí

- Úspěšně provedená revize negarantuje skutečný stav věci.
- Drahá metoda.

Automatizovaná podpora

- Revize je manuální činnost, nástrojová podpora je zejména procedurální, tj. možná pro roli sekretáře.

Desk-Checking

- Metoda spojená s kontrolou algoritmických aspektů vyvíjeného produktu.
- Označuje aktivitu, kdy autor mentálně vykonává jednotlivé kroky algoritmu nad konkrétní sadou vstupních dat.
- Walkthrough, ve kterém si je autor sám sobě recenzentem.

Ladění

- Desk-Checking s využitím softwarové podpory – debugger.
- Typicky se používá pro odhalení původu neočekávaného chování.

Simulace

Simulace

- Imitace chování vyvíjeného produktu s využitím modelu.
- Pozorování jednoho běhu systému za účelem potvrzení očekávaného chování, a nebo pro účely zjištění nových charakteristik chování systému v dané situaci.

Vlastnosti techniky

- Provedení vyžaduje imitační prostředí a nástroje.
- Model musí popisovat behaviorální složku produktu.
- Typické pro vývoj HW částí vestavných systémů.
- Simulací nelze prokázat korektnost, pouze přítomnost chyby.

Runtime analýza

- Pozorování chování systému v době jeho skutečného běhu.
- Umožňuje zachytit aspekty, které se neprojevují přímo na vnějším chování (obtížně se testují).
- Realizuje se vložením pozorovatelů přímo do spustitelného kódu (augmentace).

Typické cíle runtime analýzy

- Nekorektní použití dynamicky alokované paměti.
- Omezené možnosti detekce Race-Condition a nesprávného zamykání unikátních zdrojů.
- Detailní profilování výkonu aplikace.
- Kontrola invariantů (assertů), kontrola platnosti vstupních a výstupních podmínek a jejich vztahů.

Testování

- Pozorování chování výsledného produktu, nebo části výsledného produktu nad vybranou množinou vstupních dat.
- Realizováno s různými cíli, například s cílem detekovat chyby.
- Nejčastěji používaná technika verifikace, přestože neschopnost prokázat chybu neznamená, že se v produktu chyba nevyskytuje.

Deduktivní verifikace

Dokazování

- Matematické prokazování vlastností algoritmů.
- V případě úspěšného sestrojení důkazu je dokázaná vlastnost algoritmu skutečně garantována.

Nevýhody

- Nelze algoritmizovat (problém zastavení je nerozhodnutelný).
- Existují vlastnosti programů, které nelze dokázat ani vyvrátit.
- Vyžaduje vysoce kvalifikovaný personál.
- Pokud se nepodaří dokázat nějakou vlastnost programu, není jasné, zda tuto vlastnost program má, či nemá, a pokud ji nemá, není žádné vodítko k důvodu, proč tomu tak je.
- Neadresuje chyby vzniklé implementací.
- Obtížné nasazení nad rámec algoritmizace.

Statická analýza programů

- Zjišťování vlastností programů na základě jejich popisu v daném programovacím jazyce, tj. bez nutnosti skutečného spuštění programu.
- Zjištění vlastností jednotlivých uzlů Control-Flow grafu.

Typické použití – překladač

- Detekce mrtvého kódu.
- Detekce použití nedefinovaných proměnných.
- Typová kontrola.

Výhody a nevýhody

- Jednostranně přesné výsledky (neodhalené $\times$ falešné chyby).
- Pokrývá pouze vlastnosti programů, které jsou nezávislé na konkrétních hodnotách vstupních proměnných.
- Aplikovatelná na rozsáhlé projekty.

Symbolická exekuce

- Vykonávání programu dle Control-Flow grafu s tím, že hodnoty vstupních proměnných jsou označeny a manipulovány symbolicky (constraint solving).
- Pro jeden konkrétní průchod Control-Flow grafem lze postupně vystavět omezení na hodnoty vstupních proměnných, které k tomuto průchodu vedly – **podmínka cesty**.

Typické použití a omezení metody

- Detekce porušení invariantů, assertů, ...
- Syntéza hodnot vstupních proměnných, které vynutí vykonání programu určitým způsobem.
- Počet možných průchodů roste exponenciálně s počtem větvení (path-explosion), to omezuje velikost projektu, na který lze symbolickou exekuci smysluplně aplikovat.

Model Checking

- Verifikovaný systém se abstrahuje do modelu s konečně stavovou sémantikou, pro který se **algoritmicky** prokáže platnost dané vlastnosti systému.
- Prokázání je realizováno prohledáním celého stavového prostoru modelovaného systému.
- Dává garanci, nebo vrací protipříklady.

Omezení

- Nevhodné pro dokazování významových funkcí programů.
- Velikost stavového prostoru roste exponenciálně vzhledem k velikosti domén vstupních proměnných a počtu paralelně prováděných procesů.
- Aplikovatelné pouze na problémy do určité velikosti.

Ukázka runtime verifikace

(valgrind)

Valgrind

Nástroj pro ladění a profilování linuxových programů.

Jak pracuje

- Kód programu se přeloží do HW-neutrálního formátu (intermediate representation, IR)
- Kód v IR se označkuje, tj. doplní se kód, který pozoruje původní kód.
- Označkovaný IR se přeloží do proveditelného kódu dle odpovídající architektury.
- Původní program je vykonáván nativním HW, ale je za běhu pozorován.

Výhody

- Různé moduly valgrindu, mohou provádět různá pozorování.
- Kód je nativně prováděn nikoliv simulován.
- Aplikovatelné na rozsáhlé projekty, např. OpenOffice.

Valgrind – memcheck

Co umí detekovat:

- Neinicializovanou paměť.
- Přístup před, či za alokovaný blok.
- Nepárové volání `malloc`/`free` a `new`/`delete`.
- Pokus o uvolnění nealokované paměti.
- Přístup do již uvolněných paměťových bloků.
- Přístup na neoprávněná místa zásobníku.
- Předávání neinicializovaných hodnot systémovým voláním.
- Nesprávné použití `memcpy`.

Valgrind – použití, callgrind, kcachegrind

Spuštění aplikace:

```
my_app arg1 arg2
```

Valgrind a memcheck:

```
valgrind my_app arg1 arg2
```

valgrind a callgrind:

```
valgrind -tool=callgrind my_app arg1 arg2
```

analýza call-grafu:

```
kcachegrind callgrind.out.xxx
```

Testování

http://www.testingeducation.org/BBST/

Testování

Technické vyšetřování testovaného produktu prováděné za účelem poskytnutí kvalitativních informací zainteresovaným subjektům.

Technické

- experimentování
- logika a matematika
- modelování
- SW nástroje pro samotné testování
- pomocné SW nástroje

vyšetřování

- organizované a důkladné hledání
- sebekritické a vyzývající

Testování

testovaného produktu

- samotný kód
- neoddělitelná data
- dokumentace a specifikace
- HW
- a dalších věcí, které jsou součástí dodávky zákazníkovi

prováděné za účelem poskytnutí kvalitativních informací

- viz dále

zainteresovaným subjektům.

- někdo, kdo má zájem na tom, aby testování bylo smysluplné (šéf testovacího týmu)
- někdo, kdo má zájem na tom, aby produkt byl úspěšný (manažer produktu)
- čí zájem je možné/žádoucí ignorovat

Fundamentální otázky související s testováním

Mise

- Proč testujeme? Co se snažíme testováním dosáhnout?

Strategie

- Jakým stylem máme postupovat, abychom dosáhli cíle?

Problém orákula

- Jak vlastně poznáme, že test proběhl úspěšně?

Neúplnost

- Uvědomujeme si, že testováním nelze potvrdit absenci chyby?

Míra

- Kolik % z testovacího plánu je odtestováno?
- Jaká je míra naplnění mise testování?

Mise testování

Nejčastější mise

- Detekce chyb.
- Identifikace faktorů snižující kvalitu produktu.

Jiné cíle testování

- Vytvoření podkladů pro rozhodnutí, zda je produkt již dost dobrý na to, aby byla zahájena jeho distribuce.
- Nakolik se produkt liší (například v ovládání) od produktů momentálně dostupných na trhu?
- Posouzení, zda produkt pokrývá požadavky zadavatele.
- Je provázání souvisejících funkcí software logické a dostatečné?
- ...

Mise testování

Jiné cíle testování – pokračování

- Podpořit/nabourat manažerská rozhodnutí čísly.
- Odhadnout cenu nabízené podpory produktu po jeho uvolnění.
- Ověřit kompatibilitu a interoperabilitu vůči jiným produktům.
- Nalézt bezpečné scénáře použití produktu.
- Potvrdit soulad se specifikací.
- Certifikovat daný standard.
- Minimalizovat rizika vedoucí k právním dopadům.
- Vyhodnotit produkt pro jiného zadavatele.
- ...

Sekce

Strategie testování

Strategie

Strategie je plán, jak naplnit misi testování v kontextu konkrétního projektu.

Příklad: Uvažme program, který provádí výpočty ala tabulkový procesor v následujících 4 kontextech

a) počítačová hra
b) rané stádium vývoje komerčního produktu (mise: identifikace problémových míst, první zpětná vazba programátorům)
c) pozdní stádium vývoje komerčního produktu (mise: pomoci projektovému manažeru rozhodnout, zda je produkt hotov)
d) ovladač ozařovacího zařízení na léčbu rakoviny

Otázka:

- Budeme postupovat v různých případech stejně?

Příklad

Faktory ovlivňující výběr strategie

- Jaká je mise v jednotlivých kontextech?
- Jak agresivně budete hledat chyby?
- Jaké chyby jsou méně důležité než jiné a proč?
- Jak důkladně budete dokumentovat proces testování?

Diskuze

- Předpokládejme, že dle specifikace má program vstupní pole, na kterém očekává číselné hodnoty (program provádí výpočty). Má smysl testovat chování programu, pro situace, kdy na vstupu nejsou čísla, ale písmena (situace mimo specifikaci).

Problém orákula

Definice Orákula

Orákulum (v kontextu testování) je princip nebo mechanismus, kterým jsme schopni rozeznat, že něco není tak, jak by mělo být , tj. detekovat chybu.

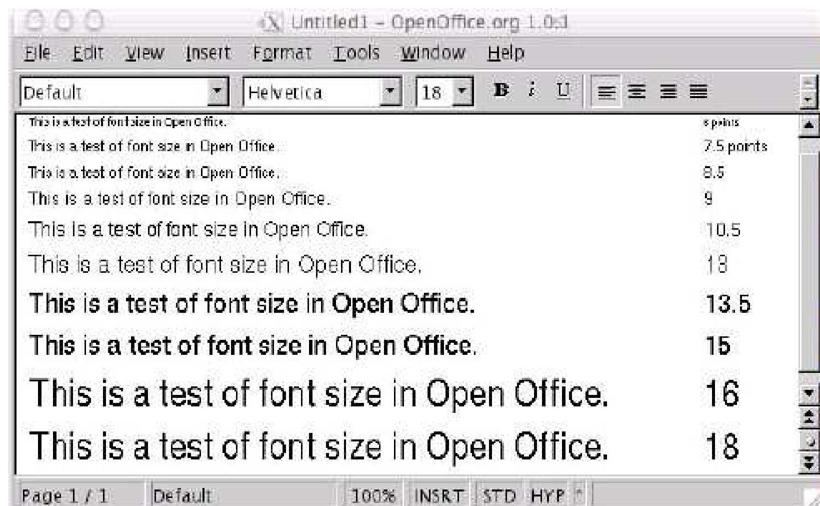
Fakta

- Tvrdí-li tester, že test neprokázal nedostatky, neznamená to, že je produkt v daném směru bezchybný.
- Výsledek každého testu může být “test proběhl v pořádku”, záleží pouze na volbě orákula.

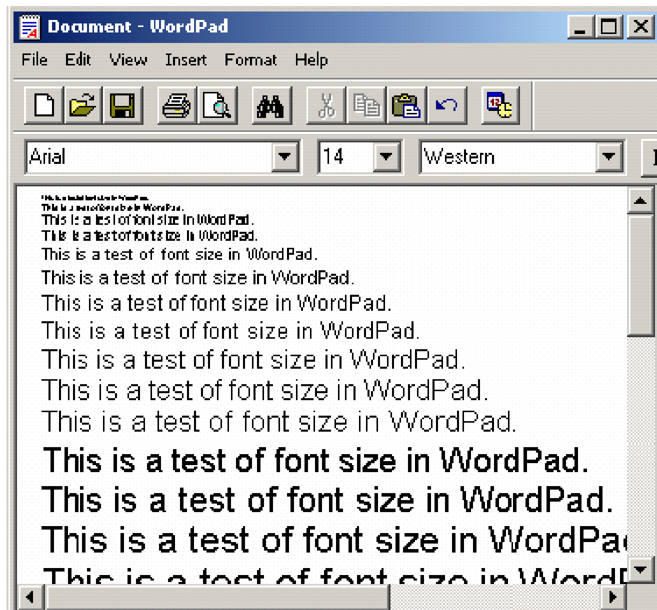
Příklad

- Fungují správně velikosti písem v programech OpenOffice, WordPad, Word?

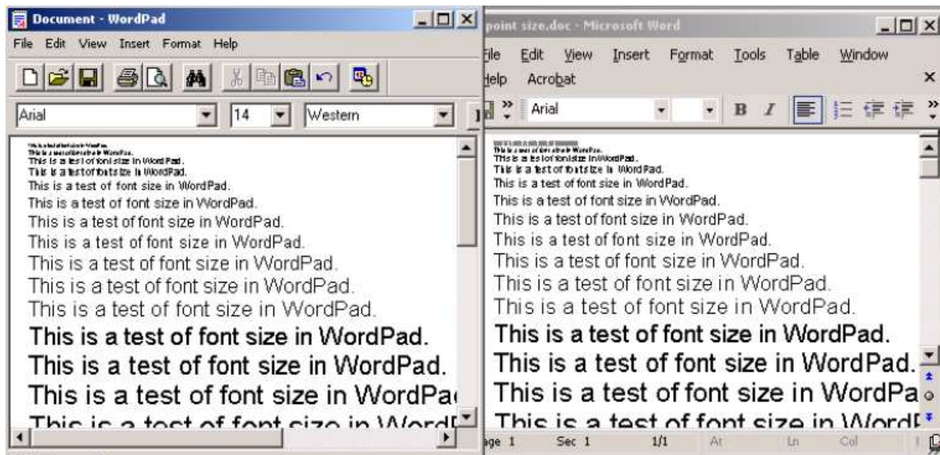
Příklad – OpenOffice 1.0



Příklad – Word PAD



Příklad – WordPad versus MS Word



Příklad – WordPad versus MS Word (zvýrazněno)

Příklad – Rozhodnutí

Otázky

- Je pozorovaný rozdíl velikostí bug ve WordPadu?
- Je pozorovaný rozdíl velikostí bug v MS Wordu?
- Je pozorovaný rozdíl velikostí vůbec bug?

Možné závěry

- Nevíme, jestli jsou velikosti písma správně, ale při porovnávání WordPadu a MS Wordu, raději věříme MS Wordu.
- Pro WordPad není třeba lpět na přesných standardech typografie.
- Pro WordPad je pozorovaný rozdíl (možná) bug, ale není to problém.

Příklad – Risk-based testing

Možný (pragmatický) pohled na věc

- Je/Není to bug? $\Longrightarrow$ Je/Není to problém?
- Je třeba znát kontext, do kterého bude testovaný produkt zasazen, případně metriky podle kterých produkt posuzuje zákazník.
- Zjednodušení procesu testování za cenu jistého rizika.

Zjednodušení

- Vynechání testů, které zřejmě neodhalí žádné problémy.
- Vynechání testů, které zřejmě odhalí pouze nezajímavé problémy.

Příklad – Kritéria posuzování

Kolik víme o typografii?

- Definice bodu (point) je nejasná.
  (`http://www.oberonplace.com/dtp/fonts/point.htm`)
- Absolutní velikost písma není lehké změřit.
  (`http://www.oberonplace.com/dtp/fonts/fontsize.htm`)

Nejasnost a náročnost posouzení výsledku testu

- Jak přesně musí velikost být v souladu se standardem, abychom prohlásili, že je velikost písma korektní?
- Kompletní shromáždění faktů a jejich vyhodnocení je příliš náročné/zdlouhavé.
- Při rozhodování o výsledku testu se používají heuristiky.

Problém orákula – rozhodovací heuristiky

Rozhodovací heuristika

- Je postup, který umožní zjednodušit a snáze vyřešit problém rozhodnutí.
- Neobsahuje žádnou skrytou znalost.
- Rada/návod/doporučení na základě kontextu.
- Negarantuje správnost rozhodnutí.
- Různé heuristiky mohou vyústit ve vzájemně rozporná rozhodnutí.

Nevýhody

- Při nesprávném použití mohou být na škodu věci.
- V obecné rovině jsou heuristiky subjektivní.

Konzistence jako heuristika

Konzistence

- Dobrá heuristika pro rozhodování o výsledku testu.

Konzistence s

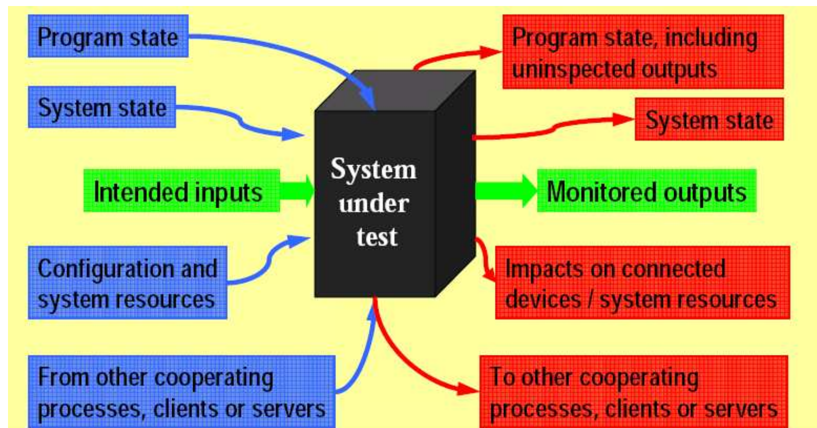
- ostatními funkcemi produktu, porovnatelnými produkty, historií, image producenta, různými prohlášeními a reklamou, specifikací či standardy, očekáváním uživatelů, účelem produktu, . . .

Výhody

- Je dostatečně objektivní.
- Je snadno popsatelná (bug report).

Širší kontext testu SW produktu

Důvody pro selhání produktu jsou často nad rámec vstup-výstupního chování produktu. Je nutné produkt testovat i nad tento rámec, což v kontextu problému orákula znamená rozhodovat i dle nepřímých výstupů.



Nedokonalost v rozhodování

Slepota z nepozornosti

- Lidský tester neuváží do rozhodnutí to, na co nedává pozor (to, co nesleduje).
- Mechanický tester neuváží do rozhodnutí to, co mu nebylo řečeno, aby do rozhodovacího procesu zahrnul.

Princip neurčitosti

- Zapojením mnoha diagnostických prostředků se zkresluje chování testovaného produktu.

Důsledek

- V rámci testu nelze z praktického hlediska sledovat všechny možné aspekty.

Problém orákula a automatizace testování

Motivace

- Automatizace procesu vylučuje lidské chyby.
- Automatizací získáme opakovatelnou proceduru.
- Větší rychlost provádění jednotlivých testů.

Problém automatizace

- Je třeba (mimo jiné) automatizovat rozhodovací proces.
- Umíme to? (Částečně)

Standardní způsob automatizace rozhodovacího procesu

- Definujeme zdroj/soubor očekávaných výstupů, pokud možno včetně nepřímých výstupů.
- Příklad: MS Word je zdrojem výstupů pro MS WordPad
- Test je úspěšný pokud výstup testovaného produktu odpovídá (jedna k jedné) výstupu dle souboru očekávaných výstupů.

Problémy automatizovaného rozhodovacího procesu

Problém definice shody

- Jak uložím výstup MS Wordu, tak abych ho mohl porovnávat s výstupem z testovaného Word Padu?
- Je přijatelná 99% shoda? Jak definovat % shody?

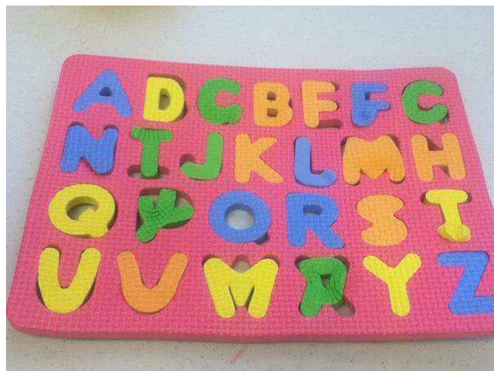
Falešná hlášení o chybách

- Použitím neaktuálních testů.
- Důsledek zjednodušování rozhodovacího algoritmu.

Nenalezené chyby

- Shodná chyba v souboru očekávaných výstupů.
- Neúplnost zdrojových dat, viz slepota z nepozornosti.

Lidský faktor v interpretaci orákula



Lidský faktor ...

- Když má tester vhodnou motivaci a je šikovný, tak ani dobré orákulum nepomůže k bezchybné interpretaci výsledku testu.

Sekce

Metody míry testování

Pokrytí jako metoda míry

Pokrytí

- Množina testováním prověřených entit programu (entity: řádky kódu, podmínky, vstupní data, větve programu, ...)
- Identifikaci netestovaných částí kódu.

Pokrytí jako míra

- Možný testovací plán je dosáhnout daného procenta pokrytí výsledného produktu.
- Procento pokrytí stávajícími testy je pak možné chápat jako míru jak daleko jsme v testovacím plánu.
- Pro manažery a vedení projektu je dobré umět změřit kolik z celkového objemu testování bylo provedeno, případně kolik zbývá.

Pokrytí jako metoda míry – nevýhody

Principiální nedostatky pokrytí

- Nepostihne zajímavá vstupní data.
- Nepostihne kód, který není součástí produktu (knihovny, ovladače, atd.)
- Netestuje produkt v kontextu běžícího OS systému (například možné okamžiky, ve kterých dochází k HW/SW přerušení a vykonání odpovídající obslužné rutiny.)

Používání pokrytí jako míry

- Úplné pokrytí negarantuje kvalitu produktu.
- Stimuluje tendenci preferovat kvantitu před kvalitou.
- Zavádějící, navozuje falešný pocit bezpečí.

Pokrytí jako metoda míry – nevýhody

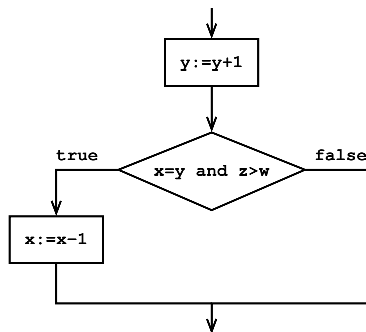
Příklad

```
Input A         // program accepts any
Input B         // integer into A and B
Print A/B
```

Pozorování

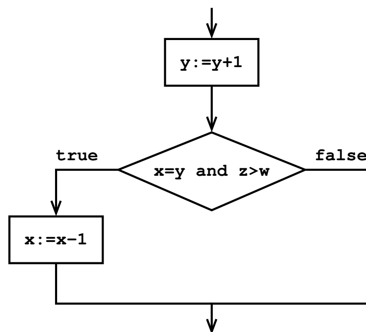
- Snadno dosáhneme úplného pokrytí.
- Například:
  input: 2,1
  output: 2
- Tento test neodhalí skrytý "bug"!

Kritéria pokrytí pro Control-Flow grafy



Existují různá kritéria pokrytí Control-Flow grafu.

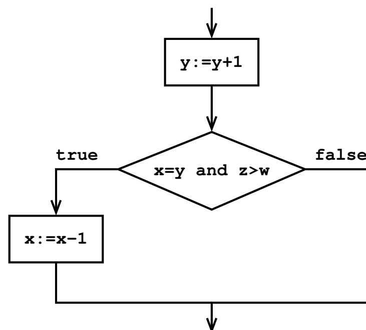
Kritéria pokrytí pro Control-Flow grafy



Statement coverage – pokrytí výrazů

- Každý výraz (přiřazení, vstup, výstup, podmínka) je proveden alespoň v jednom testu.
- Sada testů pro dosažení pokrytí: $(x = 2, y = 1, z = 4, w = 3)$

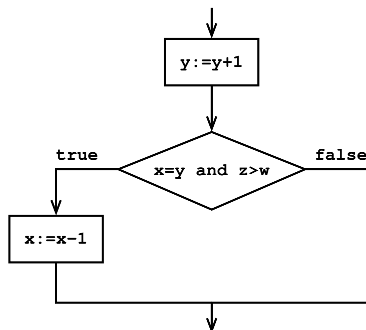
Kritéria pokrytí pro Control-Flow grafy



Edge coverage – pokrytí hran

- Každá hrana CF grafu je provedena alespoň v jednom testu.
- Sada testů pro dosažení pokrytí: $(x = 2, y = 1, z = 4, w = 3)$, $(x = 3, y = 3, z = 5, w = 7)$

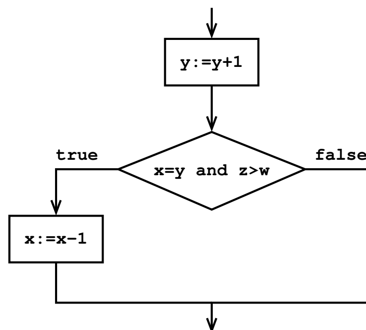
Kritéria pokrytí pro Control-Flow grafy



Condition coverage – pokrytí podmínek

- Každá podmínka je Boolovskou kombinací elementárních podmínek, například $\mathtt{x} < \mathtt{y}$ nebo $\mathtt{even}(\mathtt{x})$.
- Pokud je to možné, každá elementární podmínka je alespoň v jednom testu vyhodnocena na TRUE a alespoň v jednom testu vyhodnocena na FALSE.

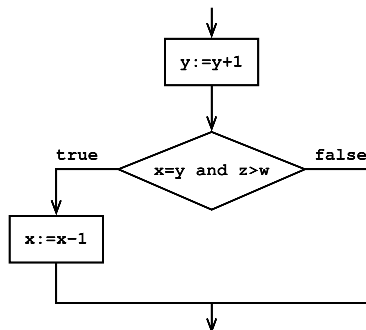
Kritéria pokrytí pro Control-Flow grafy



Condition coverage – pokrytí podmínek

- Sada testů pro dosažení pokrytí: $(x = 3, y = 2, z = 5, w = 7)$, $(x = 3, y = 3, z = 7, w = 5)$
- V obou případech je podmínka ve výrazu IF vyhodnocena na FALSE.

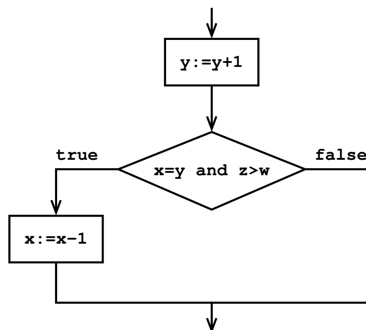
Kritéria pokrytí pro Control-Flow grafy



Edge/Condition coverage – pokrytí hran a podmínek

- Pokrytí proveditelných hran a podmínek zároveň.
- Sada testů pro dosažení pokrytí: $(x = 2, y = 1, z = 4, w = 3)$, $(x = 3, y = 2, z = 5, w = 7)$, $(x = 3, y = 3, z = 7, w = 5)$
- Je uvedená sada testů nejmenší možná?

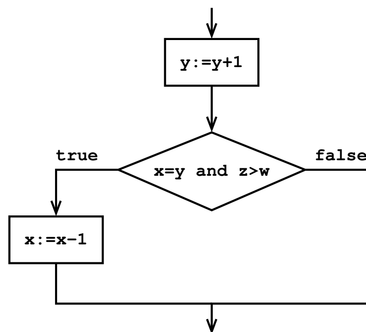
Kritéria pokrytí pro Control-Flow grafy



Multiple condition coverage – násobné pokrytí podmínek

- Každá Boolovská kombinace hodnot TRUE/FALSE, která se může objevit v nějaké rozhodovací podmínce, se musí objevit v provedení alespoň jednoho testu.

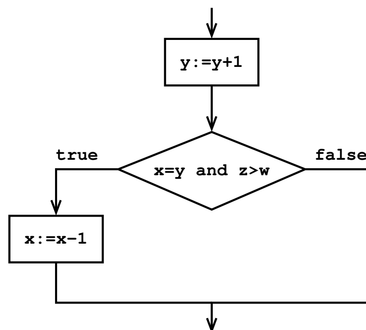
Kritéria pokrytí pro Control-Flow grafy



Multiple condition coverage – násobné pokrytí podmínek

- Sada testů pro dosažení pokrytí: $(x = 2, y = 1, z = 4, w = 3)$, $(x = 3, y = 2, z = 5, w = 7)$, $(x = 3, y = 3, z = 7, w = 5)$, $(x = 3, y = 3, z = 5, w = 6)$
- Exponenciální růst počtu testů.

Kritéria pokrytí pro Control-Flow grafy



Path coverage – Pokrytí cest

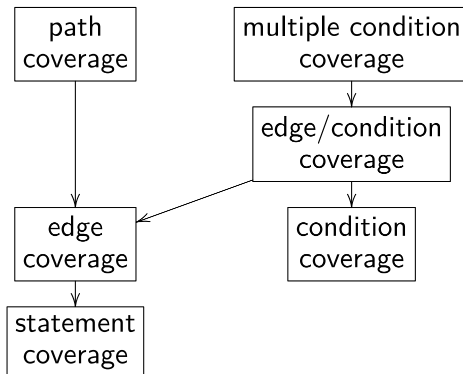
- Každá proveditelná cesta je provedena alespoň v jednom testu.
- Počet cest je obrovský, přítomnost cyklu, může vyústit v nekonečný počet cest.

Hierarchie kritérií pokrytí pro Control Flow graf

- Kritérium A **zahrnuje** kritérium B, značeno A $\rightarrow$ B, pokud dosažením pokrytí typu A také garantuje pokrytí typu B.

Hierarchie kritérií pokrytí pro Control Flow graf

- Kritérium A **zahrnuje** kritérium B, značeno A $\rightarrow$ B, pokud dosažením pokrytí typu A také garantuje pokrytí typu B.



Pokrytí cyklů

Pokrytí a průchody cyklem

- Všechna zmíněná kritéria (s výjimkou pokrytí cest) neřeší počet průchodů tělem cyklu.
- V případě existence zanořených cyklů je systematické testování různých způsobů průchodů cykly komplikované.

Ad hoc strategie pro testování cyklů

- Prověř případ, kdy se tělo cyklu přeskočí.
- Prověř případ, kdy se tělo cyklu provede přesně jednou.
- Prověř případ, kdy se tělo cyklu provede očekávaným počtem opakování.
- Pokud je známa hranice n na počet provedení těla cyklu, prověř případ, kdy je tělo cyklu provedeno $n-1$, n, a $n+1$ krát.

Pokrytí Data Flow grafu

Motivace

- Použití nedefinovaných proměnných.
- Mohou být cesty v programu, na kterých je nějaká proměnná nastavena za určitým úmyslem, ale posléze je hodnota této proměnné zneužita k jinému účelu.
- Control Flow kritéria nezaručují zahrnutí testů pokrývající popsaný případ.

Data Flow pokrytí

- Pokrytí všech míst programu, ve kterých je daná proměnná použita, ne však nutně definována podél všech cest v Control-Flow grafu.

Podpora pro Code Coverage

C/C++, Linux

- Nástroje `gcov` and `lcov`.

Příklad: lcov

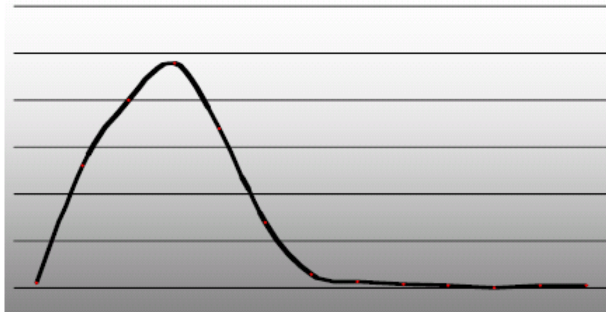
- ```
  gcc -fprofile-arcs -ftest-coverage foo.c -o foo
  lcov -d . -z
  lcov -c -i -d . -o base.info
  ./foo
  lcov -c -d . -o collect.info
  lcov -d . -a base.info -a collect.info -o result.info
  genhtml result.info
  ```
  ```

Křivky odhalených/opravených chyb jako metrika míry

Týdenní statistiky

- Počet nově odhalených chyb
- Počet opravených chyb
- Podíl nalezených chyb vůči opraveným chybám

Ukázka křivky



Weibullovo rozložení

Pozorování

- Množství nalezených chyb vykazuje Weibullovo pravděpodobnostní rozložení
- Metoda míry provedeného testování, resp. množství zbývajícího objemu testování.
- Metoda určování data uvolnění produktu na trh.

Postup

- V okamžiku, kdy dojde rozložení za vrchol, lze za předpokladu znalostí parametrů Weibullova rozložení odhadnout, kdy pravděpodobnost odhalení další chyby v produktu klesne pod danou mez.
- Parametry rozložení ovlivňují "šířku" a "výšku/strmost" kopce.
- $F(x) = 1 - e^{-ax^{-b}}$ pro $x > 0$

Weibullovo rozložení – nedostatky

Fakta způsobující nepřesnost výše uvedené metody

- Testování nesleduje očekávaný způsob používání produktu.
- Pravděpodobnost nalezení různých chyb není stejná.
- Oprava chyb může způsobit nové chyby.
- Chyby nejsou nezávislé.
- Počet chyb v produktu se mění (není dána počáteční fixní hodnota).
- Samotné zanášení chyb do systému sleduje Weibullovo rozložení.
- Epochy v testování (různé testovací postupy) jsou nezávislé.
- Parametry rozložení nejsou dány.

Závěr

- Závěry vycházející z uvedeného rozložení jsou platné pouze pro velké projekty a i tak jsou často zavádějící.

Dopady používání Weibullovských křivek

První fáze

- Snaha o strmější stoupání a rané vyvrcholení křivky.
- Jakmile se dosáhne vrcholu křivky, lze odhadnout tvar křivky a udělat první odhady data uvolnění produktu.

Dopady

- Spouštění testů nad částmi produktu, o kterých se ví, že jsou vadné, nebo nedokončené.
- Preferuje se hledání a reportování snadných chyb, namísto hledání těch skutečně závažných.
- Důraz kladen na hledání chyb, ne na vývoj testovacích nástrojů (intenzifikace X extenzifikace)
- Vykazování jedné chyby jako několik chyb menších.
- Opakované vykazování chyb (například různými testery)
- ...

Dopady používání Weibullovských křivek

Druhá fáze

- Počet nalezených chyb za čas by měl klesat.
- Z tvaru křivky lze odvodit kolik chyb za čas by se mělo vykázat.
- Snaha vykázat stabilitu křivky (blízkou nule).

Dopady

- Opakovaní úspěšných testů.
- Důraz přesunut od hledání nových chyb k důkladnému popsání nalezených.
- Slučování různých chyb do jedné.
- Odkládání nalezení chyb (např. až na "po milestone").
- Zatajování/odmítnutí/ztracení chyb!
- Neformální reportování chyb, mimo systém.
- Firemní akce pro testery.
- Programátoři neopravují chyby, dokud je testeři nenahlásí.
- ...

Neúplnost testování

Definice

Pozorování

- Prostor, který má být prohledán, je obrovský.
- Prostředky a zdroje jsou omezené.

Co není úplné testování

- Úplné pokrytí
  - každý řádek kódu
  - každé větvení
  - každou sekvenci kódu
- Testeři nenacházejí nové chyby
- Testovací plán je dokončen

Co je úplné testování

- Na konci procesu testování nejsou skryté (neznámé) nedostatky produktu.
- Pokud se objeví nový nedostatek produktu, testování nemohlo být úplné.

Ústupky spojené s časovou tísní

V časové tísni se většinou pouze

- Analyzují výsledky testů.
- Řeší problémy.
- Popisují chyby.

V časové tísni nezbývá čas na

- Návrh testů.
- Provádění testů.
- Vývoj testovacího SW.
- Revize, inspekce proběhlých testů.
- Dokumentace testů.
- Automatizace testů.
- ...

Pozorování

- Čas potřebný pro úkony spjaté s testováním je výrazně větší, než čas který je k dispozici.

Důvody neúplnosti procesu testování

Možných testů je velmi mnoho (až nekonečně mnoho).

Provést všechny možné testy znamená:

- Otestovat všechny možné hodnoty na vstupu každé vstupní proměnné.
- Otestovat všechny možné kombinace vstupů všech vstupních proměnných.
- Otestovat každý možný běh systému.
- Otestovat každou možnou konfiguraci HW a SW, včetně konfigurací hypotetických cílových serverů, které jsou mimo vaši kontrolu.
- Otestovat každý způsob, jakým může uživatel produkt použít.

Nemožnost testovat všechny možné vstupy/kombinace

Šířka datové sběrnice

- Počet testů roste exponenciálně vzhledem k počtu bitů použitých pro reprezentaci dat.
- n-bitů vynucuje 2^n testů.

Další příklady

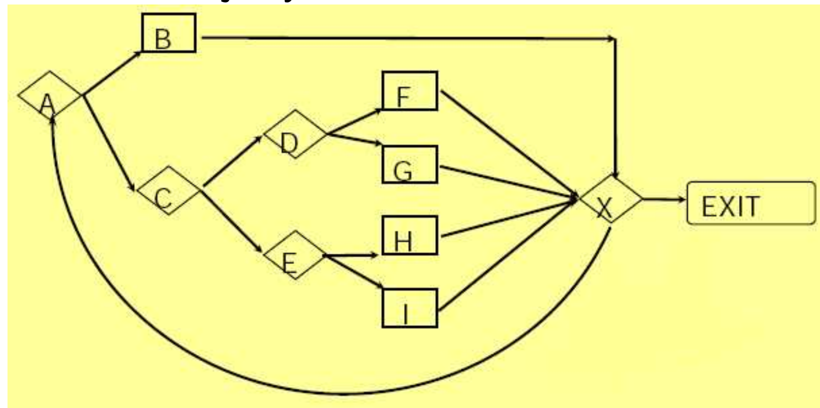
- Časování akcí
- Neplatné neočekávané vstupy (buffer overflow).
- Editované vstupy
- Velikonoční vejce [http://j-walk.com/ss/excel/eastereg.htm]

Častá praxe

- "Tohle by žádný uživatel našeho produktu neudělal."

Nemožnost testování všech běhů

Uvažme následující systém



Příklad

- Kolika způsoby lze dosáhnout EXIT ?
- Kolika způsoby lze dosáhnout EXIT , jestliže $\langle A \rangle$ lze navštívit nejvíce 20x?

Nemožnost testování všech běhů

Příklad

- V [F] je memory leak, v [B] garbage collector.
- Systém dospěje do neplatného stavu, pouze pokud se cestě s [B] bude dostatečně dlouho vyhýbat.

Fakta

- Zjednodušené testování "cest" v systému nemusí postihnout kritickou chybu.
- Kritická chyba se projeví za takových okolností, které by se nikdy jednoduchým testem neprověřovaly.
- Problém dlouhých běhů systému.

Shrnutí neúplnost a míra

Neúplnost

- Testováním nelze prokázat, že systém neobsahuje chyby.
- Otestovat všechny možné případy je nemožné.
- Existence testovacího plánu brání kreativitě testerů.

Měřitelnost

- Existují metody pro měření progrese ve fázi testování.
- Metody jsou nespolehlivé.
- Posuzování výkonnosti testovací skupiny na základě dané metriky může ovlivnit testování samotné.

IV113 Validace a verifikace

Testování černé krabice

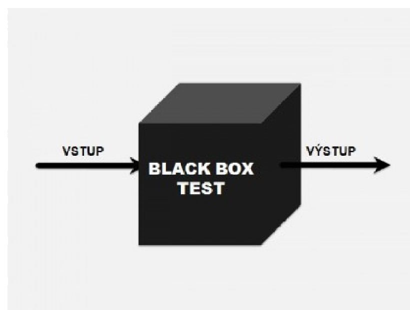
(Black-Box Testing)

www.testingeducation.org/BBST

Jiří Barnat

Black-box testování

- Na testovaný produkt je nahlíženo jako na černou skříňku.
- Vnitřní chování produktu je pro testera nepozorovatelné.
- Produkt je analyzován skrze jeho vstup-výstupní chování.
- Testování nezohledňuje zdrojový kód produktu.



White-box, Gray-box testování

White-box testování (Glass-box)

- Vnitřní chování produktu je možné pozorovat a využít.
- Tvorba a provádění testů vzhledem k vnitřnímu chování produktu, např. pro dosažení určitého stupně pokrytí.
- Vložení vnitřních chyb, kódu do produktu za účelem provedení vybraných testů.
- **Často rozšiřuje testování Black-box technikou.**

Gray-box testování

- Mezi Black-box a White-box testováním.
- Některými autory neodlišováno od White-box testování.

Techniky testování

- Doménové testování
- Kombinatorické testování
- Regresní testování
- Scénářové testování
- Funkční testování
- Fuzz testování
- Risk-based testování
- ...

Je výčet úplný?

- Publikováno víc jak 100 technik $\Rightarrow$ výčet není úplný.

Příklad – adder

Je dána následující specifikace

- Program sčítá dvě celá čísla, která se mu zadají na vstup.
- Každé číslo by mělo mít jednu nebo dvě cifry.
- Program vytiskne součet.
- Program očekává "Enter" za každým číslem.
- Program se spouští příkazem `adder`.

Jaké jsou nedostatky specifikace?

Příklad – adder

Je dána následující specifikace

- Program sčítá dvě celá čísla, která se mu zadají na vstup.
- Každé číslo by mělo mít jednu nebo dvě cifry.
- Program vytiskne součet.
- Program očekává "Enter" za každým číslem.
- Program se spouští příkazem `adder`.

Jaké jsou nedostatky specifikace?

- Pracuje se zápornými čísly? (Ano)
- Jak se program ukončí? (Ctrl+C)
- Vytiskne kam? (Na display/obrazovku)

Příklad – adder

- Jaký bude základní test?

Příklad – adder

- Jaký bude základní test?
  (něco ala 3+7)
- Kolik je možných specifikovaných vstupů?

Příklad – adder

- Jaký bude základní test?
  (něco ala 3+7)
- Kolik je možných specifikovaných vstupů?
  (celkem 199x199=39,601)
- Budete je testovat všechny?

Příklad – adder

- Jaký bude základní test?
  (něco ala 3+7)
- Kolik je možných specifikovaných vstupů?
  (celkem 199x199=39,601)
- Budete je testovat všechny?
  (je to možné, ale ne)
- Budete testovat 3+8, 4+7, 2+7, 3+6, 3+3?

Příklad – adder

- Jaký bude základní test?
  (něco ala 3+7)
- Kolik je možných specifikovaných vstupů?
  (celkem 199x199=39,601)
- Budete je testovat všechny?
  (je to možné, ale ne)
- Budete testovat 3+8, 4+7, 2+7, 3+6, 3+3?
  (zřejmě ne)
- Budete testovat 100 a více, a -100 a méně?

Příklad – adder

- Jaký bude základní test?
  (něco ala 3+7)
- Kolik je možných specifikovaných vstupů?
  (celkem 199x199=39,601)
- Budete je testovat všechny?
  (je to možné, ale ne)
- Budete testovat 3+8, 4+7, 2+7, 3+6, 3+3?
  (zřejmě ne)
- Budete testovat 100 a více, a -100 a méně?
  (ano)

Příklad – adder

- Jaký bude základní test?
  (něco ala 3+7)
- Kolik je možných specifikovaných vstupů?
  (celkem 199x199=39,601)
- Budete je testovat všechny?
  (je to možné, ale ne)
- Budete testovat 3+8, 4+7, 2+7, 3+6, 3+3?
  (zřejmě ne)
- Budete testovat 100 a více, a -100 a méně?
  (ano)

= **Doménové testování**

Sekce

Doménové testování

Doménové testování

Princip techniky

- Všech možných hodnot vstupů je příliš mnoho.
- Testování podobných hodnot vstupů je nezajímavé, testy jsou **ekvivalentní** např. vzhledem k průchodu grafem toku řízení.
- Je vhodné omezit provádění podobných testů, naopak je vhodné realizovat testy s hraničními a jinak zajímavými hodnotami vstupů.

"Matematický pohled"

- Relace ekvivalence na množině testů umožňuje rozdělit prostor všech možných testů na třídy ekvivalence.
- Produkt lze otestovat s použitím reprezentantů tříd rozkladu.

Analýza vstupních i výstupních proměnných

Tabulkou hraničních případů (THP)

Proměnná	Třída ekvivalence platných vstupů	Třída ekvivalence neplatných vstupů	Hraniční a zvláštní případy	Poznámky
První číslo	[-99,99]	< -99 > 99	-100,-99 99, 100	
Druhé číslo	[-99,99]	< -99 > 99	-100,-99 99, 100	
Součet	[-198,198]	< -198 > 198	(-99,-99) (99, 99) (-99,99) (99,-99)	Nelze vytvořit test s neplatnou hodnotou

Tabulka hraničních případů a testovací plán

Tabulka hraničních případů (THP)

- Nepřímá definice tříd ekvivalence.
- Nedílná součást testovací dokumentace.
- Klíčový nástroj pro tvorbu testovacího plánu.

THP jako součást testovacího plánu

- Stanovuje, co může být předmětem testu.
- Zahrnuje očekávané výsledky testů.
- Slouží jako kompaktní seznam testů.
- Umožňuje identifikaci testů.
- Lze použít i jako míru testování.

Budování THP v praxi

Zkušenosti z praxe

- Získat kompletní THP je náročné.
- V procesu testování je THP budována/doplňována během procesu testování.
- Kompletní THP se v praxi nedosahuje.
- THP obsahuje všechny proměnné, ale pouze u kritických proměnných je vyplněna.

Důvody

- Specifikace se tvoří za běhu. Samotný proces testování odkrývá nové vstupní proměnné.
- Proměnných je příliš mnoho, analýza mnohých proměnných by proběhla pouze za účelem vyplnění THP.

Hraniční hodnoty spojitých vstupů

Příklad

- Vstupní hodnota je desetinné číslo x z intervalu $(y, z]$.

Možné řešení

- Δ – nejmenší možná změna ovlivňující výsledek testu.
- Hraniční hodnoty: y, $y + \Delta$, z, $z + \Delta$

Δ **ovlivněna následujícími faktory**:

- Jaká je přesnost vstupního pole.
- Jaká je přesnost jednotlivých procedur, které s hodnotou nějakým způsobem pracují.
- S jakou přesností se hodnota projeví ve výsledku.

Uspořadatelné množiny

Uspořadatelné množiny

- Testování reprezentantů tříd ekvivalence a identifikace hraničních hodnot nejsou svázány pouze s číselnými doménami.
- Obecně lze doménovým přístupem analyzovat vstupy jejichž potenciální hodnoty lze uspořádat.

Pozorování

- Pokud entity nelze uspořádat, lze využít obecných měřitelných a uspořadatelných vlastností.
  - Velikost a počet entit.
  - Časování událostí.
  - Pozorovaná granularita (např. rozlišení)
  - ...

Příklad – Trojúhelník

Specifikace (Gerald Weinberg, 1969; Glen Myers 1979)

- Program přečte tři čísla a interpretuje je jako délky stran trojúhelníku.
- Na výstup vypíše, zda se jedná o trojúhelník rovnostranný, rovnoramenný, či obyčejný (ani rovnostranný ani rovnoramenný).

Jaké jsou třídy rozkladu a hraniční hodnoty?

Příklad – Trojúhelník

Specifikace (Gerald Weinberg, 1969; Glen Myers 1979)

- Program přečte tři čísla a interpretuje je jako délky stran trojúhelníku.
- Na výstup vypíše, zda se jedná o trojúhelník rovnostranný, rovnoramenný, či obyčejný (ani rovnostranný ani rovnoramenný).

Jaké jsou třídy rozkladu a hraniční hodnoty?

- Obyčejný, rovnoramenný, rovnostranný trojúhelník.
- Délky, které netvoří trojúhelník.
- Délky, které tvoří trojúhelník s nulovou výškou.
- Případy s jednou a více nulovými délkami.
- Záporná délka strany.
- Zadány 2 nebo 4 hodnoty.
- Zadány nečíselné hodnoty.

Příklad – Kumulace a kombinace vstupů

Hodnoty vstupů více proměnných, jejichž kombinace je neplatná, avšak každá zvlášť je kombinovatelná s jinými hodnotami tak, aby tvořila platnou kombinaci.

Příklad – Trojúhelník

- [3,3,6], [3,6,3], [6,3,3]

Příklad – NHL computer game

- Podle bodů ze základní skupiny (vítěství 2, remíza 1, prohra 0) se určí 8 týmů, které postupují do play-off.
- 80 zápasů v základní části.
- Součet bodů uchováván v signed byte [-128,127].
- Jaká je pozorovaná veličina a jaká její hraniční hodnota?

Příklad – Další nečíselné vstupní domény

Příklad – Adder

- Prodleva před zadáním druhého čísla.
- Zadání příliš velkého čísla.
- Vedoucí nuly či mezery.
  - Nechť $009 = 09 = 9$. Platí, že $000000000000000009 = 9$?
- Nenumerický vstup.
- Špatné chování v obsluze špatného vstupu.
  (Je specifikováno?)

Příklad adder – rozšíření THP

Proměnná	Třída ekvivalence platných vstupů	Třída ekvivalence neplatných vstupů	Hraniční a zvláštní případy	Poznámky
První číslo	[-99,99]	< -99 > 99 desetinné nečísla výrazy	-100,-99 99, 100 2.5 '/' ':' 0, null	
Druhé číslo	– || –	– || –	– || –	

Nejasná specifikace hraničních případů – příklad

Pozorování

- Specifikace je nedokonalá a dává možnost býti interpretována různě (ač správně).

Příklad specifikace

- Uvažme algoritmus, který plánuje pro hypotetickou firmu rozvoz zakázek na následující den.
- Politika firmy je preferovat zákazníky, kteří mají platné speciální povolení od managementu býti přednostně obslouženi, nebo nebyli obslouženi už 30 dnů.

Nepřesnosti vzhledem k hraničním hodnotám

- Nebyli obslouženi víc jak 30 dnů, nebo 30 dnů včetně?
- 30 dnů měřeno v době plánování, či v době dodávky?
- Dny měřeny od půlnoci, či od hodiny dodávky?
- Kdy se posuzuje platnost povolení?

Reverzní hledání domén a hraničních hodnot

Princip

- Vycházíme od možného problému a zpětným postupem hledáme vstupní data, která vedou k vyvolání problému.
- Na základě nalezených hodnot redefinujeme třídy ekvivalence.

Postup

- Identifikujeme problém.
- Kategorizujeme testy na *způsobí/nezpůsobí* chybu.
- Pokud neexistuje test, který problém vyvolá, je třeba podniknout modifikaci definice domén, aby odpovídající test nemohl být vynechán.

Domény v neuspořadatelných množinách

Pozorování

- Existují případy, kdy jednotlivé možné hodnoty vstupní proměnné nelze uspořádat.
- Nelze identifikovat hraniční případy.
- Jak testovat vstupní proměnnou pokud může nabývat velkého množství hodnot?

Doménové testování neuspořadatelných množinách

- Rozlišení objektů na základě jejich atributů.
- Identifikace ekvivalence a odpovídajících tříd.
- Volba vhodných reprezentantů jednotlivých tříd.

Neuspořadatelné množiny – Tiskárny

Testování výstupu na tiskárnu

- Existuje víc jak tisíc různých tiskáren.
- Požadujeme kompatibilitu i se staršími typy.
- Třídy / Podtřídy
  - Třídy: LJ II, LJ III, Postscript Level I, Postscript Level II, Epson
  - Podtřídy LJ II: LJ II, LJ II+, LJ IIP, LJ IID

Pozorování

- Jednotlivé nedostatky se projevují zřetelněji na různých typech tiskáren. (Analogie s uspořádanými množinami.)

Doporučení

- Je vhodné volit reprezentanty tak, aby pokryly co možná nejvíce nedostatků.

Doménové testování – Teorie

Historie

- Domény (obory hodnot) a sub-domény byly v testování používány mnohem dřív, než bylo v rámci softwarového testování teoreticky doménové testování popsáno (1988, Testing Computer Software).

Teoretický pohled

- Doména je množina prvků, na kterých je pomocí nějaké relace ekvivalence definován rozklad.
- Testování domény probíhá pomocí vybraných reprezentantů jednotlivých tříd.
- Kreativita při definici relací ekvivalence je velmi vítaná.

Doménové testování – Volba reprezentantů

Pozorování

- Hraniční případy jsou dobrou volbou.

Nevýhody hraničních případů

- Volba hraniční hodnoty jako reprezentanta je heuristika.
- Nejlepší reprezentant v dané třídě je ten, který má největší pravděpodobnost vyvolat chybu.
- Neexistují v neuspořadatelných doménách.

Praxe

- Volí se hraniční a zajímavé nehraniční případy.

Doménové testování – Závislé vstupy

Pozorování

- Závislost vstupních proměnných může způsobit, že daná hodnota jedné proměnné je hraničním případem pouze při konkrétní hodnotě jiné vstupní proměnné.

Doporučení

- V takovém případě je vhodné zvolit všechny hodnoty, které mohou býti v nějaké konfiguraci hraničním případem.

Příklad závislých proměnných — měsíc a den v měsíci

- Měsíc: 2 – 28 až 29 dnů
- Měsíc: 4,6,9,11 – 30 dnů
- Měsíc: 1,3,5,7,8,10,12 – 31 dnů

Výhody a nevýhody doménového testování

Výhody

- Základní nástroj pro boj s neúplností testování zachovávající rozumné a rovnoměrné pokrytí stavového prostoru možných testů.
- Analýza hraničních hodnot domén vstupních proměnných dává přehled o rozsahu a množství nezbytných testů.
- Hraniční případy vstupních proměnných jsou případy, při kterých se často projeví chyby produktu.

Výhody a nevýhody doménového testování

Nevýhody

- Hraniční hodnoty mohou být nejasné nebo skryté, navíc dochází ke zjemňování rozkladu v průběhu procesu.
- Neredukuje množství kombinací způsobené binárními vstupy a počtem vstupních proměnných.
- Falešný pocit bezpečí.

Použití

- Samostatně jako technika se používá zřídka.
- Vhodně doplňuje ostatní metody testování.

Kombinatorické testování

Kombinatorické testování

Pozorování

- Při testování mnoha vstupních proměnných, počet možných testů roste exponenciálně.

Cíl kombinatorického testování

- Pokrýt exponenciální prostor možných vstupů rozumným způsobem s výrazně menším počtem testů.

Princip techniky

- Identifikace podmnožiny možných testů, které splňují jisté kombinatorické vlastnosti.

Přístupy ke kombinatorickému testování

Mechanické (procedurální)

- Testované kombinace vstupů jsou určeny/vypočteny mechanickým postupem.

Založené na možném riziku

- Testované kombinace vstupů jsou určeny na základě možného rizika.

Založené na scénářovém testování

- Testované kombinace vstupů vyplývají ze zajímavých scénářů použití produktu.

Kombinace kombinatorického a doménového testování

Pozorování

- Každá jedna vstupní proměnná může sama o sobě mít mnoho možných vstupních hodnot.
- V případě testování kombinací vstupních proměnných, nelze testovat celou doménu každé proměnné.
- Domény hodnot jednotlivých proměnných se typicky předzpracují s využitím technik doménového testování.

Příklad volby reprezentantů v rámci jedné domény

- PM – příliš malá hodnota, NM – nejmenší platná hodnota
- NV – největší platná hodnota, PV – příliš velká hodnota

Pro N proměnných, je prostor možných testů

- $\{PM, NM, NV, PV\} \times \ldots \times \{PM, NM, NV, PV\}$

Mechanické kombinatorické testování

Slabé testování, verze 1

- Cílem je vybrat takovou množinu testů, aby každá proměnná byla alespoň jednou testována na každou hodnotu ze své domény.

Možné testy

	V1	V2	V3
Test1	NM	NM	NM
Test2	NV	NV	NV
Test3	PM	PM	PM
Test4	PV	PV	PV

Pozorování

- Jaký je smysl testů 3 a 4?
- Je možné, že testy 3 a 4 vůbec nelze provést.

Mechanické kombinatorické testování

Slabé testování, verze 2

- Cílem je vybrat takovou množinu testů, aby každá proměnná byla alespoň jednou testována na každou hodnotu ze své domény, tak aby žádný test neobsahoval neplatné vstupy u víc jak jedné proměnné.

	V1	V2	V3
Test1	NM	NM	NM
Test2	NV	NV	NV
Test3	PM	NM	NV
Test4	NV	PM	NV
Test5	NM	NV	PM
Test6	PV	NV	NM
Test7	NM	PV	NM
Test8	NV	NM	PV

Slabé testování, verze 3

- Cílem je vybrat takovou množinu testů, aby každá proměnná byla alespoň jednou testována na každou platnou hodnotu ze své domény.
- Obecně označována jako technika "All singles"

Možné testy

	V1	V2	V3
Test1	NM	NM	NM
Test2	NV	NV	NV

Mechanické kombinatorické testování

Silné testování, verze 1

- Cílem je otestovat všechny možné kombinace vstupů.

Možné testy

- 4x4x4=64 možností

Silné testování, verze 2

- Cílem je otestovat všechny možné kombinace platných vstupů.

Možné testy

- 2x2x2=8 možností

Silné testování, verze 3

- Cílem je otestovat všechny N-tice, tj. zvolit takovou sadu testů, aby každá podmnožina vstupních proměnných o N prvcích byla silně testována na všechny (platné) vstupy.
- Obecné označení: "all N-tuples", "all-pairs", "all triples".

Možné testy pro N=2 musí pokrýt následující kombinace

V1	V2
NM	NM
NV	NM
NM	NV
NV	NV

V1	V3
NM	NM
NV	NM
NM	NV
NV	NV

V2	V3
NM	NM
NV	NM
NM	NV
NV	NV

	V1	V2	V3
Test1	NM	NM	NM
Test2	NV	NM	NV
Test3	NM	NV	NV
Test4	NV	NV	NM

Příklad

Tvorba tabulky pokrývající všechny dvojice

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C   V2:D,E   V3:F,G   V4:H,I   V5:J,K   V6:L,M

V1	V2	V3	V4	V5	V6
A	D				
A	E				
B	D				
B	E				
C	D				
C	E				

- 6 testů

Příklad

Tvorba tabulky pokrývající všechny dvojice

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C V2:D,E V3:F,G V4:H,I V5:J,K V6:L,M

V1	V2	V3	V4	V5	V6
A	D	F			
A	E	G			
B	D	G			
B	E	F			
C	D	G			
C	E	F			

- 6 testů

Příklad

Tvorba tabulky pokrývající všechny dvojice

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C V2:D,E V3:F,G V4:H,I V5:J,K V6:L,M

V1	V2	V3	V4	V5	V6
A	D	F	H		
A	E	G	I		
B	D	G	I		
B	E	F	H		
C	D	G	H		
C	E	F	I		

- 6 testů

Příklad

Tvorba tabulky pokrývající všechny dvojice

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C   V2:D,E   V3:F,G   V4:H,I   V5:J,K   V6:L,M

V1	V2	V3	V4	V5	V6
A	D	F	H	J	
A	E	G	I	K	
B	D	G	I	K	
B	E	F	H	J	
C	D	G	H	J	
C	E	F	I	K	

- 6 testů

Příklad

Tvorba tabulky pokrývající všechny dvojice

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C V2:D,E V3:F,G V4:H,I V5:J,K V6:L,M

V1	V2	V3	V4	V5	V6
A	D	F	H	J	
A	E	G	I	K	
B	D	G	I	J	
B	E	F	H	K	
C	D	G	H	K	
C	E	F	I	J	

- 6 testů

Příklad

Tvorba tabulky pokrývající všechny dvojice

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C V2:D,E V3:F,G V4:H,I V5:J,K V6:L,M

V1	V2	V3	V4	V5	V6
A	D	F	H	J	L
A	E	G	I	K	M
B	D	G	I	J	L
B	E	F	H	K	M
C	D	G	H	K	M
C	E	F	I	J	L

V1	V2	V3	V4	V5	V6
A	D	F	H	J	L
A	E	G	I	K	M
B	D	G	I	J	M
B	E	F	H	K	L
C	D	G	H	K	L
C	E	F	I	J	M

Příklad

Tvorba tabulky pokrývající všechny dvojice

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C V2:D,E V3:F,G V4:H,I V5:J,K V6:L,M

V1	V2	V3	V4	V5	V6
A	D	F	H	J	L
A	E	G	I	K	M
			H		M
B	D	G	I	J	M
B	E	F	H	K	L
			I		L
C	D	G	H	K	L
C	E	F	I	J	M

- 8 testů místo původních 3x2x2x2x2x2=96 testů

Tabulka testů pokrývající jednotlivé případy (all singles)

- Pozorované proměnné V1,V2,V3,V4,V5 a V6
- V1:A,B,C V2:D,E V3:F,G V4:H,I V5:J,K V6:L,M

V1	V2	V3	V4	V5	V6
A	D	F	H	J	L
B	E	G	I	K	M
C					

All pairs – komplikace

Závislé proměnné

- Některé kombinace hodnot jsou neplatné, přestože hodnoty vyskytující se v neplatné kombinaci jsou platné v jiných kombinacích.
- Viz příklad s počty dnů v jednotlivých měsících.

Praxe

- Vypočítat all-pairs může být v praxi natolik komplikované, že se od all-pairs kombinatorického testování sklouzne k all-singles technice a náhodným kombinacím.
- Praktické řešení je používat all-singles plus kombinace, které dříve způsobily chyby, nebo jsou něčím zajímavé.

All-pairs shrnutí

All-pairs kombinatorické testování

- Významně redukuje prostor možných testů.
- Nástavba doménového testování.
- Vhodná pro verifikaci kombinací **nezávislých** proměnných.

Použití all-pairs na závislých proměnných

- Neplatnou hodnotu v dané kombinací je možné zaměnit jinou platnou hodnotou.

Náhodné kombinace

- Po zpracování jednotlivých domén vstupních proměnných bývá testování náhodných kombinací možných vstupů stejně efektivní jako testování metodou all-pairs.
- Získání náhodných kombinací je výrazně snazší, než výpočet all-pairs tabulek.

Scénářové testování

Scénářové testování

Princip techniky

- Testovat produkt pomocí fiktivních příběhů použití.

Základní cíl

- Otestovat produkt způsobem, který připomíná/napodobuje skutečné použití produktu.

Vlastnosti scénářů

Motivační

- Udává jak, ale také proč, je produkt takto používán.
- Odpovědná osoba by měla trvat na nápravě chyby, pokud se uvedeným scénářem nějaká chyba projeví.

Reálný

- Scénář by měl být věrohodný, tj. nevyprávět příběh, který by se mohl stát, ale příběh, který se pravděpodobně stane.

Netriviální

- Scénář by měl zahrnovat komplexní použití programu v komplexním prostředí na netriviálních datech.

Čitelný

- **Výsledky scénářového testu by měly být čitelné a zřejmé, tj. neměly by být skryty v komplexním výstupu programu.**

Scénářové testování a nejasnosti v požadavcích na produkt

Pozorování

- Tvorba scénářových testů je podobná procesu analýzy požadavků a tvorby specifikace.
- Scénářové testování vynáší na povrch "pod koberec zametená" nevyřešená rozhodnutí ohledně požadavků na produkt a specifikace.

Nedostatky požadavků a specifikace

- Snaha testera je poukázat v jakém kontextu se problém projeví a jaké to může mít následky.
- Je důležité udržet si roli testera.
- Tester nehledá řešení problému.
- Tester nedělá kompromisy v návrhu produktu (nepromíjí), naopak snaží se prokázat dopady možných kompromisů.

Scénář pro test programu pro výpočet daně z příjmů

Na začátku loňského roku byl Pavel rozvedený a z předchozího manželství měl 2 dospělé děti, z nichž jedno v březnu zemřelo ve věku 19 let při autonehodě, druhé dosáhlo věku 26 let v srpnu. Pavel celý rok pracoval na částečný (2/3) úvazek jako dělník v zavedené firmě s hrubým ročním příjmem 360.000Kč. Ze začátku roku se seznámil s pohlednou Helenou, od února žili ve společné domácnosti, v březnu se vzali a v říjnu se jim narodila dcera. Aby mohl Pavel novou rodinu uživit, začal v červnu podnikat. Na rozjezd podnikání si vzal půjčku ze Stavebního spoření, kterou však splatil v prosinci, když konečně skončilo vleklé dědické řízení, ve kterém Pavel zdědil 421.456 Kč. Na úrocích z půjčky Pavel zaplatil 37.210 Kč. Na konci roku byla finanční bilance Pavla jako podnikatele záporná, konkrétně -13.000 Kč. Vzhledem k rizikovému těhotenství Helena pracovala pouze první dva měsíce v roce a její příjmy nepřesáhly 25.000 Kč. Pavel s Helenou uplatňují společné zdanění manželů.

Nevýhody scénářového testování

Nevhodné pro testování v rané fázi vývoje

- Scénář by měl být komplexní, dosud neimplementované funkce brání provedení testu.
- Jednotlivé funkce má smysl testovat nejprve samostatně.

Nevhodné pro metriku pokrytí kódu programu

- Pokrytí kódu scénářem je náročné a vede scénář do netypických situací.

Znovupoužití scénářů může být neefektivní

- Příprava dobrého zcela nového scénáře je náročná.
- Tvorba nových scénářů modifikací existujících nemusí odhalit zcela jiné typy chyb.
- Na odhalené chyby lze připravit jiné typy testů a ty například použít v regresním testování.

Testování odvozené od možných rizik (Risk-based testing)

Motivace

Pozorování

- Jednotlivé dosud probrané metody testování se příliš zaměřují na jednotlivé aspekty/způsoby použití produktu.
- Důkladná realizace jedné z metod se postupem času stává nákladná a neúčinná v detekci jiných nedostatků produktu.

Pragmatický pohled

- Jednotlivé testy předepsané důkladnou aplikací daných testovacích metod je třeba podrobit analýze a zvážit, zda jejich provedení pomůže naplnit misi.
- Volbu správné strategie, metody testování a realizovaných testů je možné provést na **základě možných rizik**.

Riziko

Riziko

- Možnost utrpět ztrátu či být postižen nějakou škodou.

Dimenze rizika v softwarovém inženýrství

- Jakým způsobem může program selhat.
- Jak pravděpodobné je, že program selže.
- Jaké jsou důsledky selhání programu.

Princip techniky testování rizikem

- Uvědomění si, jakým způsobem může program selhat.
- Návrh testů, které odhalí myšlené (potencionální) selhání.

Testování odvozené od možných rizik

- Přirozeně (nevědomky) používaná metoda, často již samotnými vývojáři systému.
- Typická metoda pro hledání chyb.
- Snadno se prolíná s ostatními metodami.

Příklad – využití rizika v doménovém testování

- V číselných doménách je často používána 0 jako jeden z hraničních případů. Důvodem je mimo jiné i to, že existuje riziko dělení nulou.
- I na volbu hraničních případů lze nahlížet jako na volbu zdůvodněnou rizikem záměny $<$ a $\leq$.

Rizikové testování – cyklus

Testovací cyklus rizikového testování

- Analýza selhání
- Zdokonalení procesu analýzy
- Určení priorit jednotlivých rizik
- Provedení odpovídajících testů
- Ohlášení chyb
- Náprava některých problémů

Klíčová otázka rizikového testování

Otázka

- Jakým způsobem může program selhat?

Hledání odpovědi

- Mnoha způsoby, úplného výčtu není možné efektivně dosáhnout.
- Pro identifikaci rizik se používají zejména zkušenosti z předchozích projektů, a různé heuristiky.

Detekce rizik

Správa projektu

- Nové nebo modifikované části produktu.
- Nová technologie v produktu.
- Proces učení.
- Části vyvíjené v časové/finanční tísni.

Lidský faktor

- Distribuovaný tým.
- Osobní vztahy.
- Nečekané “features”.
- Práce odvedená třetí stranou.
- Práce odvedená zdarma.

Detekce rizik

Specifikace a požadavky

- Nepřesná/konfliktní specifikace.
- Záhadná mlčenlivost.
- Požadavky vyvíjející se za běhu.

Výskyt chyb

- Chybové části.
- Opravené části.
- Komplexní části.

Proces testování

- Málo testované/netestované části.
- Nedostatečná různorodost testovacích technik.
- Neopravitelné chyby.

Detekce rizik

Prevence

- Testování částí, které v případě projevu chyby mohou mít důsledky.

Možné důsledky projevení chyby

- Špatná publicita,
- právním následky,
- výrazné škody,
- nesoulad s požadavky,
- zneužití produktu,
- zklamání většiny uživatelů,
- ohrožení strategického postavení,
- neuspokojení V.I.P. osob, ...

Dimenze rizika v softwarovém inženýrství

- Jakým způsobem může program selhat.
- **Jak pravděpodobné je, že program selže.**
- **Jaké jsou důsledky selhání programu.**

Odhad míry rizika

- Ohodnocení pravděpodobnosti výskytu a závažnosti důsledků číslem v rozmezí [1-10]
- Míra rizika = [Pravděpodobnost] x [Důsledky]

Management

- Preference odstranění chyb s větší mírou rizika.

Míra rizika může být zavádějící

Nedostatky

- Jaká je pravděpodobnost výskytu chyby v produktu?
- Jaký je rozdíl mezi známkou 3, 4 a 5?

Příklad – Borland's Turbo C++

- Chyba: poškození projektového souboru
- Pravděpodobnost: 1 (very rare)
- Závažnost: 10 (critical)
- Míra rizika uvedené chyby: 10
- Nejzávažnější chyba projektu, ale v měřítku míry rizika dosahuje pouze 26% maxima.
- Chyba [5]x[5] je o 50% závažnější.

Další techniky testování černé krabice

Nejpoužívanější techniky black-box testování

- Doménové testování
- Testování na základě rizika
- Scénářové testování

- Fuzz testování
- Testování vůči specifikaci (Specification-based)
- Testování zátěží (Stress)
- Testování uživatelem (User)

Fuzz testování

Princip

- Tvorba nových testů náhodnou modifikací (mutační testování) nebo vypočtenou modifikací (whitebox fuzz testování) vstupních dat existujících testů.
- Genetické algoritmy pro tvorbu nových testů.

Motivace

- V black-box variantě levná a překvapivě úspěšná metoda.
- Docílit průchod jiných částí kódu (zvýšit pokrytí).

Problémy

- Ve white-box variantě je systematický výpočet modifikace vstupních dat náročný.

Testování vůči specifikaci

Princip

- Testování jednotlivých tvrzení ve specifikaci.

Motivace

- Chceme soulad se specifikací.
- Prokážeme souladu se specifikací.
- Metoda správy testování a testovacího úsilí jako celku.

Problémy

- Nevyjádřená, implicitní a nejasná fakta ve specifikaci.
- Zaměřeno na pokrytí specifikace, spíše než na eliminaci možných rizik.

Testování zátěží

Princip

- Pozorovat chování produktu při enormním zatížení.
- Vynutit chybu v produktu způsobenou jeho zahlcením.

Motivace

- Získání důvěra v chování produktu v kritických momentech.
- Prevence odhalení nových chyb masovým používáním produktu uživateli.

Problémy

- Generování odpovídající zátěže.

Testování uživatelem

Princip

- Simulace uvolnění produktu.
- Uvolnění produktu omezené skupině uživatelů, kteří mají za úkol ověřit funkčnost produktu.
- Beta test.

Motivace

- Kvalitu posuzují sami cíloví zákazníci. Ti odhalí skutečné problémy produktu.

Problémy

- Vyžaduje téměř finální verzi produktu.
- Získávání beta testerů.

IV113 Validace a verifikace

Testování se znalostí kódu, automatizace a symbolická exekuce

Jiří Barnat

Funkční testování
(Unit testing)

Funkční testování

Princip techniky

- Oddělené testování malých částí kódu na úrovni vnitřních rozhraní (API).
- Testování jednotlivých funkcí bez využití znalosti jejich implementace (interní black-box testing).
- Vyžaduje modulární kód.

Základní cíl

- Prověřit, že isolované funkce systému fungují správně.
- Odladěná podřešení se lépe kombinují do funkčního celku.

Globální postup

- Identifikují se vnitřní funkce produktu.
- Pro každou identifikovanou funkci se vytvoří test.

Testování funkcí systému v isolaci

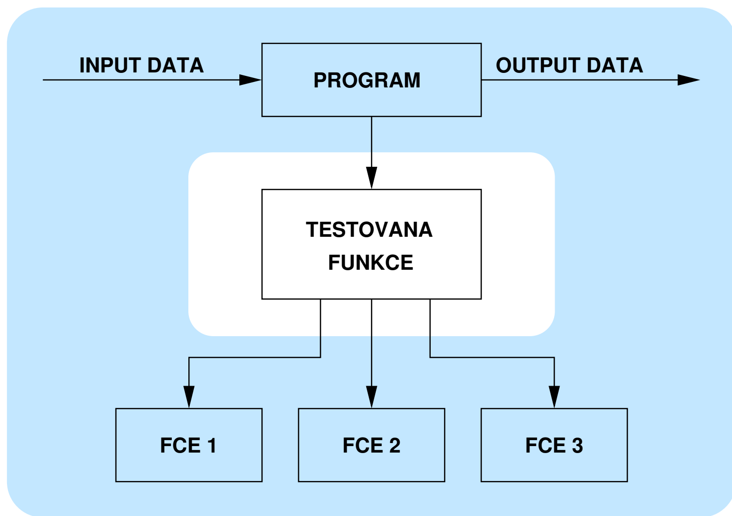
Testovaní isolované funkce

- Funkce isolovaná od systému není samostatně spustitelná.
- Je třeba vyvinout software, který umožní testování funkce.
- **Implementace testovacího prostředí probíhá jako součást implementace funkce.**

Testovací prostředí

- Vzhledem k testované funkci má vnější a vnitřní část.
- **Vnější část** slouží jako hlavní, samostatně spustitelný program, který volá testovanou funkci s danými parametry, sbírá a tiskne relevantní výsledky volání funkce.
- **Vnitřní část** simuluje chování dalších funkcí volaných testovanou funkcí.

Obecné schéma testovací prostředí funkce



Funkční testování – frameworky

Podpora pro funkční testování

- **Java**: jUnit, TestNG, qc4j
- **C++**: CUnit, TUT, QuickCheck++
- **Haskell**: QuickCheck, SmallCheck, HUnit
- **Python**: PyUnit, nose, qc
- **Ruby**: Test::Unit, RushCheck
- ...

Rozcestník

`http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`

Co je to funkce a kde je získat

Funkce

- Funkce je něco, co program může dělat.
- "Features", schopnosti, entity identifikované svými schopnostmi, ...

Identifikace funkcí

- Ze specifikace, či manuálu.
- Z uživatelského rozhraní.
- Z nápovědy v GUI/TUI.
- Prohledáním zdrojového kódu (názvy členských funkcí tříd, texty chybových hlášek, ...).

Seznam funkcí a popis funkcí

Seznam funkcí

- Základní dokument funkčního testování.

Informace obsažené v seznamu funkcí

- Kategorizace funkce, tj. označení skupiny funkcí s podobnou, či související funkcionalitou.
- Vstupy funkce
  - Maximum/Minimum, hraniční případy.
  - Speciální případy
- Výstupy funkce
- Rozsah působnosti funkce (není-li dána kategorizací).
- Možnosti/volby (options) funkce.
- Okolnosti, za kterých se funkce chová odlišně (globální konfigurace programu, verze a typ OS, ...)

Testování funkcí

Orákulum

- Je nutné vědět (nebo mít určeno) jak se pozná, že daný test dané funkce uspěl.
- Orákulum, může být součástí seznamu funkcí.

Neúplnost testování

- Není-li možné otestovat všechny vstupy, je vhodné použít princip doménového testování.

Konfigurace systému a vliv prostředí

- Testy funkce je vhodné opakovat v potenciálně různých podmínkách, při nichž je možné funkci využít.

Testování negativních případů

- Funkce by se měla testovat na to, že dělá to, co dělat má, ale i na to, že nedělá to, co dělat nemá.

Výhody funkčního testování

Pokrytí (coverage)

- Technika vhodná k realizaci úplného pokrytí testy.
- Vhodná technika k budování testovacího plánu, potažmo k měření míry splnění testovacího plánu.

Ranné testování

- Lze testovat už částečně vzniklý kód.
- Vede k rychlému odhalení mnoha chyb.
- Základem pro testy řízený vývoj produktu a další agilní metody vývoje.

Nevýhody funkčního testování

Rizika použití funkčního testování

- Pokud je v projektu přítomno, ochabuje potřeba realizovat jiné metody testování.
- Produkt, který je vystavěn z korektních funkcí, nemusí být korektní.

Nedostatky funkčního testování

- Neuvažuje interakci jednotlivých funkcí na stejné úrovni.
- S rostoucí integrací funkcí "ztrácí sílu".
- Nezachycuje chování funkcí v dlouhodobém běhu.
- Často se zaměřuje na testování schopnosti jako takové, ale ne na testování krajních případů.
- Neřeší otázky typu: Byla funkcí produktu naplněna uživatelova potřeba?

Regresní testování

Regresní testování

Princip techniky

- Opakování vybrané sady úspěšně proběhnuvších testů.

Hlavní důvod použití

- Riziko zanesení chyb změnou kódu.

Další uplatnění

- Potvrzení stability chování/výkonu produktu.
- Nástroj pro prokázání množství odvedené práce klientům.
- Psychologická podpora vývojářů.
  (Vývojáři mohou být odvážnější při změnách kódu.)

Prohřešky odhalené pomocí regresního testování

Oprava chyby je nedostatečná.

- Záplata je neúčinná.
- Záplata odstraní symptomy, ne však chybu samotnou.

Oprava chyby má vedlejší účinky.

- Výskyt nově zanesených chyb.
- Znovu vyvolání opravených chyb.

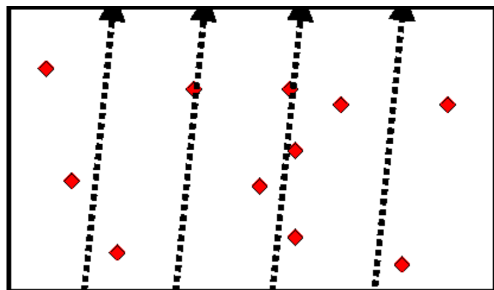
Produkt nelze sestavit.

- Typicky ve spojení se systémem pro kontrolu verzí.

Detekce nových chyb

- Z principu nedetekuje nové chyby, až na ...
- ... chyby závislé na předchozím použití produktu.
- ... chyby, jež se projeví díky přítomnosti nového kódu.

Příklad – Analogie s minovým polem



Aspekty ovlivňující výběr testů

Podstata regresního testování je v opakování testů.

Ptáme se:

- Které testy mají být součástí sady?
- Jaký je důvod pro opakování právě těchto testů?
- Jak přesně se mají jednotlivé testy v sadě vyhodnocovat?

Pozorování

- Podobně jako testování na základě rizika, i regresní testování jde napříč předchozími technikami testování.

Různé pohledy na volbu sady testů

Procedurální

- Opakujeme vybranou, nadále stejnou, sadu stejně vyhodnocovaných testů stejným způsobem.

Ekonomické

- Opakujeme všechny snadno opakovatelné a vyhodnotitelné testy.

Zaměřené na snížení rizika

- Opakujeme existující testy, jež v minulosti odhalily chyby, a volíme testy, jež pokrývají kritické části produktu.

Podpora při vývoji produktu

- Volíme testy, které pomohou rozhodnout, zda je korektní/smysluplné modifikovat produkt navrženým způsobem (sledujeme např. výkon aplikace).

Ekonomický faktor v regresním testování

Pozorování

- Produkt se vyvíjí a spolu s ním je nutné vyvíjet i testy, které se mají znovu spustit.
- Udržovat testy v aktuálním spustitelném stavu může být nákladné.
- **Nebrání údržba starých testů ve vývoji nových testů pro dosud neotestované části produktu?**

Regresní testování a funkční testování

Pozorování

- Regresní testování se typicky realizuje pomocí funkčního testování (unit testing).

Důvody

- Ve větších projektech je typické, že programátor samotný tvoří testy pro část kódu, který vyvíjí.
- Automatizovatelné od raného stádia vývoje.

Rizika

- Po dokončení vývoje modulů, metoda postrádá smysl.
- Sadu testů je nutné obohacovat o integrační testy.
  - Unit testy si tvoří sami vývojáři, jakmile však dochází k integraci, je pro vytvoření testu nutná znalost všech integrovaných částí, což může být na rámec působnosti každého jednotlivého vývojáře.

Automatizované a autonomní testování

Automatizovaná procedura testování

- "Jedním příkazem" se spustí sada testů, ty se provedou, vyhodnotí a na výstupu se zobrazí statistiky, případně identifikují neúspěšné testy.

Autonomní procedura testování

- Spouští se bez explicitního příkazu testera/uživatele.
- Spouštěcí mechanizmy
  - časová periodicita (každou půlnoc)
  - událostmi řízené spouštění
    (commit do systému pro správu verzí)

Buildbot

Připomenutí

- Znovu-spuštění testů – **stroj**
- Vyhodnocení výsledků – **stroj**+člověk

$\Rightarrow$ Regresní testování

BuildBot

- Systém pro podporu automatické kompilace a testování.
- Umožňuje spouštění testů na různých platformách.
- `buildbot.net`

Jiná řešení

- travis-ci, cdash, tinderbox, ...
- `http://nixos.org/hydra/`, ...

Další techniky white-box testování

Testování s použitím modelu

Princip

- Modelování produktu jako konečného automatu.
- Odvozování vlastností a nutné množiny testů na základě modelu.

Motivace

- Přiblížení se formální verifikaci.
- Matematická garance vlastností modelu potažmo produktu samotného.
- Generování minimální množiny testů.

Problémy

- Náročnost budování věrného modelu.

Symbolická exekuce

Motivace

Problém

- Detekovat chybu, která nastává pouze pro některé vstupy, je obtížné.
- Viz neúplnost testování.

Co bychom chtěli

- Testovat program na všechny možné vstupy.

Myšlenka

- Vykonávání programu, při němž jsou hodnoty vstupních proměnných označeny symboly a během výpočtu manipulovány symbolicky.

Příklad

Program	Vybrané konkrétní hodnoty	Symbolická reprezentace
`read(A)`		
	$A = 3$	$A = \alpha$
`A = A * 2`		
	$A = 6$	$A = \alpha * 2$
`A = A + 1`		
	$A = 7$	$A = (\alpha * 2) + 1$
`output(A)`		

Větvení v programu a podmínka cesty

Pozorování

- Větvení v kódu programu klade další omezení na možné hodnoty symbolických vstupů.

Příklad

```
1 if (A == 2)
2   then ...
3   else ...
```

$A = (\alpha * 2) + 1$

$(\alpha * 2) + 1 = 2$

$(\alpha * 2) + 1 \neq 2$

Podmínka cesty (Path condition)

- Formule nad symboly označující vstupní hodnoty.
- Kóduje historii výpočtu, tj. kumuluje omezení jež vyplynula z podmínek v místech větvení programu během výpočtu (z počátečního až do aktuálního bodu).
- Iniciálně prázdná (true).

Pozorování

- Podmínka cesty může být nesplnitelná.
- Tento jev indikuje nerealizovatelnost průchodu programem asociovaného s danou cestou.

Příklad 1

```
1 if (A == B)        A = α, B = β
2   then                                α = β
3     if (A == B)
4       then ...                        α = β ∧ α = β
5       else ...  (přeškrtnuto)         α = β ∧ α ≠ β    is UNSAT
6   else ...                            α ≠ β
```

Příklad 2 % – operace modulo

```
1 A=A%2                          A = α%2
2 if (A == 3) then ...  (then ... přeškrtnuto)    α%2 = 3    is UNSAT
3             else ...                            α%2 ≠ 3
```

Strom symbolické exekuce

Pozorování

- Možné průchody programem lze seskupit a reprezentovat stromovou strukturou – **strom symbolické exekuce**.
- Struktura stromu vzniká rozbalováním grafu toku řízení.

Strom symbolické exekuce

- Vrchol stromu je tvořen lokací programu, symbolickou valuací proměnných a podmínkou cesty, například:

lokace	valuace	podmínka cesty
#12	$A = \alpha + 2, B = \alpha + \beta - 2$	$\alpha = 2 * \beta - 1$

- Hrana mezi vrcholy odpovídá vykonání příkazu na dané lokaci s odpovídající aktualizací symbolické valuace.
- Větvení v programu způsobí větvení ve stromové struktuře a aktualizaci podmínek cest.

Příklad stromu symbolické exekuce

Program

```
1 input A,B
2 if (B<0) then
3   return 0
4 else
5   while (B > 0)
6   { B=B-1
7     A=A+B
8   }
9 return A
```

DODO = DOdělej DOma

Vlastnosti stromu symbolické exekuce

- Nedochází ke spojování vrcholů (dosažení identické trojice nevede k žádné zpětné/křižné hraně).
- Jedna programová lokace se může vyskytovat ve vícero vnitřních uzlech.
- Strom může obsahovat nekonečné cesty.

Path explosion problem

- Pro netriviální programy je počet větví stromu symbolické exekuce obrovský.
- Počet cest roste exponenciálně vzhledem k počtu průchodů větvícími lokacemi programu.

Použití symbolické exekuce ve verifikaci

Analýza stromu symbolické exekuce

- Prohledávání do šířky, strom je potencionálně nekonečný.

Informace získané o programu

- Určení proveditelných a neproveditelných cest.
- Detekce dosažitelnosti dané lokace.
- Detekce chyb (dělení nulou, přístup mimo pole, porušení invariantu lokace, atd.).

Syntéza testovacích dat

- Je-li podmínka cesty pro nějaký symbolický běh splnitelná, modelem této formule jsou konkrétní vstupní hodnoty programu, které si vynutí výpočet programu podle dané cesty.
- Syntéza testů zvyšující pokrytí kódu.

Automatické generování testů

Princip

1. Vygenerujeme náhodné vstupní hodnoty (náhodná cesta).
2. S danými hodnotami provedeme průchod stromem symbolické exekuce a zaznamenáme podmínku cesty.
3. Z podmínky cesty vytvoříme novou podmínku cesty tím, že negujeme formuli vybraného větvení.
4. Najdeme vstupní data vyhovující nové podmínce cesty.
5. Opakujeme od bodu 2 (pokud nové testy zvyšují pokrytí).

Realizace

- Heuristiky pro výběr větvení, jehož podmínka bude negována.
- Augmentace kódu pro zaznamenání podmínky cesty.

Limity symbolické exekuce

Nerozhodnutelnost

- Použití kompletní aritmetiky na neomezených doménách implikuje obecnou nerozhodnutelnost problému splnitelnosti.
- Strom symbolické exekuce může být nekonečný (rozbalování cyklů s dynamickým počtem opakování).

Výpočetní náročnost

- Exploze cest.
- Algoritmy pro splnitelnost formulí na omezených doménách.

Omezení

- Jak realizovat operace nad nečíselnými proměnnými?
- Jak reprezentovat dynamické datové struktury?
- Jak symbolicky vyhodnotit volání externí funkce?

Sekce

Automatizace testu splnitelnosti

Problém SAT

Problém splnitelnosti – SAT

- Problém rozhodnout, zda existuje valuace boolovských proměnných formule výrokové logiky taková, že formule je v této valuaci pravdivá.

Vlastnosti

- Jeden z nejznámějších NP-úplných problémů.
- Pro daný problém není znám polynomiální algoritmus.
- Existující řešiče SAT jsou velmi efektivní a díky mnohým heuristikám zvládají řešit překvapivě velké instance problému.

Nástroj Z3

ZZZ aka **Z3**

- Nástroj vyvíjený v Microsoft Research.
- Řešič instancí problémů SAT a SMT.
- WWW interface — `http://www.rise4fun.com/Z3`
- Binární API pro použití v jiných aplikacích.

Rozhodněte pomocí Z3

- Je splnitelná formule $(a \vee \neg b) \wedge (\neg a \vee b)$?

Použití Z3 – SAT

Reformulace formule pro Z3

$(a \vee \neg b) \wedge (\neg a \vee b)$

```
(declare-const a Bool)
(declare-const b Bool)
(assert (and (or a (not b)) (or (not a) b)))
(check-sat)
(get-model)
```

Odpověď od Z3

```
sat
(model
  (define-fun b () Bool
    false)
  (define-fun a () Bool
    false)
)
```

Problém splnitelnosti v dané teorii – SMT

Satisfiability Modulo Theory – SMT

- Problém rozhodnout splnitelnost formule prvořádové logiky s rovností, predikáty a funkčními symboly kódující jednu či více zvolených teorií.
- Typicky používané teorie
  - Aritmetika celých a desetinných čísel.
  - Teorie datových struktur (seznamy, pole, bitové vektory, ...).

Jiný pohled (převzato z Wikipedie)

- Na SMT lze také nahlížet jako na jistou formu hledání řešení vyhovující sadě daných omezení, tudíž lze to také chápat jako jistý formalizovaný přístup k oblasti programování s omezeními (constraint programming).

Řešte pomocí Z3 `http://rise4fun.com/Z3/tutorial/guide`

- Existují celá nenulová čísla x a y taková, že y=x*(x-y)?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x y))))
(assert (not (= y 0)))
(check-sat)
(get-model)
```

- Existují celá nenulová čísla x a y taková, že y=x*(x-(y*y))?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x (* y y)))))
(assert (not (= x 0)))
(check-sat)
```

Splnitelnost a platnost

Pozorování

- Formule je platná právě když její negace není splnitelná.

Důsledek

- Řešiče SAT a SMT lze využít jako nástroje pro dokazování platnosti formulovaných tvrzení.

Syntéza modelu

- Řešiče SAT nejen rozhodují splnitelnost formulí, ale v případě splnitelnosti vrací požadovanou valuaci proměnných, pro níž je formule pravdivá.
- Na rozdíl od dokazovacích nástrojů tak poskytují "protipříklad" v případě neplatnosti dokazovaného tvrzení.

Konkolické Testování

Motivace

Problém

- Principielní nerozhodnutelnost proveditelnosti cesty.
- V praxi typicky nerozhodnutelnost znamená nesplnitelnost.
- Vynecháním těchto cest můžeme minout chybu.
- Provedením těchto cest můžeme nalézt nereálnou chybu.

Částečné řešení

- Současné použití konkrétních a symbolických hodnot vstupních proměnných a využití konkrétních hodnot pro rozhodnutí jinak nerozhodnutelné instance splnitelnosti.
- Heuristika.
- Zajímavý případ (korektní): UNKNOWN $\Longrightarrow$ SAT
- **Con**crete and Symb**olic Testing** = **Concolic Testing**

Hypotetická ukázka konkolického testování

Program

```
1 input A,B
2 if (A==(B*B)%30) then
3   ERROR
4 else
5   return A
```

Konkolické testování

1. A=22, B=7 (náhodné hodnoty)
2. (22==(7*7)%30) je *False*, podmínka cesty: $\alpha \neq (\beta * \beta)\%30$
3. Test dopadl OK
4. Syntéza dat z negace PC: $\alpha = (\beta * \beta)\%30$ – **UNKNOWN**
5. Využití konkrétních hodnot: $\alpha = (7 * 7)\%30$ – **SAT**, $\alpha = 19$
6. A=19, B=7
7. Test odhalil chybovou lokaci na řádku 3.

Nástroj SAGE

Příběh nástroje SAGE

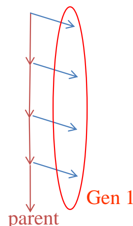
Systematic Testing for Security: Whitebox Fuzzing

Patrice Godefroid
Michael Y. Levin and David Molnar

http://research.microsoft.com/projects/atg/
Microsoft Research

Whitebox Fuzzing (SAGE tool)

- Start with a well-formed input (not random)
- Combine with a generational search (not DFS)
  - Negate 1-by-1 each constraint in a path constraint
  - Generate many children for each parent run
  - Challenge all the layers of the application sooner
  - Leverage expensive symbolic execution
- Search spaces are huge, the search is partial… yet effective at finding bugs !



Example: Dynamic Test Generation

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt > 3) crash();
}
```

input = "good"

Dynamic Test Generation

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt > 3) crash();
}
```

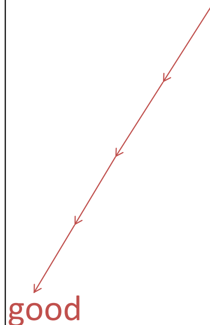
```
input = "good"

Path constraint:

I0 != 'b'
I1 != 'a'
I2 != 'd'
I3 != '!'
```

Negate a condition in path constraint
Solve new constraint → new input

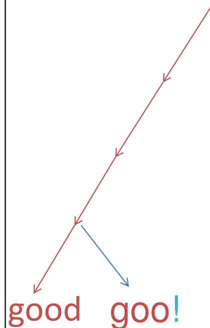
Depth-First Search




```
input = "good"

void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;    I0 != 'b'
    if (input[1] == 'a') cnt++;    I1 != 'a'
    if (input[2] == 'd') cnt++;    I2 != 'd'
    if (input[3] == '!') cnt++;    I3 != '!'
    if (cnt > 3) crash();
}
```

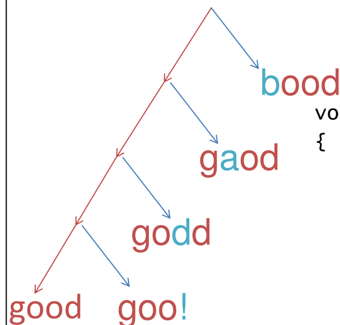
Depth-First Search




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;   I0 != 'b'
    if (input[1] == 'a') cnt++;   I1 != 'a'
    if (input[2] == 'd') cnt++;   I2 != 'd'
    if (input[3] == '!') cnt++;   I3 == '!'
    if (cnt > 3) crash();
}
```

Příběh nástroje SAGE

Generational Search



Four "Generation 1" test cases !

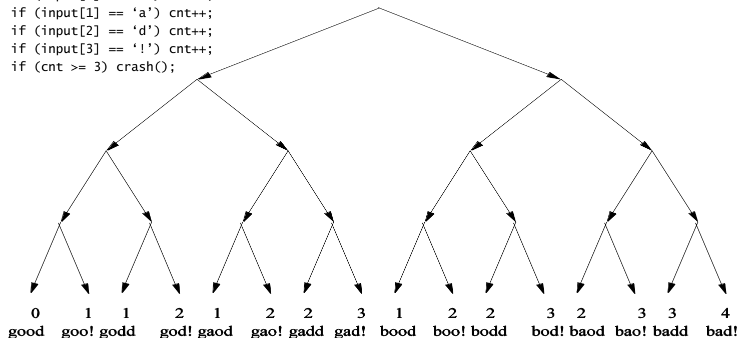
```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

I_0 == 'b'
I_1 == 'a'
I_2 == 'd'
I_3 == '!'

Příběh nástroje SAGE

The Search Space

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 3) crash();
}
```



Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 0 – seed file

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 00 00 00 00 00 ; RIFF............
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 1

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF....*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 2

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 3

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 00 00 00 00 ; ....strh........
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 4

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 5

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 00 00 00 00 ; ....strf........
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 6

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 7

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 C9 9D E4 4E ; ............ÉäN
00000060h: 00 00 00 00                                     ; ....
```

Generation 8

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 9

Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....strf²uv:(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 10 – crash bucket 1212954973!

Initial Experiences with SAGE

- Since 1st internal release in April'07: tens of new security bugs found
- Apps: image processors, media players, file decoders,... Confidential !
- Bugs: Write A/Vs, Read A/Vs, Crashes,... Confidential !
- Many bugs found triaged as "security critical, severity 1, priority 1"

IV113 Validace a verifikace

Formální verifikace algoritmů

Jiří Barnat

Verifikace algoritmů

Validace a Verifikace

- Jeden z obecných cílů V&V je prokázat správné chování algoritmů.

Připomenutí

- Proces testování je neúplný.
- Testováním dokážeme odhalit chybu, ale nedokážeme prokázat bezchybnost.

Závěr

- Je zapotřebí jiného způsobu verifikace systémů.

Formální verifikace

Cíl formální verifikace

- Cílem je prokázat, že systém pracuje správně takovým způsobem, aby míra důvěry ve výsledek procesu verifikace byla stejná, jako míra důvěry v matematický důkaz.

Nutné podmínky pro realizaci

- Formálně přesně definovaná sémantika chování systému.
- Formálně přesně stanovené požadavky na systém.

Formální metody verifikace

- Deduktivní verifikace
- Ověřování modelu (Model Checking)
- Abstraktní interpretace

Deduktivní verifikace

Korektnost programu

Program je korektní pokud

- Pokud pro platný vstup **skončí** a vrátí **korektní** výsledek.
- Dokazují se dvě tvrzení: že je program parciálně korektní, a že výpočet programu vždy skončí.

Parciální korektnost (Korektnost, Soundness)

- Pokud výpočet programu nad vstupními hodnotami, pro které je program definován, skončí, je výsledek výpočtu správný.

Terminace (Úplnost, Konvergence, Completeness)

- Pro vstupní hodnoty, pro které je program definován, výpočet programu skončí.

Sekvenční programy

- Vstup-výstupně uzavřené konečné programy.
  - Hodnoty vstupu jsou známy před vykonáním programu.
  - Výstup po skončení programu uložen ve výstupní proměnné.
- Quick sort, největší společný dělitel, ...

Princip verifikace

- Na programy a jednotlivé instrukce se nahlíží jako na transformátory stavů.
- Cílem je prokázat, že vstupní a výstupní hodnoty se k sobě mají v odpovídajícím vztahu.
- Tj. verifikovat korektnost postupu aplikovaného za účelem transformace vstupních proměnných na odpovídající výstupní proměnné.

Vyjadřování vlastností programů

Stav výpočtu

- Stav výpočtu programu je jednoznačně určen lokací (hodnotou čítače instrukcí) a valuací proměnných.

Atomické predikáty

- Základní tvrzení o jednotlivých stavech, jejichž pravdivost lze určit na základě konkrétní valuace proměnných.
- Příklady atomických propozic: $(x == 0)$, $(x1 >= y3)$.
- Pozor na rozsah platnosti odkazovaných proměnných!

Popis množiny stavů

- Specifikovány boolovskou kombinací atomických predikátů
- Příklad: $(x == m) \wedge (y > 0)$

Vyjadřování vlastností programů – Assertions

Assertion

- Pro dané místo programu (lokaci) definuje podmínku na valuace proměnných, jež jsou v dané lokaci programu návrhářem algoritmu očekávány.
- Invariant lokace programu.

Assertions – důkazy korektnosti

- Přiřazení vlastností jednotlivým místům grafu toku řízení.
- Robert Floyd: Assigning Meanings to Programs (1967)

Detekce chyb — Assertion Violation

Testování

- Porušení assertu jako orákulum.

Run-Time kontrola

- Ověřování invariantů lokací v době běhu programu.
- Případná chyba ve výpočtu, která vede k assertion violation, je snáze lokalizována díky vazbě invariantu na konkrétní lokaci v programu.

Neodhalené chyby

- Nevhodně zvolená vstupní data.
- Nedeterministické vykonávání programu (paralelismus).

Hoarův dokazovací systém

Hoarův dokazovací systém

Princip

- Programy = transformátory stavů.
- Specifikace = vztah mezi vstupním a výstupním stavem.

Hoarova logika

- Navržena za účelem ukazování parciální korektnosti programů.
- Nechť P a Q jsou predikáty a S program, pak

$$\{\mathrm{P}\}\ \mathrm{S}\ \{\mathrm{Q}\}$$

  je tzv. *Hoarova trojice*.

Zamýšlený význam trojice $\{P\}\ S\ \{Q\}$

- S je program, který transformuje stav splňující *vstupní podmínku* P na stav, který splňuje *výstupní podmínku* Q.

Příklad

- $\{z = 5\}\ x = z * 2\ \{x > 0\}$
- Platná trojice, výstupní podmínka by mohla být přesnější.
- Silnější výstupní podmínka: $\{x > 5 \wedge x < 20\}$, zjevně platí, že $\{x > 5 \wedge x < 20\} \implies \{x > 0\}$.

Nejslabší vstupní podmínka (weakest precondition)

- P je nejslabší vstupní podmínka, pokud
- platí $\{P\}S\{Q\}$ a zároveň
- $\forall P'$, pro které platí $\{P'\}S\{Q\}$, platí $P' \implies P$.

Postup dokazování {P} S {Q}

- Zvolíme vhodné podmínky P' a Q'
- Dokazujeme {P'} S {Q'}, P $\implies$ P' a Q' $\implies$ Q.
- V Hoarově důkazovém systému jsou axiomy a odvozovací pravidla, ze kterých se vytvářejí platné trojice pro strukturálně složitější programy.
- Pokud se podaří vyskládat transformaci P' na Q', tak {P'} S {Q'} je platná trojice.
- P $\implies$ P' a Q' $\implies$ Q **se dokazují běžným způsobem.**

Axiom

- Axiom pro přiřazení: $\{\phi[\text{x nahrazeno k}]\}\ x := k\ \{\phi\}$

Význam

- Trojice $\{P\}x := y\{Q\}$ je axiomem v Hoarově dokazovacím systému, pokud platí, že P je shodné s Q, ve kterém byly všechny výskyty x nahrazeny výrazem y.

Příklad

- {y+7>42} x:=y+7 {x>42} — je axiom
- {r=2} r:=r+1 {r=3} — není axiom
- {r+1=3} r:=r+1 {r=3} — je axiom

Příklad

- Dokažte, že následující program vrátí hodnotu větší než 0, pokud je spuštěn pro hodnotu 5.
- Program: $out := in * 2$

Důkaz

1) Formulujeme Hoarovu trojici:
   $\{in = 5\}\ out := in * 2\ \{out > 0\}$
2) Odvodíme vhodnou vstupní podmínku programu:
   $\{in * 2 > 0\}$
3) Dokážeme Hoarovu trojici:
   $\{in * 2 > 0\}\ out := in * 2\ \{out > 0\}$ (axiom)
4) Dokážeme pomocné tvrzení:
   $(in = 5) \implies (in * 2 > 0)$

Hoarův dokazovací systém – Příklad pravidla

Pravidlo

- Sekvenční kompozice: $\frac{\{\phi\}S_1\{\chi\} \wedge \{\chi\}S_2\{\psi\}}{\{\phi\}S_1;S_2\{\psi\}}$

Význam

- Jestliže S_1 transformuje stav splňující ϕ na stav splňující χ a S_2 transformuje stav splňující χ na stav splňující ψ, pak sekvence $S_1; S_2$ transformuje stav splňující ϕ na stav splňující ψ.

Dokazování

- Pro účely důkazu $\{\phi\}S_1; S_2\{\psi\}$ je nutné nalézt χ a ukázat $\{\phi\}S_1\{\chi\}$ a $\{\chi\}S_2\{\psi\}$.

Hoarův dokazovací systém – částečná korektnost

Axiom pro **skip**: $\{\phi\}$ **skip** $\{\phi\}$

Axiom pro **:=**: $\{\phi[x := k]\}x\mathbf{:=}k\{\phi\}$

Sekvenční kompozice: $\dfrac{\{\phi\}S_1\{\chi\}\wedge\{\chi\}S_2\{\psi\}}{\{\phi\}S_1;S_2\{\psi\}}$

Alternativa: $\dfrac{\{\phi\wedge B\}S_1\{\psi\}\wedge\{\phi\wedge\neg B\}S_2\{\psi\}}{\{\phi\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\{\psi\}}$

Iterace: $\dfrac{\{\phi\wedge B\}S\{\phi\}}{\{\phi\}\textbf{while } B \textbf{ do } S \textbf{ od } \{\phi\wedge\neg B\}}$

Důsledek: $\dfrac{\phi \Longrightarrow \phi', \{\phi'\}S\{\psi'\}, \psi' \Longrightarrow \psi}{\{\phi\}S\{\psi\}}$

Hoarova logika – Příklad 2

Dokažte, že pro $n \geq 0$ **počítá** $n!$**.**

```
r = 1;

while (n≠ 0) {
   r = r * n;

   n = n - 1;
}
```

Poznámky k důkazu:

Dokažte, že pro $n \geq 0$ **počítá** $n!$**.**

- $\{\ n \geq 0 \wedge t{=}n\ \}$ $\{P\}$

```
r = 1;

while (n≠ 0) {
   r = r * n;

   n = n - 1;
}
```

$\{\ r{=}t!\ \}$ $\{Q\}$

Poznámky k důkazu:

- Formulace dokazovaného jako Hoareho trojice.
- Všimněme si použití pomocné proměnné t.

Hoarova logika – Příklad 2

Dokažte, že pro $n \geq 0$ počítá $n!$.

- $\{\ n \geq 0 \wedge t{=}n\ \}$ $\{P\}$
  ```
  r = 1;
  ```
  $\{\ n \geq 0 \wedge t{=}n \wedge r = 1\ \}$ $\{I_1\}$
  ```
  while (n≠ 0) {
     r = r * n;

     n = n - 1;
  }
  ```
  $\{\ r{=}t!\ \}$ $\{Q\}$

Poznámky k důkazu:

- $\{n \geq 0 \wedge t{=}n \wedge 1{=}1\}$ r=1 $\{\ n \geq 0 \wedge t{=}n \wedge r{=}1\ \}$
- $(n \geq 0 \wedge t{=}n) \implies (n \geq 0 \wedge t{=}n \wedge 1{=}1)$

Dokažte, že pro $n \geq 0$ počítá $n!$.

```
{ n ≥ 0 ∧ t=n }                                        {P}
r = 1;
{ n ≥ 0 ∧ t=n ∧ r = 1 }                                {I1}
while (n≠ 0)   { r=t!/n! ∧ t ≥ n ≥ 0 }   {             {I2}
   r = r * n;

   n = n - 1;
}
{ r=t! }                                               {Q}
```

Poznámky k důkazu:

- Invariant cyklu: $\{I_2\} \equiv \{\ r{=}t!/n!\ \wedge t \geq n \geq 0\ \}$
- $I_1 \implies I_2 \qquad (\ I_2 \wedge \neg(n{\neq}0)\ ) \implies Q$

Dokažte, že pro $n \geq 0$ počítá $n!$.

```
{ n ≥ 0 ∧ t=n }                                        {P}
r = 1;
{ n ≥ 0 ∧ t=n ∧ r = 1 }                                {I1}
while (n≠ 0)   { r=t!/n! ∧ t ≥ n ≥ 0 }    {            {I2}
   r = r * n;
   { r=t!/(n-1)! ∧ t ≥ n > 0 }                         {I3}
   n = n - 1;
}
{ r=t! }                                               {Q}
```

Poznámky k důkazu:

- $\{ r*n = t!/(n\text{-}1)! \wedge t \geq n > 0 \}$ r=r*n $\{I_3\}$
- $I_2 \wedge (n \neq 0) \implies (r*n = t!/(n\text{-}1)! \wedge t \geq n > 0)$

Hoarova logika – Příklad 2

Dokažte, že pro $n \geq 0$ počítá $n!$.

```
{ n ≥ 0 ∧ t=n }                                   {P}
r = 1;
{ n ≥ 0 ∧ t=n ∧ r = 1 }                           {I1}
while (n≠ 0)   { r=t!/n! ∧ t ≥ n ≥ 0 }   {        {I2}
   r = r * n;
   { r=t!/(n-1)! ∧ t ≥ n > 0 }                    {I3}
   n = n - 1;
}
{ r=t! }                                          {Q}
```

Poznámky k důkazu:

- $\{\ r = t!/(n\text{-}1)!\ \wedge\ t \geq (n\text{-}1) \geq 0\ \}\ n{=}n\text{-}1\ \{I_2\}$
- $I_3 \implies (\ r = t!/(n\text{-}1)!\ \wedge\ t \geq (n\text{-}1) \geq 0\ )$

Hoareho logika a úplnost

Pozorování

- Díky Hoareho logice jsme převedli důkaz korektnosti programu na sadu matematických tvrzení, typicky využívající Peanovu aritmetiku.

Poznámka o korektnosti a (ne)úplnosti

- Hoarova logika je korektní, tj. pokud je možné dokázat $\{P\}S\{Q\}$, pak vykonání programu S ze stavu splňujícím P může skončit pouze ve stavu splňujícím Q.
- Je-li důkazový systém dostatečně silný na popis aritmetiky celých čísel, je nutně neúplný, tj. existují tvrzení, které nelze v systému dokázat a nelze dokázat ani jejich negaci.

Hoareho logika a obecně dokazování v praxi

Potíže s tvorbou důkazů

- Pro potřeby důkazu je často nutné vhodně zesílit vstupní podmínku nebo oslabit výstupní podmínku.
- Je velmi obtížné hledat invarianty cyklů.

Důkaz v praxi – částečná korektnost

- Často se zjednodušuje na konstatování invariantů a prokázání, že se skutečně jedná o invarianty cyklu (typicky indukcí).

Více o Hoarově logice na FI

- IV022 Návrh a verifikace algoritmů
- IA159 Formal Verification Methods

Prokazování terminace

Dobře založená doména

- Částečně uspořádaná množina prvků, ve které neexistuje nekonečně dlouhá klesající posloupnost.
- Příklady: $(\mathbf{N},<)$, $(\mathit{FinPowerSet}(\mathbf{N}),\subseteq)$

Prokázání terminace

- Každému vrcholu grafu toku řízení je přiřazen výraz, jež se vyhodnocuje nad společnou vhodně zvolenou dobře založenou doménou.
- Prokáže se, že hodnota výrazu asociovaného s vrcholem grafu toku řízení neroste podél žádné hrany grafu.
- Prokáže se, že hodnota výrazu asociovaného s vrcholem grafu toku řízení striktně klesá podél alespoň jedné hrany každého cyklu v grafu.

Sekce

Automatizace deduktivní verifikace

Princip automatizace deduktivní verifikace

Předzpracování

- Přepis programu (transformace) do vhodného mezi-jazyka.
- Příklady jazyků: Boogie (Microsoft Research), Why3 (INRIA)

Strukturální analýza a konstrukce kostry důkazu

- Nalezení trojic Hoareho logiky a invariantů cyklů.
- Vstupní a výstupní podmínky dodány spolu s verifikovaným programem.
- Generování znění pomocných lemat (proof obligations).

Řešení pomocných lemat

- Využití nástroju pro automatické dokazování k prokázání pomocných lemat.

Řešení pomocných lemat

Nástroje pro automatizované dokazování

- Uživatel vede nástroj k sestrojení důkazu požadovaného.
- HOL, ACL2, Isabelle, PVS, Coq, ...

Redukce na problém splnitelnosti negace

- Využití řešičů splnitelnosti.
- Z3, ...

Důkaz

- Konečná posloupnost transformací předpokladů ψ na požadovaný závěr φ s využitím axiomů daného dokazovacího systému a již dokázaných tvrzení.

Pozorování

- Pro systémy s konečným počtem axiomů a odvozovacích pravidel lze důkazy dané délky systematicky generovat, tj. pro platná tvrzení lze v konečném čase nalézt důkaz.
- Všechny rozumné dokazovací systémy mají nekonečně mnoho axiomů. Uvažme např. axiom $x = x$, ve skutečnosti je to pouze zkratka za axiomy 1=1, 2=2, 3=3, ...).
- Důkazy lze generovat, pokud všechny axiomy a pravidla lze alespoň enumerovat.

Hledání důkazu platného tvrzení

- Potenciálních konečných posloupností k ověření je příliš (nekonečně) mnoho.
- Obecně pro nalezení důkazu daného tvrzení v daném dokazovacím systému neexistuje algoritmus, je např. nerozhodnutelné, zda program zastaví pro každý vstup.
- Bez rozumné strategie, nelze očekávat nalezení důkazu platného tvrzení v rozumně krátké době.
- Strategie hledání důkazu je udávána uživatelem s odpovídající kvalifikací a matematickým cítěním.

Verifikace s nástrojem pro dokazování

Nástroje pro podporu dokazování (Theorem Provery)

- Cílem je pro danou množinu axiomů a důkazový systém nalézt důkaz daného tvrzení.
- Důkaz se hledá střídavě dvěma způsoby:
  - Algoritmický mód – aplikace odvozených pravidel a axiomů
    - Řízen uživatelem nástroje
    - Dedukce, resoluce, unifikace, přepisování, ...
  - Hledací mód – hledá nová platná tvrzení
    - Využití hrubé výpočetní síly.

Existující nástroje

- Popis dokazovacího systému, programu i požadovaného tvrzení probíhá ve vstupním jazyce dokazovacího nástroje.

Možné výsledky procesu dokazování

Výstup procesu dokazování

a) Důkaz nalezen a ověřen.
b) Důkaz nenalezen.
    - Tvrzení platí, lze dokázat, ale důkaz se nám nepodařilo nalézt.
    - Tvrzení platí, ale nelze jej v daném systému dokázat.
    - Tvrzení neplatí.

Pozorování

- V případě nenalezení důkazu metoda nedává žádnou nápovědu k tomu, proč se tvrzení nepodařilo dokázat.

Dafny

`http://rise4fun.com/dafny`

IV113 Validace a verifikace

Ověřování modelu pro Lineární Temporální Logiku

Jiří Barnat

Motivace

Ověřování kvality

- Testování je neúplné, nedává garanci správnosti.
- Deduktivní verifikace je drahá.

Typický kontext výskytu chyb

- Neočekávané či krajní vstupy.
- Interakce komponent.
- Paralelizmus (nelze testovat).

Model Checking – Ověřování modelu

- Automatizovaný proces verifikace pro ...
- ... paralelní a distribuované systémy.

Verifikace paralelních a reaktivních programů

Paralelní programy

Paralelní kompozice

- Komponenty souběžně přispívají k transformaci výchozího stavu na cílový.
- Významová funkce pro paralelně běžící programy vznikne libovolným proložením atomických akcí jednotlivých komponent. (Interleaving.)

Nekompozicionalita významových funkcí

- Významovou funkci paralelního programu nelze získat jako složení významových funkcí jednotlivých komponent.
- Výsledek může být závislý na proložení akcí.

Příklad nekompozicionality

Příklad

- Systém: (y=x; y++; x=y) ∥ (y=x; y++; x=y)
- Vstup-výstupní proměnná x
- Významová funkce obou procesů je λx->x+1.
- Složení významových funkcí: (λx->x+1)·(λx->x+1).
- (λx->x+1)·(λx->x+1) 0 = 2

2 konkrétní běhy

- Stav $= (x, y_1, y_2)$
- (0,-,-) $\xrightarrow{y_1=x}$ (0,0,-) $\xrightarrow{y_2=x}$ (0,0,0) $\xrightarrow{y_1++}$ $\xrightarrow{x=y_1}$ (1,1,0) $\xrightarrow{y_2++}$ $\xrightarrow{x=y_2}$ (**1**,1,1)
- (0,-,-) $\xrightarrow{y_1=x}$ (0,0,-) $\xrightarrow{y_1++}$ $\xrightarrow{x=y_1}$ (1,1,-) $\xrightarrow{y_2=x}$ (1,1,1) $\xrightarrow{y_2++}$ $\xrightarrow{x=y_2}$ (**2**,1,2)

Vlastnosti paralelních programů

Pozorování

- Konkrétní časování událostí souvisejících s interakcí programů je forma vstupu.
- Paralelní programy jsou svým způsobem reaktivní systémy, neboť nejsou dopředu známa všechna vstupní data.

Důsledek

- U paralelních a reaktivních programů často požadujeme (specifikujeme) chování, která se nedají vyjádřit s použitím vstupních a výstupních podmínek.

Vlastností paralelních/reaktivních programů

Příklady specifikace

- Události A a B nastanou dříve, než událost C.
- Uživateli program není dovoleno vložit novou hodnotu vstupu, dokud program přechozí hodnotu nezpracuje.
- Není pravda, že procedura X bude souběžně vykonávána procesem P i Q (vzájemné vyloučení).
- Každá akce A vyvolá sekvenci reakcí B,C a D.

Formalizace slovního popisu

- Lze formalizovat pomocí temporálních logik.
- Amir Pnueli, 1977

Dokazování pomocí formulí temporálních logik

Pozorování

- Pro modální logiky je možné vystavět podobné důkazové systémy, jako pro Hoarovu logiku.
- Tyto důkazové techniky vykazují stejné/podobné nevýhody jako verifikace paralelních programů s využitím vstupních a výstupních podmínek.

Model checking

- Jiná metoda ověření platnosti formule temporální logiky.

Ověřování modelu
(Model Checking)

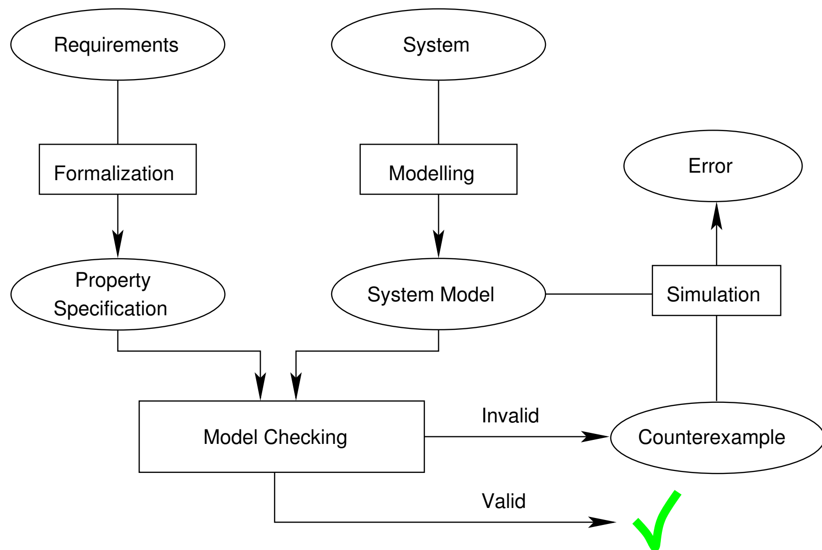
Ověřování modelu – přehled

- Vytvoříme formální model $\mathcal{M}$ verifikovaného systému.
- Specifikaci vyjádříme formulí φ zvolené temporální logiky.
- Rozhodneme, zda $\mathcal{M} \models \varphi$. Tj., zda $\mathcal{M}$ je modelem formule φ. (Odtud název ověřování modelu.)

Volitelné

- Postranním produktem rozhodnutí může být (případně větvící se) posloupnost stavů, dokládající rozhodnutí. Tato posloupnost se běžně označuje slovem protipříklad (často produkována pouze za účelem ukázání neplatnosti φ).
- Proces rozhodnutí lze pro konečné (a některé nekonečné) modely systému automatizovat.

Ověřování modelu – schéma



Automatické nástroje pro ověřování modelu

Model-checkery

- Softwarové nástroje, které pro model systému a temporální vlastnost provedou rozhodnutí o splnění dané vlastnosti modelem.
- SPIN, UppAll, SMV, Prism, DIVINE . . .

Modelovací jazyky

- Procesy popsány jako rozšířené konečné automaty.
- Rozšíření umožňuje podmínit provedení přechodu/akce platností boolovského výrazu, případně synchronizací s akcí jiného souběžně běžícího procesu.

Modelování a formalizace verifikovaného systému

Formalizace stavového prostoru – Atomické propozice

Připomenutí

- Na systém lze nahlížet jako na množinu stavů, které se mění vlivem vykonávání akcí programu.
- Stav = valuace modelovaných proměnných.

Atomické propozice

- Základní tvrzení vyjadřujících se o kvalitách jednotlivých stavů. (Např. $max(x, y) \geq 3, \ldots$)
- Platnost atomických propozic musí být algoritmicky rozhodnutelná na základě daného stavu, tj. s využitím valuace modelovaných proměnných.
- Množina základních o stavu pozorovatelných jevů je ovlivněna mírou abstrakce použitou při modelování systému.

Kripkeho struktury

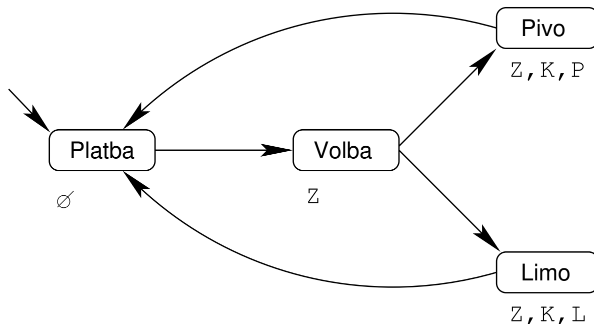
- Nechť je dána množina atomických propozic AP.
- Kripkeho struktura je čtveřice (S, T, I, s_0), kde
  - S je (konečná) množina stavů,
  - $T \subseteq S \times S$ je přechodová relace,
  - $I : S \to 2^{AP}$ je interpretace AP.
  - $s_0 \in S$ je počáteční stav.

Kripkeho přechodové systémy

- Je-li dána množina Act akcí proveditelných programem, je možné Kripkeho struktury rozšířit o označení přechodů.
- Kripkeho přechodový systém je pětice $(S, T, I, s_0, \mathcal{L})$, kde
  - (S, T, I, s_0) je Kripkeho struktura,
  - $\mathcal{L} : T \to Act$ je značkovací funkce.

Kripkeho struktura – příklad

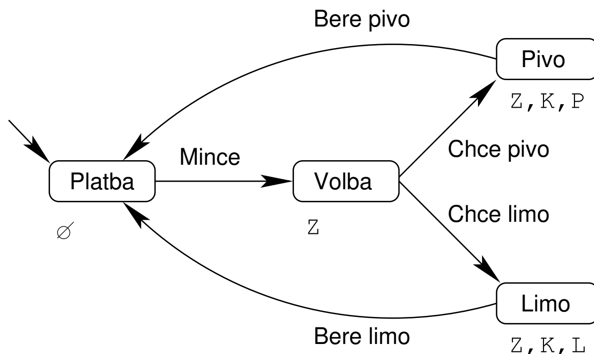
Kripkeho struktura



AP={Z – zaplaceno, K – kelímek, L – limo, P – pivo}

Kripkeho struktura – příklad

Kripkeho přechodový systém



AP={Z – zaplaceno, K – kelímek, L – limo, P – pivo}

Formalizace stavového prostoru – běh systému

Běh

- Maximální (tj. nerozšiřitelný) sled v grafu indukovaného Kripkeho strukturou, počínající v iniciálním stavu Kripkeho struktury.
- Nechť $M = (S, T, I, s_0)$ je Kripkeho struktura.
- Běh je posloupnost stavů $\pi = s_0, s_1, s_2, \ldots$ taková, že $\forall i \in \mathbb{N}_0.(s_i, s_{i+1}) \in T$.

Konečné běhy – úmluva

- Maximální sled může být konečný: $\pi = s_0, s_1, s_2, \ldots, s_k$, pokud $\nexists s_{k+1} \in S.(s_k, s_{k+1}) \in T$.
- Z technických důvodu lze na konečné maximální sledy nahlížet jako na nekonečné běhy, které vzniknou opakováním posledního stavu daného maximálního sledu.
- Maximální sled $s_0, \ldots, s_k$ se chápe jako běh $s_0, \ldots, s_k, s_k, s_k, \ldots$.

Implicitní a explicitní popis systému

Pozorování

- Kripkeho struktura, jež udává konkrétní chování systému, se často nepopisuje výčtem stavů a hran (explicitní forma), ale pouze programem (implicitní forma).
- Implicitní zápis může být exponenciálně úspornější.

Generování stavového prostoru

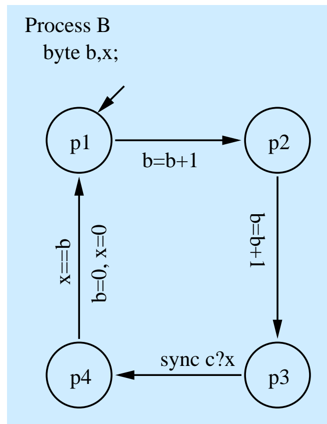
- Výpočet explicitní reprezentace z implicitního popisu.
- Interpretace implicitního zápisu sleduje přesně formálně definovaná pravidla.

Praxe

- Programovací jazyky nemají přesnou sémantiku.
- Využívají se umělé modelovací jazyky.

Příklad modelovacího jazyka – DVE

- Konečný automat
  - Stavy (Lokace)
  - Iniciální stav
  - Přechody
  - (Akceptující stavy)
- Přechody rozšířené o
  - Stráž
  - Synchronizaci s předáváním hodnot
  - Efekt (přiřazení)
- Lokální proměnné
  - `integer`, `byte`
  - `channel`



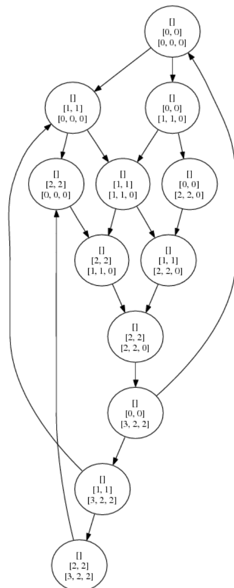
Příklad modelu v jazyce DVE

```
channel {byte} c[0];

process A {
byte a;
state q1,q2,q3;
init q1;
trans
q1→q2 { effect a=a+1; },
q2→q3 { effect a=a+1; },
q3→q1 { sync c!a; effect a=0; };
}

process B {
byte b,x;
state p1,p2,p3,p4;
init p1;
trans
p1→p2 { effect b=b+1; },
p2→p3 { effect b=b+1; },
p3→p4 { sync c?x; },
p4→p1 { guard x==b; effect b=0, x=0; };
}

system async;
```



Sémantika ukázána simulací

State: []; A:[q1, a:0]; B:[p1, b:0, x:0]
0 ⟨0.0⟩: q1 → q2 { effect a = a+1; }
1 ⟨1.0⟩: p1 → p2 { effect b = b+1; }
Command:1

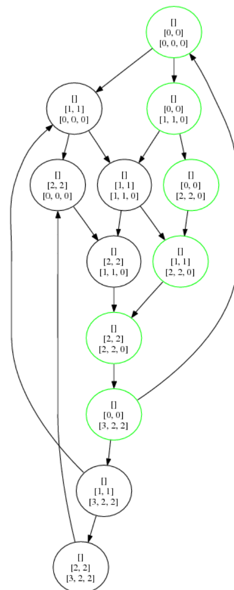
State: []; A:[q1, a:0]; B:[p2, b:1, x:0]
0 ⟨0.0⟩: q1 → q2 { effect a = a+1; }
1 ⟨1.1⟩: p2 → p3 { effect b = b+1; }
Command:1

State: []; A:[q1, a:0]; B:[p3, b:2, x:0]
0 ⟨0.0⟩: q1 → q2 { effect a = a+1; }
Command:0

State: []; A:[q2, a:1]; B:[p3, b:2, x:0]
0 ⟨0.1⟩: q2 → q3 { effect a = a+1; }
Command:0

State: []; A:[q3, a:2]; B:[p3, b:2, x:0]
0 ⟨0.2&1.2⟩: q3 → q1 { sync c!a; effect a = 0; }
p3 → p4 { sync c?x; }
Command:0

State: []; A:[q1, a:0]; B:[p4, b:2, x:2]



Formalizace vlastností

Specifikace jako jazyky nekonečných slov

Problém

- Jak se formálně vyjadřovat o kvalitách běhu?
- Jak mechanicky rozhodovat, že běh má danou kvalitu?

Řešení

- Konečný automat jako mechanický pozorovatel běhu.
- Běh je nekonečný!
- Konečné automaty pro jazyky nekonečných slov (ω-regulární jazyky)
- Büchi akceptační podmínka – automat akceptuje slovo pokud nekonečněkrát projde koncovým stavem.

Automaty nad nekonečnými slovy

Büchi automaty

- Büchi automat je pětice $A = (\Sigma, S, s, \delta, F)$, kde
  - Σ je konečná abeceda znaků,
  - S je konečná množina stavů,
  - $s \in S$ je iniciální stav,
  - $\delta : S \times \Sigma \to 2^S$ je přechodová relace, a
  - $F \subseteq S$ je množina koncových stavů.

Jazyk akceptovaný Büchi automatem A

- Běh ρ automatu A nad nekonečným slovem $w = a_1 a_2 \ldots$ je sekvence stavů $\rho = s_0, s_1, \ldots$ taková, že $s_0 \equiv s$ a $\forall i : s_i \in \delta(s_{i-1}, a_i)$.
- $inf(\rho)$ – množinu stavů, které se v ρ vyskytly nekonečně krát.
- Běh ρ je akceptující, pokud $inf(\rho) \cap F \neq \emptyset$.
- Jazyk akceptovaný automatem A je množina všech slov, pro které existuje akceptující běh. Označujeme $L(A)$.

Přehlednost zápisu/nákresu

Pozorování

- AP={X,Y,Z},
- Hrana označená $\{X\}$ značí, že platí X a neplatí Y a Z.
- Pokud je třeba vyjádřit že platí X, neplatí Z a na platnosti Y nezáleží, je třeba vést hrany pod $\{X\}$ a $\{X, Y\}$.

Množiny AP jako boolovské formule

- Jednotlivé hrany mezi dvěma stejnými vrcholy pod různými kombinacemi atomických propozic lze sloučit do jedné hrany ohodnocené boolovskou formulí.

Příklad

- Hrany {X}, {Y}, {X,Y}, {X,Z}, {Y,Z} a {X,Y,Z} lze nahradit jednou hranou ohodnocenou formulí $X \vee Y$.
- Pokud vůbec nezáleží na platnosti při provedení hrany, je možné ji označit štítkem $\mathit{true} \equiv X \vee \neg X$.

Vyjádřete Büchi automatem

Systém

- Prodejní automat
- $\Sigma = 2^{\{Z,K,L,P\}}$,
- $jeZ = \{A \in \Sigma \mid Z \in A\}$, $jeK = \{A \in \Sigma \mid K \in A\}$, ...

Vlastnosti

- Prodejní automat vydá alespoň jeden kelímek s nápojem.
- Prodejní automat vydá alespoň jeden kelímek s limonádou.

- Prodejní automat vydá nekonečně mnoho kelímků.
- Prodejní automat vydá nekonečně mnoho kelímků s pivem.

- Prodejní automat nevydá kelímek s nápojem, bez zaplacení.
- Pokaždé, když zaplatím, dostanu kelímek s nápojem.

Lineární temporální logika

LTL neformálně

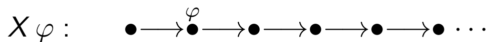
Formule φ

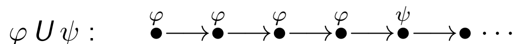
- Vyhodnocuje se nad jedním během systému.
- Vyjadřuje se o platnosti atomických propozic ve stavech daného běhu.

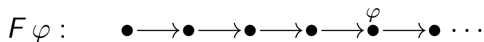
Temporální operátory LTL

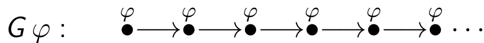
- $F\,\varphi$ — (Future) Někde v běhu platí φ.
- $G\,\varphi$ — (Globally) V daném běhu vždy platí φ.
- $\varphi\,U\,\psi$ — (Until) Někde platí ψ a do té doby platí φ.
- $X\,\varphi$ — (Next) V příštím stavu platí φ.
- $\varphi\,W\,\psi$ — (Weak Until) Jako Until ale ψ nemusí nastat.
- $\varphi\,R\,\psi$ — (Release) ψ platí dokud neplatí $\varphi \wedge \psi$.

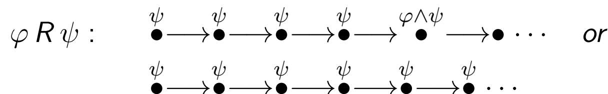
Grafické znázornění sémantiky temporálních operátorů

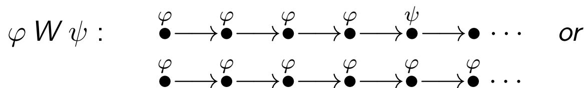












Syntax LTL

Nechť AP je množina atomických propozic. Pak

- Je-li $p \in AP$, pak p je formule.
- Je-li φ formule, pak $\neg\varphi$ je formule.
- Jsou-li φ a ψ formule, pak $\varphi \vee \psi$ je formule.
- Je-li φ formule, pak $X\,\varphi$ je formule.
- Jsou-li φ a ψ formule, pak $\varphi\,U\,\psi$ je formule.

Alternativní zápis

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid X\,\varphi \mid \varphi\,U\,\varphi$$

Syntaktické zkratky

Výroková logika

- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$
- $\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$
- $\varphi \Leftrightarrow \psi \equiv (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

Temporální operátory

- $F\,\varphi \equiv \mathit{true}\; U\,\varphi$
- $G\,\varphi \equiv \neg F\,\neg\varphi$
- $\varphi\, R\, \psi \equiv \neg(\neg\varphi\, U\, \neg\psi)$
- $\varphi\, W\, \psi \equiv \varphi\, U\, \psi \vee G\,\varphi$

Alternativní syntax

- $F\varphi \equiv \diamond\varphi$
- $G\varphi \equiv \Box\varphi$
- $X\varphi \equiv \circ\varphi$

Modely formulí LTL

Model LTL formule

- Je dána množina atomických propozic AP.
- Modelem LTL formule je běh π Kripkeho struktury.

Značení

- Nechť $\pi = s_0, s_1, s_2, \ldots$.
- Suffix běhu π počínaje stavem s_k budeme značit jako $\pi^k = s_k, s_{k+1}, s_{k+2}, \ldots$.
- K-tý stav běhu π, budeme značit jako $\pi(k) = s_k$.

Sémantika LTL

Předpoklady

- Je dána množina atomických propozic AP.
- Je dán běh π Kripkeho struktury $M = (S, T, I, s_0)$.
- φ, ψ jsou syntakticky správné LTL formule.
- $p \in AP$ je atomická propozice.

Sémantika

$$
\begin{array}{rcl}
\pi \models p & \text{iff} & p \in I(\pi(0)) \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi \vee \psi & \text{iff} & \pi \models \varphi \text{ or } \pi \models \psi \\
\pi \models X\,\varphi & \text{iff} & \pi^1 \models \varphi \\
\pi \models \varphi\, U\, \psi & \text{iff} & \exists k. 0 \leq k, \pi^k \models \psi \text{ and} \\
 & & \forall i. 0 \leq i < k, \pi^i \models \varphi
\end{array}
$$

Sémantika temporálních operátorů

$$\pi \models F\,\varphi \quad \text{iff} \quad \exists k. k \geq 0, \pi^k \models \varphi$$

$$\pi \models G\,\varphi \quad \text{iff} \quad \forall k. k \geq 0, \pi^k \models \varphi$$

$$\begin{aligned} \pi \models \varphi\,R\,\psi \quad \text{iff} \quad & (\exists k. 0 \leq k, \pi^k \models \varphi \wedge \psi \text{ and} \\ & \forall i. 0 \leq i < k, \pi^i \models \psi) \\ & \text{or } (\forall k. k \geq 0, \pi^k \models \psi) \end{aligned}$$

$$\begin{aligned} \pi \models \varphi\,W\,\psi \quad \text{iff} \quad & (\exists k. 0 \leq k, \pi^k \models \psi \text{ and} \\ & \forall i. 0 \leq i < k, \pi^i \models \varphi) \\ & \text{or } (\forall k. k \geq 0, \pi^k \models \varphi) \end{aligned}$$

Verifikace s použitím LTL

- Na systém nahlížíme jako na množinu možných běhů.
- Systém splňuje vlastnost specifikovanou LTL formulí pokud (a jen tehdy) všechny běhy systému začínající v iniciálním stavu systému splňují danou formuli.
- Kterýkoliv běh systému, který začíná v iniciálním stavu a nesplňuje danou formuli může být dokladem toho, že systém nesplňuje specifikovanou vlastnost.

Tvrzení

- Pokud konečně stavový systém nesplňuje vlastnost specifikovanou formulí LTL, tak to lze dokladovat během π, který lze vyjádřit jako $\pi = \pi_1 \cdot (\pi_2)^\omega$, kde

  $\pi_1 = s_0, s_1, \ldots, s_k$
  $\pi_2 = s_{k+1}, s_{k+2}, \ldots, s_{k+n}$, kde $s_k \equiv s_{k+n}$.

- Běhy s uvedenou vlastností nazýváme *lasa* (lasso shape).

LTL Model Checking s využitím teorie formálních jazyků a automatů

Spojitost s jazyky nekonečných slov

Pozorování 1

- Systém je množina (nekonečných) běhů.
- Na systém lze nahlížet jako na jazyk nekonečných slov.

Pozorování 2

- Dva různé běhy (různé posloupnosti stavů) jsou z hlediska platnosti dané formule ekvivalentní, pokud se shodují v interpretaci atomických proměnných.
- Jestliže $\pi = s_0, s_1, \ldots$, pak $I(\pi) \stackrel{def}{\Longleftrightarrow} I(s_0), I(s_1), I(s_2), \ldots$.

Pozorování 3

- Každý běh danou formuli buď splňuje, anebo nesplňuje.
- Každá LTL formule vymezuje množinu splňujících běhů.

Redukce problému na jazykovou inkluzi

Problém

- Je dána množina AP, Kripkeho struktura $M = (S, T, I, s_0)$ a specifikace LTL formulí φ.
- Splňuje systém M specifikaci φ? ($M \models \varphi$)

Jazyky nekonečných slov

- Nechť $\Sigma = 2^{AP}$.
- Jazyk L_{sys} všech běhů systému M:

$$L_{sys} = \{I(\pi) \mid \pi \text{ je běh v } M\}.$$

- Jazyk L_φ všech běhů splňující formuli φ:

$$L_\varphi = \{I(\pi) \mid \pi \models \varphi\}.$$

Pozorování

- Systém M splňuje specifikaci φ právě když $L_{sys} \subseteq L_\varphi$.

L_{sys} a L_φ vyjádřeny Büchi automatem

Tvrzení 1

- Pro každou LTL formuli φ existuje (a lze efektivně sestrojit) Büchi automat A_φ takový, že $L_\varphi = L(A_\varphi)$.
- [Vardi, Wolper 1986]

Tvrzení 2

- Pro každou Kripkeho strukturu $M = (S, T, I, s_0)$ lze sestrojit Büchi automat A_{sys} takový, že $L_{sys} = L(A_{sys})$.
- Konstrukce A_{sys}
  - Nechť AP je množina atomických propozic.
  - Pak $A_{sys} = (S, 2^{AP}, s_0, \delta, S)$, kde
  - $q \in \delta(p, a)$ právě když $(p, q) \in T \wedge I(p) = a$.

Synchronní kompozice Büchi automatů – zjednodušená

Tvrzení

- Nechť $A = (S_A, \Sigma, s_A, \delta_A, F_A)$ a $B = (S_B, \Sigma, s_B, \delta_B, F_B)$ jsou Büchi automaty nad shodnou abecedou Σ. Pak existuje (lze efektivně sestrojit) Büchi automat $A \times B$ takový, že $L(A \times B) = L(A) \cap L(B)$.

Pozorování

- Büchi automat A_{sys} je zkonstruován tak, že $F_A = S_A$, tj. má všechny stavy akceptující.

Konstrukce $A \times B$ pro případ, že $F_A = S_A$

- $A \times B = (S_A \times S_B, \Sigma, (s_A, s_B), \delta_{A\times B}, S_A \times F_B)$
- $(p', q') \in \delta_{A\times B}((p, q), a)$ pro všechna

  $p' \in \delta_A(p, a)$
  $q' \in \delta_B(q, a)$

- V plné obecnosti je konstrukce $A \times B$ složitější.

Redukce na problém prázdnosti Büchi automatu

Tvrzení

- Pro každou LTL formuli φ platí: $co{-}L(A_\varphi) = L(A_{\neg\varphi})$.

Redukce $M \models \varphi$ na problém prázdnosti $L(A_{sys} \times A_{\neg\varphi})$

- $M \models \varphi \iff L_{sys} \subseteq L_\varphi$
- $M \models \varphi \iff L(A_{sys}) \subseteq L(A_\varphi)$
- $M \models \varphi \iff L(A_{sys}) \cap co{-}L(A_\varphi) = \emptyset$
- $M \models \varphi \iff L(A_{sys}) \cap L(A_{\neg\varphi}) = \emptyset$
- $M \models \varphi \iff L(A_{sys} \times A_{\neg\varphi}) = \emptyset$

Redukce problému na detekci akceptujícího cyklu

Tvrzení

- Büchi automat $A = (S, \Sigma, s_0, \delta, F)$ akceptuje neprázdný jazyk právě když existují stav $s \in F$ a slova $w_1, w_2 \in \Sigma^*$ taková, že $s \in \hat{\delta}(s_0, w_1)$ a $s \in \hat{\delta}(s, w_2)$.
- Tj. graf Büchi automatu obsahuje dosažitelný akceptující cyklus (cyklus přes nějaký akceptující stav).

Rozhodovací procedura pro problém $M \models \varphi$?

- Zkonstruuje se produktový automat $(A_{sys} \times A_{\neg\varphi})$.
- Graf produktového automatu se prověří na přítomnost akceptujících cyklů.
- Obsahuje-li graf akceptující cyklus, pak $M \not\models \varphi$.
- Neobsahuje-li graf akceptující cyklus, pak $M \models \varphi$.

Domácí úkol

Připomenutí

- Pro hodnocení stupněm A, nutno realizovat všechny domácí úlohy.

Domácí úkol

- Sestudujte způsob převodu LTL formule na ekvivalentní Buchi automat a implementujte ve Vámi zvoleném programovacím jazyce.
- Kontrola-demonstrace bude realizována během ústního zkoušení.

IV113 Validace a verifikace

Detekce akceptujícího cyklu

Jiří Barnat

Připomenutí

Problém

- Kripkeho struktura M
- LTL formule φ
- $M \models \varphi$?

Řešení pomocí Büchiho automatů

- A_{sys} – automat akceptující běhy modelu
- $A_{\neg\varphi}$ – automat akceptující běhy porušující vlastnost φ
- $L(A_{sys}) \cap L(A_{\neg\varphi}) = L(A_{sys} \times A_{\neg\varphi})$
- $L(A_{sys} \times A_{\neg\varphi}) \neq \emptyset \iff$ model má běh porušující φ
- $L(A_{sys} \times A_{\neg\varphi}) = \emptyset \iff M \models \varphi$

Algoritmus pro detekci akceptujících cyklů

Vstup algoritmu

- Produktový automat ve formě tří následujících funkcí
  - $|F|_init()$ — Vrací iniciální stav automatu.
  - $|F|_succs(s)$ — Pro daný stav vrací jeho přímé následníky.
  - $|Accepting|(s)$ — O stavu řekne, zda je či není akceptující.

Výstup algoritmu

- Přítomen / Nepřítomen
- Protipříklad.

Algoritmus

- Využívá vnořené prohledávání do hloubky – Nested DFS.
- Vnější procedura detekuje akceptující stavy, vnitřní procedura testuje, zda akceptující stav je dosažitelný ze sebe sama (leží na cyklu).

Detekce akceptujících cyklů

Problém

- Je dán Büchiho automat $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$.
- Je jazyk akceptovaný automatem $\mathcal{A}$ neprázdný?

Redukce na detekci akceptujícího cyklu v grafu

- Nechť $G = (S, E)$, kde
  $E = \{(u, v) \in S \times S \mid \exists a \in \Sigma \text{ takové, že } v \in \delta(u, a)\}$
  je graf Büchiho automatu.
- $L(\mathcal{A})$ je neprázdný právě když graf automatu $\mathcal{A}$ obsahuje dosažitelný akceptující cyklus, tj. cyklus jehož alespoň jeden vrchol v je akceptující ($v \in F$) a zároveň dosažitelný z iniciálního stavu ($(s_0, v) \in E^*$).

Algoritmické řešení

1) V grafu Büchiho automatu identifikuj všechny dosažitelné akceptující vrcholy. (Vnější procedura.)
2) Pro každý takto identifikovaný vrchol ověř, že není dosažitelný ze sebe sama. (Vnitřní procedura.)

Dosažitelnost v grafu

- Standardní grafový algoritmus.
- Výpočet množiny dosažitelných, případně akceptujících dosažitelných vrcholů lze provést v čase $\mathcal{O}(|V| + |E|)$.
- S obecným algoritmem detekce dosažitelnosti je detekce přítomnosti akceptujícího cyklu proveditelná v čase $\mathcal{O}(|V| + |E| + |F|(|V| + |E|))$.
- Pokud ale použijeme strategii prohledávání do hloubky, lze dosáhnout času $\mathcal{O}(|V| + |E|)$.

Prohledávání grafu do hloubky

```
proc Reachable(V,E,v0)
  Visited = ∅
  DFS(v0)
  return (Visited)
end

proc DFS(vertex)
  if vertex ∉ Visited
    then /* Visits vertex */
      Visited := Visited ∪ {vertex}
      foreach { v | (vertex,v)∈ E } do
        DFS(v)
      od
      /* Backtracks from vertex */
  fi
```

Barevné značení vrcholů při DFS

Pozorování

- Při prohledávání grafu procedurou `DFS` se každý vrchol grafu nachází v jednom ze tří možných stavů.

Barevné značení vrcholů

- Bílý vrchol - dosud nebyl navštíven.
- Šedý vrchol - navštívený, ale dosud nebyl backtrackován.
- Černý vrchol - navštívený i backtrackovaný.

Zásobník rekurze

- Šedé vrcholy tvoří cestu od počátečního vrcholu k vrcholu, který je v daný okamžik algoritmem zpracováván.

Vlastnosti DFS, $G = (V, E)$ a $v_0 \in V$

Pozorování

- Pokud pro dva různé vrcholy v_1, v_2 platí, že
  - $(v_0, v_1) \in E^*$,
  - $(v_1, v_1) \notin E^+$,
  - $(v_1, v_2) \in E^+$.
- Pak procedura $\mathtt{DFS}(v_0)$ backtrackuje z vrcholu v_2 dříve než backtrackuje z vrcholu v_1.

DFS post-order

- Pokud $(v, v) \notin E^+$ a $(v_0, v) \in E^*$, pak po skončení procedury $\mathtt{DFS}(v)$, volané v rámci výpočtu $\mathtt{DFS}(v_0)$, jsou všechny vrcholy u takové, že $(v, u) \in E^+$ navštívené a backtrackované.

Detekce akceptujících cyklů v čase $\mathcal{O}(|V| + |E|)$

Předpoklady

- Volání vnitřní procedury pro akceptující vrchol v ohlásí cyklus a ukončí algoritmus, pokud akceptující vrchol v leží na akceptujícím cyklu.

Klíčová myšlenka

- Vnořené procedury jsou volány v pořadí, ve kterém vnější procedura z akceptujících vrcholů backtrackuje, tj. dle DFS post-orderu.

Detekce akceptujících cyklů v čase $\mathcal{O}(|V| + |E|)$

```
proc Detekce akceptujících cyklů
  Visited := ∅
  DFS(v0)
end

proc DFS(vertex)
  if (vertex) ∉ Visited
    then Visited := Visited ∪ {vertex}
    foreach {s | (vertex,s) ∈ E} do
      DFS(s)
    od
    if IsAccepting(vertex)
      then DetectCycle(vertex)
    fi
  fi
end
```

Detekce akceptujících cyklů v čase $\mathcal{O}(|V| + |E|)$

Pozorování

- Pokud podgraf dosažitelný z vrcholu s neobsahuje akceptující cyklus, pak žádný akceptující cyklus přes vrchol r ležící mimo podgraf dosažitelný z s neprochází stavem dosažitelným z s.

Tvrzení

- Pokud vnitřní procedura pro vrchol v skončí aniž by ohlásila cyklus, tak podgraf dosažitelný z vrcholu v neobsahuje akceptující cyklus.

Důsledek vedoucí k lineárnímu algoritmu

- Každou vnitřní proceduru lze omezit na vrcholy dosud nenavštívené v žádné předchozí vnitřní proceduře.

$\mathcal{O}(|V| + |E|)$ **algoritmus**

1) Vnitřní procedury budou spouštěny pro akceptující vrcholy v pořadí, ve kterém vnější procedura z těchto vrcholů backtrackuje.
2) Vnitřní procedury nenavštěvují vrcholy navštívené v předchozích vnitřních procedurách.

Detekce cyklu ve vnitřních procedurách

Tvrzení

- Pokud je následníkem právě zpracovávaného vrcholu šedý vrchol (tj. vrchol na zásobníku rekurzivních volání procedury *DFS*), pak graf obsahuje cyklus.

Využití

- Ve vnořené proceduře není nutné dosáhnout přesně vrcholu, pro který je detekce cyklu volána, ale stačí dosáhnout vrcholu na zásobníku vnější procedury.

$\mathcal{O}(|V| + |E|)$ Algoritmus

```
proc Detekce akceptujících cyklů
  Visited := Nested := in_stack := ∅
  DFS(v_0)
  Exit("Nepřítomen")
end
```

```
proc DFS(vertex)
  if (vertex) ∉ Visited
    then Visited := Visited ∪ {vertex}
    in_stack := in_stack ∪ {vertex}
    foreach {s | (vertex,s) ∈ E} do
      DFS(s)
    od
    if IsAccepting(vertex)
      then DetectCycle(vertex)
    fi
    in_stack := in_stack \ {vertex}
  fi
end
```

```
proc DetectCycle (vertex)
  if vertex ∉ Nested
    then Nested := Nested ∪ {vertex}
    foreach {s | (vertex,s) ∈ E} do
      if s ∈ in_stack
        then WriteOut(in_stack)
          Exit("Přítomen")
        else DetectCycle(s)
      fi
    of
  fi
end
```

Složitost

Vnější procedura

- Časová složitost: $\mathcal{O}(|V| + |E|)$
- Prostorová složitost: $\mathcal{O}(|V|)$

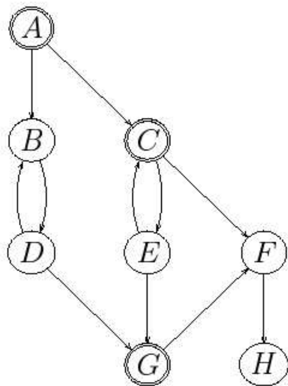
Vnitřní procedury

- Celková časová složitost: $\mathcal{O}(|V| + |E|)$
- Prostorová složitost: $\mathcal{O}(|V|)$

Složitost

- Časová složitost: $\mathcal{O}(|V|+|E|+|V|+|E|) = \mathcal{O}(|V|+|E|)$
- Prostorová složitost: $\mathcal{O}(|V| + |V|) = \mathcal{O}(|V|)$

DFS – příklad



- 1st DFS: A,B,D,B,G,F,H,H,F,G
  1st DFS stack: A,B,D,G
  visited: A,B,D,F,G,H / –
- 2nd DFS: G,F,H,H,F,G
  visited: A,B,D,F,G,H / F,G,H
- 1st DFS: G,D,B,C,E,C,G,E,F,C
  1st DFS stack: A,C
  visited: all / F,G,H
- 2nd DFS: C,E,C
  counterexample: A,C,E,C

visited state backtrack non-accepting state backtrack accepting state

Typy Büchi automatů a jejich využití při verifikaci

Podtřídy Büchi automatů

Terminální Büchi automaty

- Všechny akceptující cykly v automatu jsou právě ve formě smyčky nad akceptujícím stavem strážené výrazem `true`.

Slabé Büchi automaty (weak)

- Každá silně souvislá komponenta automatu je striktně tvořena buď pouze akceptujícími stavy, nebo pouze neakceptujícími.

Dopad na verifikační proceduru

Automat $A_{\neg\varphi}$

- Pro řadu LTL formulí φ je $A_{\neg\varphi}$ terminální nebo slabý.
- $A_{\neg\varphi}$ je typicky velmi malý (max desítky stavů).
- Typ $A_{\neg\varphi}$ lze zjistit před procesem verifikace.
- Typy komponent $A_{\neg\varphi}$
  - Neakceptující – bez akceptujícího cyklu.
  - Striktně akceptující – každý cyklus je akceptující.
  - Smíšené – obsahuje akceptující i neakceptující cykly.

Produktový automat

- Prohledávaný graf je synchronní produkt A_S a $A_{\neg\varphi}$.
- Typ komponent $A_S \times A_{\neg\varphi}$ určen odpovídajícími komponentami $A_{\neg\varphi}$.

Dopad na verifikační proceduru – Terminální BA

$A_{\neg\varphi}$ je terminální Büchi automat

- Pro důkaz existence akceptujícího cyklu stačí prokázat dosažitelnost stavu akceptujícího ve složce $A_{\neg\varphi}$.
- Proces verifikace se redukuje na analýzu dosažitelnosti.

„Safety" vlastnosti

- Vlastnost φ, pro které je $A_{\neg\varphi}$ terminální BA.
- Typická slovní formulace: „Nenastane špatná událost."
- Pro verifikaci stačí analýza dosažitelnosti.

Dopad na verifikační proceduru – Slabé BA

$A_{\neg\varphi}$ je slabý Büchi automat

- Neobsahuje smíšené komponenty.
- Pro důkaz existence akceptujícího cyklu stačí prokázat dosažitelnost cyklu v akceptující komponentě.
- Lze detekovat pomocí jednoduchého DFS.
- Znám optimální algoritmus, který nevyžaduje DFS.

„Slabé" LTL vlastnosti

- Vlastnost φ, pro které je $A_{\neg\varphi}$ slabý BA.
- Typická vlastnost je „response": $G\,(a \implies F\,(b))$

Klasifikace LTL formulí

Klasifikace

- Každá LTL formule patří do jedné z následujících tříd:
  Reactivity, Recurrence, Persistance, Obligation, Safety, Guarantee

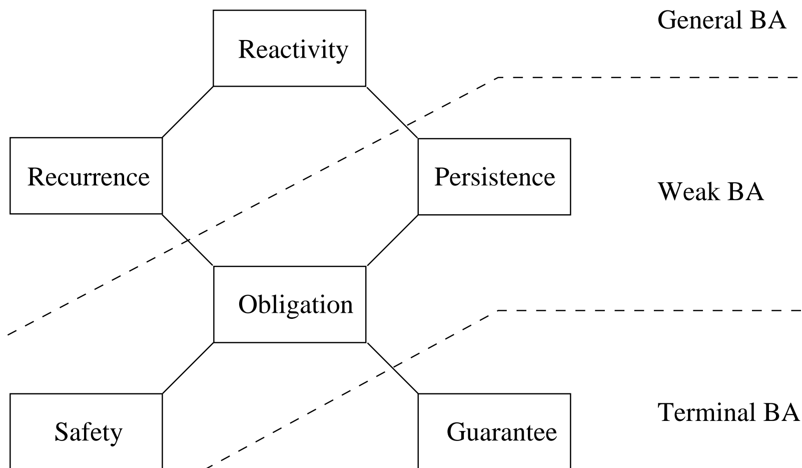
Zajímavá fakta

- Je-li vlastnost ze třídy Guarantee, je popsatelná terminálním Büchi automatem.
- Je-li vlastnost ze tříd Persistance, Obligation nebo Safety je popsatelná slabým Büchi automatem.

Při verifikaci se formule neguje ($\varphi \mapsto A_{\neg\varphi}$)

- $\varphi \in$ Safety $\iff$ $\neg\varphi \in$ Guarantee.
- $\varphi \in$ Recurrence $\iff$ $\neg\varphi \in$ Persistance.

Klasifikace LTL formulí



Boj se stavovou explozí

Stavová exploze

Co je to stavová exploze

- Systém bývá popsán jako paralelní kompozice procesů.
- Vzájemným proložením možných chování jednotlivých procesů vznikají různé možné stavy systému jako celku.
- Počet dosažitelných stavů systému může být až exponenciálně větší než součet počtu stavů procesů.

Důsledek

- Do operační paměti počítače nezle uložit všechny stavy produktového automatu.
- Je obtížně detekovat přítomnost akceptujícího cyklu.

Vybrané metody boje se stavovou explozí

Komprese stavových vektorů

- Neztrátové komprese
- Ztrátové – heuristiky

On-The-Fly verifikace

Symbolická reprezentace stavového prostoru

Redukce počtu stavů produktového automatu

- Redukce zaváděním atomických bloků
- Redukce částečným uspořádáním akcí
- Redukce symetrií

Paralelní/Distribuovaná verifikace

On-The-Fly verifikace

Pozorování

- Graf lze zadat pomocí funkcí (tzv. implicitní definice)
  - $|F|_init()$ — Vrací iniciální vrchol grafu.
  - $|F|_succs(s)$ — Pro daný vrchol vrací jeho přímé následníky.
  - $|Accepting|(s)$ — O stavu řekne, zda je či není akceptující.

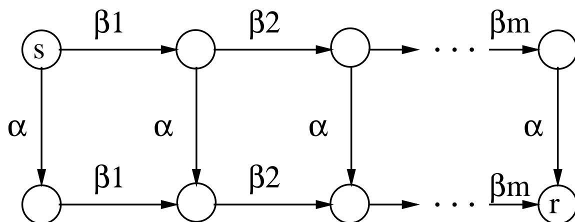
On-the-fly verifikace

- Pokud graf obsahuje akceptující cyklus, algoritmus je schopen v některých případech detekovat přítomnost akceptujícího cyklu, aniž by přitom navštívil (a tedy uložil) všechny vrcholy grafu.
- Problém $\mathcal{M} \models \varphi$ je někdy možné rozhodnout bez nutnosti úplné enumerace vrcholů a hran automatu $A_{sys} \times A_{\neg\varphi}$.
- Metoda verifikace s výše popsanou vlastností se označuje jako on-the-fly metoda verifikace.

Příklad

- Uvažme systém tvořený dvěma procesy A a B.
- A je tvořen jednou akcí α, B sekvencí akcí $\beta_1, \ldots, \beta_m$.

Neredukovaný stavový prostor:

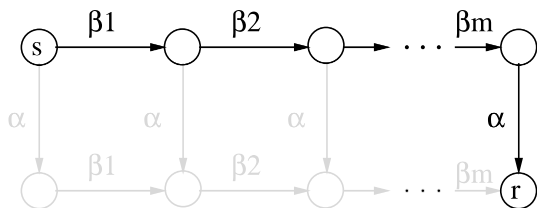


Vlastnost: Je dosažitelný stav r?

Redukce uspořádáním akcí

Pozorování

- Běhy $(\alpha\beta_1\beta_2\ldots\beta_m)$, $(\beta_1\alpha\beta_2\ldots\beta_m)$, ..., $(\beta_1\beta_2\ldots\beta_m\alpha)$ jsou vzhledem k dané vlastnosti ekvivalentní.
- Je dostačující zvážit pouze jednoho reprezentanta z třídy ekvivalence, například $(\beta_1\beta_2\ldots\beta_m\alpha)$.



- Reprezentanta lze získat odkládáním akce α.

Redukce uspořádáním akcí

Princip redukce

- Při generování grafů se místo funkce všech přímých následníků, uvažují pouze někteří přímí následníci vrcholu.
- V důsledku toho je počet vygenerovaných stavů produktového automatu menší.

Technická realizace

- Optimální způsob výběru následníků je obtížný.
- Nástroje implementují různé heuristiky.
- Redukovaný stavový prostor musí zachovávat přítomnost akceptujícího cyklu.
- Formule nesmí obsahovat operátor X (podtřída LTL).

Anglický název

- Partial Order Reduction

Princip

- Nesnaží se zmenšit paměťové nároky výpočtu.
- Snaží se efektivně využít větší množství výpočetních zdrojů.

Problémy algoritmu Nested DFS

- Algoritmus přistupuje operační paměť velmi náhodně. Prosté swapování na disk nefunguje (výprask OS).
- Simulace Nested DFS algoritmu v prostředí s distribuovanou pamětí je pomalá (předávání peška).
- Není znám způsob jak DFS post-order napočítat efektivně paralelním algoritmem. (Otevřený problém.)

Paralelní algoritmy pro detekci cyklu

Pozorování

- Místo Nested DFS algoritmu se používají jiné (časově neoptimální) algoritmy, jejichž výpočet ale lze dobře paralelizovat.

	Složitost	Optimalita	On-The-Fly
Nested DFS	O(V+E)	Ano	Ano
OWCTY			
obecné Büchi automaty	O(V.(V+E))	Ne	Ne
slabé Büchi automaty	O(V+E)	Ano	Ne
MAP	O(V.V.(V+E))	Ne	Částečně
OWCTY+MAP			
obecné Büchi automaty	O(V.(V+E))	Ne	Částečně
slabé Büchi automaty	O(V+E)	Ano	Částečně

Ověřování modelu – shrnutí

Princip rozhodovací procedury a stavová exploze

Platnost formule

- Požadovaná vlastnost může být porušena při jednom jediném konkrétním proložení akcí.
- Rozhodnutí je podloženo analýzou grafu stavového prostoru verifikovaného programu.

Stavová exploze

- Není-li řečeno jinak, je třeba uvážit všechna možná proložení akcí.
- Počet stavů, do kterých se může dostat paralelní program, je exponenciálně větší než velikost zápisu paralelního programu.

Výhody metody ověřování modelu

Obecná technika aplikovatelná na různé typy systémů

- Hardware, software, zabudované systémy, ...

Garance výsledku (matematicky podložen)

- Rozhodovací procedura tvrdí, že $\mathcal{M} \models \varphi$, tehdy a jen tehdy, pokud to skutečně platí.

Existence podpůrných nástrojů – "model checkerů"

- Verifikace tzv. na kliknutí myši / zmačknutí tlačítka.
- Minimální účast uživatele v rozhodovacím procesu.
- Automatická identifikace a generování protipříkladů.

Nevýhody metody ověřování modelu

Vhodná pouze k verifikaci konkrétních transformací

- Nelze použít na obecný důkaz toho, že program například pro zadané n spočítá hodnotu $n!$.
- Lze však ověřit, že pro hodnotu 5 program vrátí 120.

Verifikuje pouze model systému

- Platnost formule, neznamená splnění specifikace systémem.
- Nutnost vytváření modelu.

Velikost stavového prostoru

- Aplikovatelné (pouze) na konečně stavové systémy.
- Vzhledem k stavové explozi je praktická aplikovatelnost techniky omezena na relativně malé modely.

Verifikuje jen to, co je specifikováno

- Co není vyjádřeno formulemi, to se neverifikuje.

IV113 Validace a verifikace

Převod LTL formule na Büchi automat

Jiří Barnat

Připomenutí

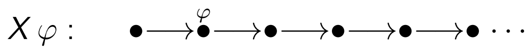
Problém

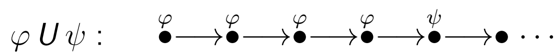
- Kripkeho struktura M
- LTL formule φ
- $M \models \varphi$?

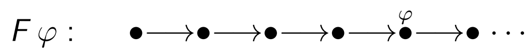
Řešení pomocí Büchi automatů

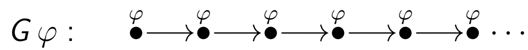
- A_{sys} – automat akceptující běhy modelu
- $A_{\neg\varphi}$ – automat akceptující běhy porušující vlastnost φ
- $L(A_{sys}) \cap L(A_{\neg\varphi}) = L(A_{sys} \times A_{\neg\varphi})$
- $L(A_{sys} \times A_{\neg\varphi}) \neq \emptyset \iff$ model má běh porušující φ
- $M \models \varphi \iff A_{sys} \times A_{\neg\varphi}$ nemá akceptující cyklus

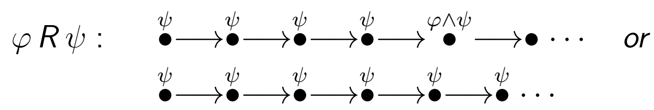
Grafické znázornění sémantiky temporálních operátorů

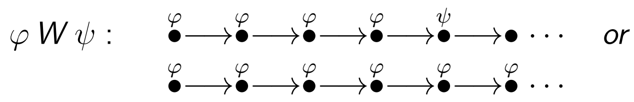












Příklad

Nechť $\Sigma = \{a, b, c\}$, najděte Büchi automat, který akceptuje ω-regulární jazyk definovaný následující LTL formulí.

a) $a\,U\,b$
b) $a\,U\,(X\,b)$
c) $\neg(a\,U\,(X\,b))$
d) $a\,U\,(b\,U\,c)$
e) $\neg(a\,U\,(b\,U\,c))$

Algoritmus převodu LTL formule na Büchi automat

Postup převodu

Vstup: Množina atomických propozic AP, LTL formule φ.
Výstup: Büchi automat A takový, že $L(A) = L_\varphi$.

Postup:

- Formule φ se převede do normální formy.
- Vypočítá se přechodový graf budoucího automatu.
- Graf se doplní na zobecněný Büchi automat.
- Zobecněný BA se převede na standardní Büchi automat.

Normální forma LTL formule

Řekneme, že LTL formule je v normální formě, pokud neobsahuje operátory F a G, a všechny operátory unární negace jsou aplikovány na podformule tvořené pouze atomickou propozicí.

Syntax

$$\varphi ::= p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid X\,\varphi \mid \varphi\, U\, \varphi \mid \varphi\, R\, \varphi$$

Pravidla pro převod do normální formy

$$\begin{aligned}
\neg(\varphi \vee \psi) &\equiv (\neg\varphi) \wedge (\neg\psi) \\
\neg(\varphi \wedge \psi) &\equiv (\neg\varphi) \vee (\neg\psi) \\
\neg X\varphi &\equiv X(\neg\varphi) \\
\neg(\varphi\, U\, \psi) &\equiv (\neg\varphi\, R\, \neg\psi) \\
\neg(\varphi\, R\, \psi) &\equiv (\neg\varphi\, U\, \neg\psi)
\end{aligned}$$

Příklad

Nechť $AP = \{a, b\}$. Převeďte následující LTL formule do normální formy

a) $G(F(a))$
b) $F(G(a))$
c) $\neg(G(F(a))$
d) $G(a \implies F(b))$
e) $\neg(a\, U\, (G\, b))$

Zobecněné Büchi automaty

Büchi automaty

- $A = (\Sigma, S, s, \delta, F)$
- $F \subseteq S$ je množina koncových stavů.
- Běh ρ je akceptující, pokud $inf(\rho) \cap F \neq \emptyset$.

Zobecněné Büchi automaty

- $A = (\Sigma, S, s, \delta, \mathcal{F})$
- $\mathcal{F} \subseteq 2^S$ je systém množin koncových stavů.
- Běh ρ je akceptující, pokud $\forall F_i \in \mathcal{F}$ platí $inf(\rho) \cap F_i \neq \emptyset$.

Příklad

Nechť $\Sigma = \{a, b\}$ a $L = \{w \in \Sigma^\omega \mid \mathit{inf}(w) = \{a, b\}\}$.

Najděte zobecněný BA $\mathcal{A}$ takový, že $L(\mathcal{A}) = L$.

Tvrzení

- Ke každému zobecněnému Büchi automatu A existuje (normální) Büchi automat B takový, že $L(A) = L(B)$.

Konstrukce

- Nechť $A = (\Sigma, S, s, \delta, \{F_1, \ldots, F_n\})$.
- $B = (\Sigma, S \times \{0, \ldots, n\}, (s, 0), \delta', S \times \{n\})$, kde
- $(q, y) \in \delta'((p, x), a)$ pokud $q \in \delta(p, a)$ a pro x a y platí
  - jestliže $q \in F_i$ a $x = i - 1$, tak $y = i$
  - jestliže $x = n$, tak $y = 0$
  - jinak $x = y$.

Příklad ZBA $\rightarrow$ BA

ZBA: $\mathcal{F} = \{F_1 = \{p\}, F_2 = \{q\}\}$

	a	b
$\rightarrow$ p	p	q
q	p	q

BA: $F = \{(p, 2), (q, 2)\}$

	a	b
$\rightarrow$ (p,0)	(p,1)	(q,0)
(q,0)	(p,1)	(q,0)
(p,1)	(p,1)	(q,2)
(q,1)	(p,1)	(q,2)
$\leftarrow$ (p,2)	(p,0)	(q,0)
$\leftarrow$ (q,2)	(p,0)	(q,0)

Výpočet přechodového grafu

Přechodový graf intuitivně

Pozorování

- Přechod v Büchi automatu stráží jeden stav běhu.
- Pro definici přechodu, je třeba vědět, co platí v aktuálním stavu, a co má platit v následujícím stavu běhu.

Rozbalené definice modálních operátorů

$$
\begin{array}{lcll}
X\,a & \equiv & tt & \wedge \quad X(\,a\,) \\
a\,U\,b & \equiv & a & \wedge \quad X(\,a\,U\,b\,) \\
 & \vee & b & \wedge \quad X(\,tt\,) \\
a\,R\,b & \equiv & b & \wedge \quad X(\,a\,R\,b\,) \\
 & \vee & a \wedge b & \wedge \quad X(\,tt\,)
\end{array}
$$

Přechodový graf intuitivně

Pozorování

- Přechod v Büchi automatu stráží jeden stav běhu.
- Pro definici přechodu, je třeba vědět, co platí v aktuálním stavu, a co má platit v následujícím stavu běhu.

Rozbalené definice modálních operátorů

New		Now	Next
$X\,a$	$\equiv$		a
$a\,U\,b$	$\equiv$	a	$a\,U\,b$
	$\vee$	b	
$a\,R\,b$	$\equiv$	b	$a\,R\,b$
	$\vee$	$a \wedge b$	

Přechodový graf LTL formule

Uzel je uspořádaná pětice

- Id – Číslo
  - Unikátní označení uzlu.
- Incoming – Množina označení uzlů.
  - Množina přímých předchůdců vrcholu ve výsledném grafu.
  - Kóduje hrany grafu.
- Now – Množina LTL formulí.
  - Seznam podformulí, které platí v daném uzlu.
- New – Množina LTL formulí.
  - Množina ještě nezpracovaných formulí, které musí být splněny v tomto uzlu.
- Next – Množina LTL formulí.
  - Seznam formulí, které musí být splněny v následujícím uzlu.

Graf LTL formule

Vytvoření grafu (výpočet množiny uzlů)

proc $create_graph(\varphi)$
  $N = (new_ID(), \{init\}, \emptyset, \{\varphi\}, \emptyset)$
  return $expand(N, \emptyset)$
end

Pomocné funkce

- $expand(n, Nodes)$
  - Funkce volaná pro uzel n a dosud známe uzly $Nodes$.
  - Vrací množinu uzlů (po zpracování uzlu n).
- $new_ID()$
  - Každé volání této funkce vrátí dosud nevrácené číslo.
- $Neg(_)$
  - $Neg(A) = \neg A$ pro všechny $A \in AP$.
  - $Neg(True) = False$
  - $Neg(False) = True$
  - $\neg\neg A = A$

Graf LTL formule – funkce expand

```
proc expand(q, Nodes)
  if New(q) == ∅
    then if (∃r ∈ Nodes takový, že Now(r) == Now(q) ∧ Next(r) == Next(q))
           then Incoming(r) = Incoming(r) ∪ Incoming(q)
                return Nodes
           else N = (new_ID(), {ID(q)}, ∅, Next(q), ∅)
                return expand(N, Nodes ∪ {q})          /* q je nový uzel */
         fi
    else let η ∈ New(q)
         New(q) = New(q) ∖ {η}
         if η ∈ Now(q)                                /* η již byla zpracována */
           then return expand(q, Nodes)
         fi
         switch (η)    /* pokračuj podle typu nejvnějšnějšího operátoru η */
         ...
         end
  fi
end
```

Graf LTL formule – funkce expand (switch)

```
switch (η)                    /* pokračuj podle typu nejvnějšnějšího operátoru η */

  case (η ∈ (AP ∪ Neg(AP) ∪ {True, False}))
    if (η == False ∨ Neg(η) ∈ Now(q))
      then return Nodes
       else N = (new_ID(), Incoming(q), Now(q) ∪ {η}, New(q), Next(q))
            return expand(N, Nodes)
    fi
  end

  case (η ≡ φ U ψ)
    N1 = (new_ID(), Incoming(q),
            Now(q) ∪ {η}, New(q) ∪ {φ}, Next(q) ∪ {φ U ψ})
    N2 = (new_ID(), Incoming(q),
            Now(q) ∪ {η}, New(q) ∪ {ψ}, Next(q))
    return expand(N2, expand(N1, Nodes))
  end

  . . .
```

Graf LTL formule – funkce expand (switch)

case $(\eta \equiv \varphi\, R\, \psi)$
  N1 = $(\textit{new_ID}(), \textit{Incoming}(q),$
    $\textit{Now}(q) \cup \{\eta\}, \textit{New}(q) \cup \{\varphi, \psi\}, \textit{Next}(q))$
  N2 = $(\textit{new_ID}(), \textit{Incoming}(q),$
    $\textit{Now}(q) \cup \{\eta\}, \textit{New}(q) \cup \{\psi\}, \textit{Next}(q) \cup \{\varphi\, R\, \psi\})$
  return $\textit{expand}(\mathrm{N2}, \textit{expand}(\mathrm{N1}, \mathrm{Nodes}))$
end

case $(\eta \equiv \varphi \vee \psi)$
  N1 = $(\textit{new_ID}(), \textit{Incoming}(q),$
    $\textit{Now}(q) \cup \{\eta\}, \textit{New}(q) \cup \{\varphi\}, \textit{Next}(q))$
  N2 = $(\textit{new_ID}(), \textit{Incoming}(q),$
    $\textit{Now}(q) \cup \{\eta\}, \textit{New}(q) \cup \{\psi\}, \textit{Next}(q))$
  return $\textit{expand}(\mathrm{N2}, \textit{expand}(\mathrm{N1}, \mathrm{Nodes}))$
end

. . .

Graf LTL formule – funkce expand (switch)

```
case (η ≡ φ ∧ ψ)
   N = (new_ID(), Incoming(q),
          Now(q) ∪ {η}, New(q) ∪ {φ, ψ}, Next(q))
   return expand(N, Nodes)
end

case (η ≡ X φ)
   N = (new_ID(), Incoming(q),
          Now(q) ∪ {η}, New(q), Next(q) ∪ {φ})
   return expand(N, Nodes)
end

end                                          /* end of switch */
```

Příklad výpočet grafu pro formuli X(a)

Vypočtené uzly

Id:	2
Incoming:	init
Now:	X(a)
New:	$\emptyset$
Next:	a

Id:	4
Incoming:	2
Now:	a
New:	$\emptyset$
Next:	$\emptyset$

Id:	5
Incoming:	4,5
Now:	$\emptyset$
New:	$\emptyset$
Next:	$\emptyset$

Výpočet

Id	Incoming	Now	New	Next
1	init	$\emptyset$	$\{X(a)\}$	$\emptyset$
2	init	$\{X(a)\}$	$\emptyset$	$\{a\}$

Uzel 2 je nově vypočtený uzel.

Id	Incoming	Now	New	Next
3	2	$\emptyset$	$\{a\}$	$\emptyset$
4	2	$\{a\}$	$\emptyset$	$\emptyset$

Uzel 4 je nově vypočtený uzel.

Id	Incoming	Now	New	Next
5	4	$\emptyset$	$\emptyset$	$\emptyset$

Uzel 5 je nově vypočtený uzel.

Id	Incoming	Now	New	Next
6	5	$\emptyset$	$\emptyset$	$\emptyset$

Uzel 6 je shodný s uzlem 5.
$\text{Incoming}(5) = \text{Incoming}(5) \cup \{5\}$

Příklad $a\,U\,(b\,U\,c)$

01| in, ∅, {aU(bUc)}, ∅
02| in, {aU(bUc)}, {a}, {aU(bUc)}
03| in, {aU(bUc)}, {bUc}, ∅
04| in, {aU(bUc),a}, ∅, {aU(bUc)}
Uzel 04 je nově vypočtený uzel.
05| 04, ∅, {aU(bUc)}, ∅
06| 04, {aU(bUc)}, {a}, {aU(bUc)}
07| 04, {aU(bUc)}, {bUc}, ∅
08| 04, {aU(bUc),a}, ∅, {aU(bUc)}
Uzlu 04 je přidán předchůdce 04.

07| 04, {aU(bUc)}, {bUc}, ∅
09| 04, {aU(bUc),bUc}, {b}, {bUc}
10| 04, {aU(bUc),bUc}, {c}, ∅
11| 04, {aU(bUc),bUc,b}, ∅, {bUc}
Uzel 11 je nově vypočtený uzel.
12| 11, ∅, {bUc}, ∅
13| 11, {bUc}, {b}, {bUc}
14| 11, {bUc}, {c}, ∅
15| 11, {bUc,b}, ∅, {bUc}
Uzel 15 je nově vypočtený uzel.
16| 15, ∅ {bUc}, ∅
17| 15, {bUc}, {b}, {bUc}
18| 15, {bUc}, {c}, ∅
19| 15, {bUc,b}, ∅, {bUc}
Uzlu 15 je přidán předchůdce 15.

Příklad $a\ U\ (b\ U\ c)$ – pokračování

18| 15, {bUc}, {c}, $\emptyset$
20| 15, {bUc,c}, $\emptyset$, $\emptyset$
Uzel 20 je nově vypočtený uzel.
21| 20, $\emptyset$, $\emptyset$, $\emptyset$
Uzel 21 je nově vypočtený uzel.
22| 21, $\emptyset$, $\emptyset$, $\emptyset$
Uzlu 21 je přidán předchůdce 21.

14| 11, {bUc}, {c}, $\emptyset$
23| 11, {bUc,c}, $\emptyset$, $\emptyset$
Uzlu 20 je přidán předchůdce 11.

10| 04, {aU(bUc),bUc}, {c}, $\emptyset$
24| 04, {aU(bUc),bUc,c}, $\emptyset$, $\emptyset$
Uzel 24 je nově vypočtený uzel.
25| 24, $\emptyset$, $\emptyset$, $\emptyset$
Uzlu 21 je přidán předchůdce 24.

Příklad $a\,U\,(b\,U\,c)$ – pokračování

03| in, {aU(bUc)}, {bUc}, $\emptyset$
26| in, {aU(bUc),bUc}, {b}, {bUc}   27| in, {aU(bUc),bUc}, {c}, $\emptyset$
28| in, {aU(bUc),bUc,b}, $\emptyset$, {bUc}
Uzlu 11 je přidán předchůdce *in*.

27| in, {aU(bUc),bUc}, {c}, $\emptyset$
29| in, {aU(bUc),bUc,c}, $\emptyset$, $\emptyset$
Uzlu 24 je přidán předchůdce *in*.

Převedení grafu na zobecněný Büchi automat

Předpoklady

- Dána množina AP.
- *Nodes* je množina vrcholů grafu LTL formule.

Zobecněný Büchi automat $A = (S, \Sigma, \delta, init, \mathcal{F})$

- $S = Nodes \cup \{init\}$
- $\Sigma = 2^{AP}$
- $r' \in \delta(r, \alpha)$ pokud
    - $r \in Incoming(r')$, $\alpha \in \Sigma$
    - α splňuje omezení dané množinou $((AP \cup \neg AP) \cap Now(r'))$
- $\mathcal{F} = \{F_1, \ldots, F_n\}$
    - Pro každou podformuli ve tvaru $\varphi\, U\, \psi$ definujeme F_i.
    - $F_i = \{r \in Nodes \mid \psi \in Now(r) \vee \varphi U \psi \notin Now(r)\}$.

Příklad $a\,U\,(b\,U\,c)$ – Zobecněný Büchi automat

Přechodová funkce (stráže)

	a	b	c	tt
→ init	04	11	24	
04	04	11	24	
11		15	20	
15		15	20	
20				21
21				21
24				21

$\mathcal{F}$ – akceptující množiny

- $F_{aU(bUc)}$ — {11,15,20,21,24}
- F_{bUc} — {04,20,21,24}

IV113 Validace a verifikace

Ověřování modelu pro CTL a logiky větvícího se času

Jiří Barnat

Lineární x Větvící se čas

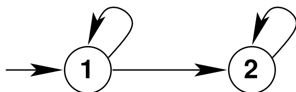
Pnueli, 1977

- Systém lze chápat jako množinu sekvencí stavů – **běhů**.
- Vlastnosti systému lze vymezit vlastnostmi běhů.
- Vlastnosti běhů lze popsat temporální logikou **lineárního času**.

Clarke & Emerson, 1980

- Systém lze chápat jako strom možných pokračování, tzv. **výpočetní strom**. V každém okamžiku chodu systému existuje (konečně mnoho) možných pokračování (budoucích stavů).
- Systém lze vymezit vlastnostmi výpočetního stromu.
- Vlastnosti stromu lze popsat temporální logikou **větvícího se času**.

Systém a výpočetní strom z iniciálního stavu



Logika CTL
(Computation Tree Logic)

CTL neformálně

Možné výpočty

- Je-li dán výpočetní strom a jeden z jeho vrcholů, pak podstrom určený daným vrcholem udává všechny možné běhy, které systém z daného stavu může provést.
- O každém jednom takovém běhu mluvíme jako o možném výpočtu (možné budoucnosti).

CTL formule umožňují

- Specifikovat vlastnosti stavů pomocí atomických propozic.
- Kvantifikovat přes možné výpočty z daného stavu.
- Omezovat množinu možných výpočtů pomocí (kvantifikovaných) LTL operátorů.

Příklad

- $\varphi \equiv EF(a)$
- Je možné provést výpočet, ve kterém jednou bude platit a.

Syntax CTL

Nechť AP je množina atomických propozic. Pak

- Je-li $p \in AP$, pak p je formule.
- Je-li φ formule, pak $\neg\varphi$ je formule.
- Jsou-li φ a ψ formule, pak $\varphi \vee \psi$ je formule.
- Je-li φ formule, pak $EX\,\varphi$ je formule.
- Jsou-li φ a ψ formule, pak $E[\varphi\,U\,\psi]$ je formule.
- Jsou-li φ a ψ formule, pak $A[\varphi\,U\,\psi]$ je formule.

Alternativní zápis (Backus-Naur form)

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\,\varphi \mid E[\varphi\,U\,\varphi] \mid A[\varphi\,U\,\varphi]$$

Syntaktické zkratky

Standardní

- Klasické syntaktické zkratky výrokové logiky
- Syntaktické zkratky z LTL
  - $F\,\varphi \equiv \mathit{true}\; U\,\varphi$
  - $G\,\varphi \equiv \neg F\,\neg\varphi$

Odvozené temporální operátory CTL

- $EF\,\varphi \equiv E[\mathit{true}\; U\,\varphi]$
- $AF\,\varphi \equiv A[\mathit{true}\; U\,\varphi]$
- $EG\,\varphi \equiv \neg AF\,\neg\varphi$
- $AG\,\varphi \equiv \neg EF\,\neg\varphi$
- $AX\,\varphi \equiv \neg EX\,\neg\varphi$

Modely CTL formulí

Model CTL formule

- Je dána množina atomických propozic AP.
- Modelem CTL formule je stav $s \in S$ Kripkeho struktury $M = (S, T, I, s_0)$.

Připomenutí

- Běh v Kripkeho struktuře je maximální cesta začínající v daném (iniciálním) stavu.
- Na konečné běhy nahlížíme jako na nekonečné, které vzniknou opakováním posledního stavu.

Značení

- Nechť $s \in S$ je stav Kripkeho struktury $M = (S, T, I, s_0)$.
- $P_M(s) = \{\pi \mid \pi \text{ je běh začínající ve stavu } s\}$

Předpoklady

- Je dána množina atomických propozic AP.
- Je dán stav $s \in S$ Kripkeho struktury $M = (S, T, I, s_0)$.
- φ, ψ jsou syntakticky správné CTL formule.
- $p \in AP$ je atomická propozice.

Sémantika

$$
\begin{array}{rcl}
s \models p & \text{iff} & p \in I(s) \\
s \models \neg\varphi & \text{iff} & \neg(s \models \varphi) \\
s \models \varphi \vee \psi & \text{iff} & s \models \varphi \text{ or } s \models \psi \\
s \models EX\,\varphi & \text{iff} & \exists \pi \in P_M(s).\pi(1) \models \varphi \\
s \models E[\varphi\, U\, \psi] & \text{iff} & \exists \pi \in P_M(s).(\exists k \geq 0.(\pi(k) \models \psi \text{ and} \\
 & & \forall 0 \leq i < k.\pi(i) \models \varphi)) \\
s \models A[\varphi\, U\, \psi] & \text{iff} & \forall \pi \in P_M(s).(\exists k \geq 0.(\pi(k) \models \psi \text{ and} \\
 & & \forall 0 \leq i < k.\pi(i) \models \varphi))
\end{array}
$$

Příklad

Vyjádřete pomocí CTL formule

- Je možné dosáhnout stav, ve kterém platí a, ale neplatí b.
- Pokud systém obdrží žádost Req, pak v konečném čase vygeneruje potvrzení Ack.
- V každém možném výpočtu nekonečně mnohokrát platí b.
- Vždy je možné systém restartovat (dosáhnout stavu $Restart$).

Model Checking CTL

Definice problému

Model checking CTL

- Je dána Kripkeho struktura $M = (S, T, I, s_0)$.
- Je dána CTL formule φ.
- Problém: Platí, že $M, s_0 \models \varphi$?

Alternativně

- Je dána Kripkeho struktura $M = (S, T, I, s_0)$.
- Je dána CTL formule φ.
- Problém: Spočítat množinu $\{s \mid M, s \models \varphi\}$.

Pojmenování

- Výše uvedené přístupy se někdy také označují jako
  - Local model-checking problém — $M, s_0 \models \varphi$.
  - Global model-checking problém — $\{s \mid M, s \models \varphi\}$.
- Neplést s vlastností algoritmů.
  - Local algorithm for global model-checking.

Algoritmus pro CTL Model-Checking — Idea

Pozorování

- Znám-li pro každý stav platnost formulí φ a ψ, snadno odvodím platnost formulí $\neg\varphi$, $\varphi \vee \psi$, $EX\,\varphi$, $\ldots$.

Idea algoritmu pro CTL Model Checking

- Je dána Kripkeho struktura $M = (S, T, I)$ a formule φ.
- Spočítám značkovací funkci $label : S \rightarrow 2^{2^{\varphi}}$, která o každém stavu $s \in S$ Kripkeho struktury M řekne, jaké podformule formule φ platí v daném stavu.
- Platí, že $s_0 \models \varphi \iff \varphi \in label(s_0)$.
- Funkci *label* budu počítat postupně pro jednotlivé podformule formule φ, a to od nejjednodušších podformulí (atomické propozice) ke složitějším (až po podformuli φ).

Podformule CTL formule

Podformule formule φ

- Je dána CTL formule φ.
- Množinu všech podformulí formule φ označujeme 2^φ.
- Množina 2^φ je definována induktivně dle struktury φ.

Definice 2^φ

1) $\varphi \in 2^\varphi$ (φ je podformule φ)
2) Jestliže $\eta \in 2^\varphi$ a
   - $\eta \equiv \neg\psi$, pak $\psi \in 2^\varphi$ (ψ je podformule φ)
   - $\eta \equiv \psi_1 \vee \psi_2$, pak $\psi_1, \psi_2 \in 2^\varphi$ (ψ_1, ψ_2 jsou podformule φ)
   - $\eta \equiv EX\,\psi$, pak $\psi \in 2^\varphi$ (ψ je podformule φ)
   - $\eta \equiv E[\psi_1\, U\, \psi_2]$, pak $\psi_1, \psi_2 \in 2^\varphi$ (ψ_1, ψ_2 jsou podformule φ)
   - $\eta \equiv A[\psi_1\, U\, \psi_2]$, pak $\psi_1, \psi_2 \in 2^\varphi$ (ψ_1, ψ_2 jsou podformule φ)
3) Žádná jiná formule není podformulí φ.

Transformace formule

Pozorování

- Je snazší prokazovat, platnost existenčně kvantifikovaných modálních operátorů než platnost univerzálně kvantifikovaných modálních operátorů.
- Pro účely verifikace CTL formule φ nad daným Kripkeho systémem M, vyjádříme formuli φ v modifikovaném tvaru.

Alternativní základní syntax CTL

- $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\,\varphi \mid E[\varphi\, U\, \varphi] \mid EG\,\varphi$

Příklad

- Jak se vyjádří $EG\,\varphi$ v původní základní syntaxi CTL?
- Jak se definují podformule CTL formule φ je-li φ zapsána pomocí alternativní syntax?

Algoritmus pro CTL Model-Checking

VSTUP: Kripkeho struktura $M = (S, T, I, s_0)$, CTL formule φ.
VÝSTUP: *True*, pokud $s_0 \models \varphi$, jinak *False*.

```
proc CTLMC(φ, M)
   label := I
   Solved := AP ∩ 2^φ
   while φ ∉ Solved do
         foreach ( η ∈ {¬ψ1, ψ1 ∨ ψ2, EX ψ1, E[ψ1 U ψ2], EG ψ1 | ψ1, ψ2 ∈ Solved})do
            if (η ∈ 2^φ and η ∉ Solved)
               then label := updateLabel(η, label, M)
                    Solved := Solved ∪ {η}
            fi
         od
   od
   return (φ ∈ label(s0))
end
```

Algoritmus pro CTL Model-Checking — updateLabel()

proc updateLabel(η, *label*, *M*)
  if ($\eta \equiv E[\psi_1 \, U \, \psi_2]$)
    then return *checkEU*(ψ_1, ψ_2, *label*, *M*)
  fi
  if ($\eta \equiv EG\, \psi$)
    then return *checkEG*(ψ, *label*, *M*)
  fi
  foreach (s $\in S$)do
    if ($\eta \equiv \neg\psi$ and $\psi \notin label(s)$) or
      ($\eta \equiv \psi_1 \vee \psi_2$ and ($\psi_1 \in label(s) \vee \psi_2 \in label(s)$)) or
      ($\eta \equiv EX\, \psi$ and ($\exists t \in \{t \mid (s,t) \in T\}$ takové, že $\psi \in label(t)$))
      then $label(s) := label(s) \cup \{\eta\}$
    fi
  od
  return *label*
end

Algoritmus pro označení stavů podformulí $E[\psi_1\, U\, \psi_2]$

VSTUP: Kripkeho struktura $M = (S, T, I)$,
Značkovací funkce $label : S \to 2^{\varphi}$, korektní vůči ψ_1 a ψ_2
VÝSTUP: Značkovací funkce $label : S \to 2^{\varphi}$, korektní vůči $E[\psi_1\, U\, \psi_2]$

```
proc checkEU(ψ1, ψ2, label, M)
  Q := {s | ψ2 ∈ label(s)}
  foreach ( s ∈ Q)do
    label(s) := label(s) ∪ {E[ψ1 U ψ2]}
  od
  while (Q ≠ ∅) do
        choose s ∈ Q
        Q := Q ∖ {s}
        foreach ( t ∈ {t | T(t, s)}) do           /* all immediate predecessors */
          if (E[ψ1 U ψ2] ∉ label(t) ∧ ψ1 ∈ label(t))
            then label(t) := label(t) ∪ {E[ψ1 U ψ2]}
                 Q := Q ∪ {t}
          fi
        od
  od
  return label
end
```

Silně souvislé komponenty

Podgraf

- Nechť $G = (V, E)$ je graf, tj. $E \subseteq V \times V$.
- Graf $G' = (V', E')$ nazveme podgrafem grafu G pokud platí $V' \subseteq V$ a $E' = E \cap V' \times V'$.

Podgraf $C = (V', E')$ **grafu** $G = (V, E)$ **se nazývá**

- silně souvislá komponenta, pokud $\forall u, v \in V'$ platí, že $(u, v) \in E'^*$ a $(v, u) \in E'^*$.
- maximální silně souvislá komponenta (SCC), pokud C je silně souvislá komponenta a pro každé $v \in (V \setminus V')$ platí, že $(V' \cup \{v\}, E \cap (V' \cup \{v\} \times V' \cup \{v\}))$ není silně souvislá komponenta.
- netriviální silně souvislá komponenta, pokud C je silně souvislá komponenta a $E' \neq \emptyset$.

Algoritmus pro označení stavů podformulí $EG\,\psi$

VSTUP: Kripkeho struktura $M = (S, T, I, s_0)$,
Značkovací funkce $label : S \rightarrow 2^{\varphi}$, korektní vůči ψ
VÝSTUP: Značkovací funkce $label : S \rightarrow 2^{\varphi}$, korektní vůči $EG\,\psi$

```
proc checkEG(ψ, label, M)
   S' := {s | ψ ∈ label(s)}
   SCC := {C | C je netriviální SCC grafu G' = (S', T ∩ S' × S')}
   Q := ⋃_{C∈SCC} {s | s ∈ C}
   foreach ( s ∈ Q)do
      label(s) := label(s) ∪ {EG ψ}
   od
   while Q ≠ ∅ do
         choose s ∈ Q
         Q := Q ∖ {s}
         foreach ( t ∈ (S' ∩ {t | T(t, s)}))do    /* all immediate predecessors in S' */
            if EG ψ ∉ label(t)
               then label(t) := label(t) ∪ {EG ψ}
                     Q := Q ∪ {t}
            fi
         od
   od
end
```

Složitost algoritmu pro ověřování formule CTL

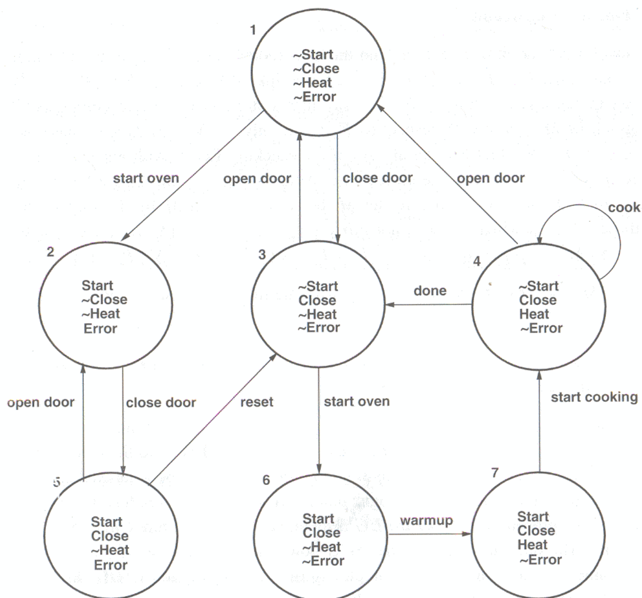
Pozorování

- Každá CTL formule φ má nejvýše $|\varphi|$ různých podformulí.
- Rozklad každého podgrafu grafu $G = (S, T)$ na SCC lze provést v čase $\mathcal{O}(|S| + |T|)$.
- Každé volání funkce *updateLabel* skončí v čase $\mathcal{O}(|S| + |T|)$.

Celková složitost

- Algoritmus *CTLMC* má časovou složitost $O(|\varphi|(|S| + |T|))$.
- Algoritmus *CTLMC* má prostorovou složitost $O(|\varphi||S|)$.

Příklad: Mikrovlnná trouba $AG(Start \implies AF(Heat))$



Příklad: Mikrovlnná trouba $AG(Start \implies AF(Heat))$

Přepis formule $\varphi \equiv AG(Start \implies AF(Heat))$

- $AG(Start \implies AF(Heat))$
- $AG(\neg(Start \wedge \neg AF(Heat)))$
- $AG(\neg(Start \wedge EG(\neg Heat)))$
- $\neg EF(Start \wedge EG(\neg Heat))$
- $\neg E[true\, U\, (Start \wedge EG(\neg Heat))]$

Platnost podformulí $[S(\varphi) = \{s \mid s \models \varphi\}]$

- $S(Start) = \{2, 5, 6, 7\}$
- $S(Heat) = \{4, 7\}$
- $S(\neg Heat) = \{1, 2, 3, 5, 6\}$
- $S(EG(\neg Heat)) = \{1, 2, 3, 5\}$
- $S(Start \wedge EG(\neg Heat)) = \{2, 5\}$
- $S(E[true\, U\, (Start \wedge EG(\neg Heat))]) = \{1, 2, 3, 4, 5, 6, 7\}$
- $S(\neg E[true\, U\, (Start \wedge EG(\neg Heat))]) = \emptyset$

Logika CTL*

CTL* jako rozšíření CTL

Pozorování

- V logice CTL není možné omezit množinu možných výpočtů libovolnou *LTL* formulí. Tj. každý modální operátor LTL musí být bezprostředně předcházen kvantifikátorem.

Logika CTL*

- Logika větvícího se času stejně jako logika CTL.
- Množiny možných běhů lze omezit libovolnou *LTL* formulí.
- V syntax logiky CTL* vystupují kvantifikátory cest jako samostatné operátory.

Příklad

- $A[p \wedge X(\neg p)]$ je formule CTL*, ale není to formule CTL.

Typy CTL* formulí

- Operátory E a A jsou samostatné, proto existují v CTL* formule jejichž modelem je běh Kripkeho struktury.
- Aplikací operátorů E a A vznikají z formulí jejichž modelem je běh Kripkeho struktury, formule, jejichž modelem je stav Kripkeho struktury.
- Rozlišujeme tedy formule stavu a formule cesty.

Syntax CTL*

formule stavu $\quad \varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid E\,\psi$

formule cesty $\quad \psi ::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid X\,\psi \mid \psi\,U\,\psi$

Sémantika CTL*

Předpoklady

- Je dána množina atomických propozic AP, $p \in AP$.
- Je dána Kripkeho struktura $M = (S, T, I)$.
- φ_1, φ_2 jsou CTL* formule stavu, ψ_1, ψ_2 formule cesty.

Sémantika

$$
\begin{array}{rcl}
M, s \models p & \text{iff} & p \in I(s) \\
M, s \models \neg\varphi_1 & \text{iff} & \neg(M, s \models \varphi_1) \\
M, s \models \varphi_1 \vee \varphi_2 & \text{iff} & M, s \models \varphi_1 \text{ or } M, s \models \varphi_2 \\
M, s \models E\,\psi_1 & \text{iff} & \exists \pi \in P_M(s). \pi \models \psi_1 \\
M, \pi \models \varphi_1 & \text{iff} & M, \pi(0) \models \varphi_1 \\
M, \pi \models \neg\psi_1 & \text{iff} & \neg(M, \pi \models \psi_1) \\
M, \pi \models \psi_1 \vee \psi_2 & \text{iff} & M, \pi \models \psi_1 \text{ or } M, \pi \models \psi_2 \\
M, \pi \models X\,\psi_1 & \text{iff} & M, \pi^1 \models \psi_1 \\
M, \pi \models \psi_1\, U\, \psi_2 & \text{iff} & \exists k \geq 0. (M, \pi^k \models \psi_2 \text{ and} \\
 & & \forall 0 \leq i < k. M, \pi^i \models \psi_1)
\end{array}
$$

Porovnání logik LTL, CTL a CTL*

Unifikace modelů

Pozorování

- Každá LTL formule je CTL* formule cesty.
- Každá CTL formule je CTL* formule stavu.
- Modelem CTL* formule cesty je běh Kripkeho struktury.
- Modelem CTL* formule stavu je stav Kripkeho struktury.
- Nevhodné pro účely porovnání.

Unifikace modelu

- Za účelem unifikace modelů definujeme, kdy CTL* formule cesty platí ve stavu Kripkeho struktury.
- Nechť ψ je CTL* formule cesty, pak

$$M, s \models \psi \quad \text{iff} \quad M, s \models A\psi$$

Motivace

Cíl

- Chceme zjistit, zda jsou vlastnosti (formule), které lze vyjádřit v jedné logice a nelze vyjádřit v jiné logice.
- Chceme zjistit, ve které logice lze vyjádřit víc vlastností.
- Chceme identifikovat vlastnosti, které nelze vyjádřit v jiné logice, tj. jak vypadá formule logiky $\mathcal{L}_1$, pro kterou neexistuje ekvivalentní formule logiky $\mathcal{L}_2$.

Ekvivalence formulí (i různých logik)

- Formule φ a ψ jsou ekvivalentní, právě když pro všechny modely $M = (S, T, I, s_0)$ a stavy $s \in S$ platí

$$M, s \models \varphi \quad \text{iff} \quad M, s \models \psi$$

Shodná expresibilita

- Temporální logiky $\mathcal{L}_1$ a $\mathcal{L}_2$ jsou shodně expresibilní (mají stejnou vyjadřovací sílu), pokud pro všechny modely $M = (S, T, I, s_0)$ a stavy $s \in S$ platí

$$\forall \varphi \in \mathcal{L}_1.(\exists \psi \in \mathcal{L}_2.(M, s \models \varphi \iff M, s \models \psi)) \quad (1)$$
$$\wedge\ \forall \psi \in \mathcal{L}_2.(\exists \varphi \in \mathcal{L}_1.(M, s \models \varphi \iff M, s \models \psi)). \quad (2)$$

Menší expresibilita

- Pokud platí pouze tvrzení (1), tj. neplatí tvrzení (2), pak je logika $\mathcal{L}_1$ méně expresibilní (má menší vyjadřovací sílu) než logika $\mathcal{L}_2$.

Tvrzení 1

- LTL a CTL jsou vyjadřovací silou neporovnatelné.
  1) $AG(EF(q))$ je CTL formule, kterou nelze vyjádřit v LTL.
  2) $FG(q)$ je LTL formule, kterou nelze vyjádřit v CTL.

Příklad – důkaz 1)

- Najděte dvě různé Kripkeho struktury a v nich identifikujte stavy, které jsou rozlišitelné CTL formulí $AG(EF(q))$ a přitom nejsou rozlišitelné žádnou LTL formulí (generují shodnou množinu běhů).

Příklad – intuice za 2) [důkaz jde nad rámec tohoto kurzu]

- Ukažte, že CTL formule $AF(AG(q))$ není ekvivalentní LTL formuli $FG(q)$.

Porovnání LTL, CTL, a CTL*

Důsledek 1

- CTL* je striktně více expresibilní než LTL.
  - Každá LTL formule je i formule CTL*.
  - CTL* formule $AG(EFq)$ není vyjádřitelná v LTL.

Důsledek 2

- CTL* je striktně více expresibilní než CTL.
  - Každá CTL formule je i formule CTL*.
  - CTL* formule $FG(q)$ není vyjádřitelná v CTL.

Pozorování

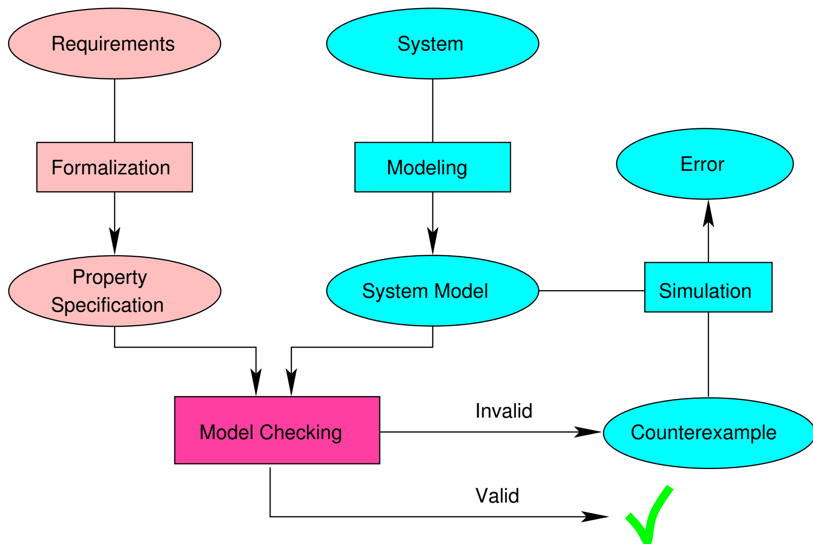
- Existují vlastnosti vyjádřitelné jak v LTL tak i v CTL.
  - CTL formule $A[p\, U\, q]$ je ekvivalentní LTL formuli $p\, U\, q$.

IV113 Validace a verifikace

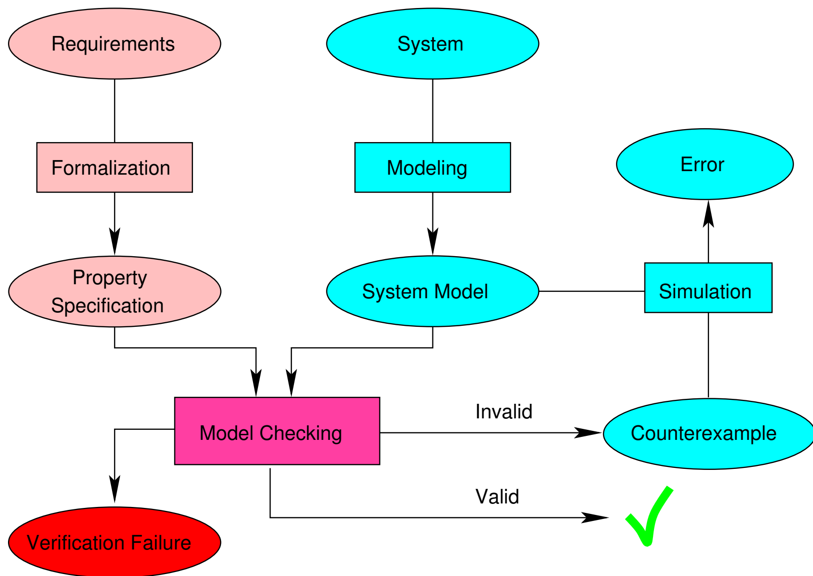
Symbolický přístup k metodě ověřování modelu

Jiří Barnat

Problém stavové exploze při ověřování modelu



Problém stavové exploze při ověřování modelu



Motivace

Pozorování

- Stav modelovaného systému je určen valuací proměnných.
- Každá proměnná má konečnou doménu, její hodnotu lze uložit s využitím pevně stanoveného počtu bitů.
- Stav je v paměti počítače reprezentován jako bitový vektor $(a_1, \ldots, a_n)$ fixní délky n.

Množiny stavů

- Algoritmy pro verifikaci uchovávají množiny stavů.
- Množinu stavů lze chápat jako množinu binárních vektorů.
- Množinu binárních vektorů lze popsat **boolovskou funkcí**.

Boolovské funkce

- Jsou formule výrokové logiky, nad konečnou množinou proměnných typu Bool.

Příklad

- Stav systému je dán valuací 4 proměnných (a1,b1,a2,b2) každé do 2 hodnotové domény {0,1}.
- Stav systému je chybový, pokud se shodují proměnné a1 a b1 a proměnné a2 a b2.
- Popište množinu chybových stavů boolovskou funkcí.

Některá možná řešení

Boolovské funkce

- Jsou formule výrokové logiky, nad konečnou množinou proměnných typu Bool.

Příklad

- Stav systému je dán valuací 4 proměnných (a1,b1,a2,b2) každé do 2 hodnotové domény {0,1}.
- Stav systému je chybový, pokud se shodují proměnné a1 a b1 a proměnné a2 a b2.
- Popište množinu chybových stavů boolovskou funkcí.

Některá možná řešení

- $(a1 \wedge b1 \wedge a2 \wedge b2) \vee (a1 \wedge b1 \wedge \neg a2 \wedge \neg b2) \vee (\neg a1 \wedge \neg b1 \wedge \neg a2 \wedge \neg b2) \vee (\neg a1 \wedge \neg b1 \wedge a2 \wedge b2)$
- $a1 \Leftrightarrow b1 \wedge a2 \Leftrightarrow b2$

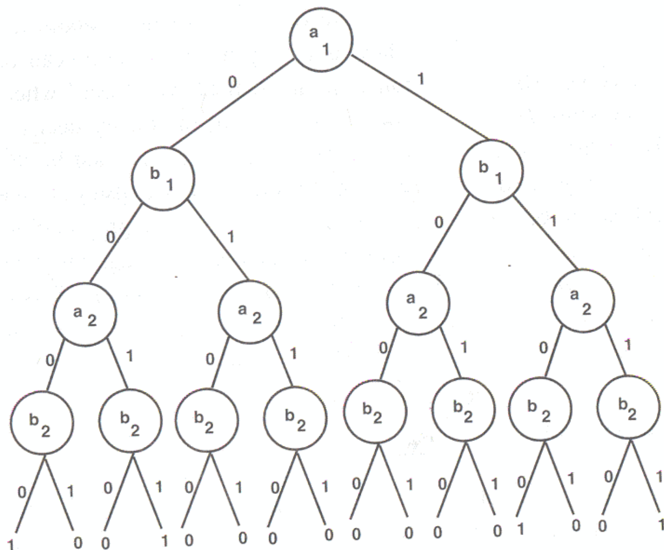
Binární rozhodovací stromy (BDTs)

- Orientovaný strom s právě jedním kořenem.
- Každý vnitřní vrchol je označen binární proměnnou (v) a má právě 2 následníky ($low(v)$, $high(v)$).
- Každý list je označen binární hodnotou (0,1).

Kódování boolovské funkce pomocí BDT

- Každá kombinace hodnot vstupních proměnných odpovídá právě jedné cestě z kořene stromu do listu.
- Hodnoty uložené v jednotlivých listech udávají hodnoty funkce pro jednotlivé vstupy.

Binární rozhodovací strom $\psi = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$



Nevýhoda BDTs

- BDTs jsou zbytečně prostorově náročné (obsahují redundantní informace).

Příklad

- Identifikujte isomorfní podstromy BDT z předchozího příkladu.

Binární rozhodovací diagram (BDD)

- Acyklický graf, jehož vrcholy mají výstupní stupeň vždy buď 0 (listy) nebo 2 (vnitřní vrcholy).
- Vrcholy BDD mají stejné atributy jako vrcholy BDT.

Vytvoření (minimálního) BDD

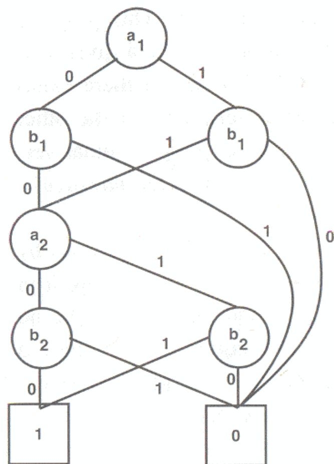
Inicializace

- Pro danou boolovskou funkci uvaž nějaké BDD (BDT).
- Eliminace nedosažitelných vrcholů
  - Odstraň vrcholy nedosažitelné z kořene BDD.
- Eliminace duplikátních listů
  1) Odstraň až na jeden stejně označené listy BDT.
  2) Všechny hrany eliminovaných stejně označených listů přepoj do zbývajícího stejně označeného listu.

Opakovaně aplikuj následující transformace

- Eliminace duplikátních (vnitřních) vrcholů
  - Pokud v BDD existují stejně označené vnitřní vrcholy u, v takové, že $low(v) = low(u)$ a $high(v) = high(u)$, tak odstraň vrchol u a hrany původně vedoucí do vrcholu u přepoj do vrcholu v.
- Eliminace zbytečných testů
  - Eliminuj vnitřní vrchol v, pokud $low(v) = high(v)$ a hrany původně vedoucí do v přepoj do vrcholu $low(v)$.

BDD pro $\psi = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$



Kódování boolovské funkce pomocí BDD

Tvrzení

- Každý vrchol v binárního rozhodovacího diagramu kóduje nějakou boolovskou funkci $F_v(x_1, \ldots, x_n)$.

Výpočet $F_v(x_1, \ldots, x_n)$ pro konkrétní hodnoty $h_1, \ldots, h_n$.

- Je-li vrchol v list, pak
  - $F_v(h_1, \ldots, h_n) = 1$, pokud je list v označen hodnotou 1.
  - $F_v(h_1, \ldots, h_n) = 0$, pokud je list v označen hodnotou 0.
- Je-li vrchol v vnitřní vrchol označený proměnou x_i, pak
  - $F_v(h_1, \ldots, h_n) = F_{low(v)}(h_1, \ldots, h_n)$, pokud $h_i == 0$.
  - $F_v(h_1, \ldots, h_n) = F_{high(v)}(h_1, \ldots, h_n)$, pokud $h_i == 1$.

Uspořádání proměnných v BDD — OBDD

Pozorování

- Každý mezivýsledek výpočtu minimálního BDD je BDD.
- Výpočet minimálního BDD lze začít v libovolném BDD.
- Pro danou boolovskou funkci existují různá BDD.

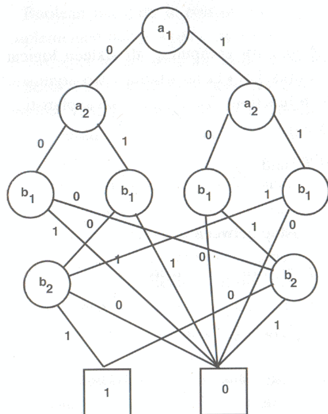
Kanonická forma BDD

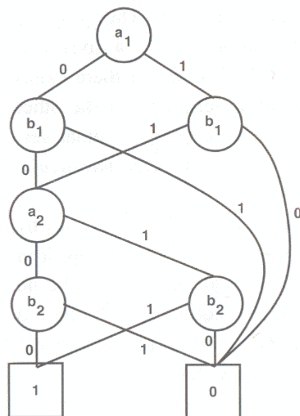
- Minimální BDD vzniklé z BDT s předem daným pořadím proměnných je určeno jednoznačně.
- BDD s fixovaným pořadím proměnných se označují jako Ordered BDD (OBDD).

Kanonizace

- Kanonizaci BDD, které respektuje pořadí proměnných lze provést v čase $\mathcal{O}(n)$, kde n je velikost původního BDD.
- [Transformace se aplikují nejprve na vrcholy, které jsou označeny větší proměnnou.]

Různá OBDD dle různého uspořádání proměnných





Operace restrikce na OBDD

Pozorování

- Každé OBDD reprezentuje nějakou boolovskou funkci.
- Boolovské funkce lze skládat pomocí unárních a binárních logických operátorů ($\neg, \wedge, \vee, \implies, XOR, \ldots$).
- Skládání lze přenést na OBDD.

Aplikace operací na OBDD – funkce *Apply*

- Máme OBDD O a O', která odpovídají funkcím f a f'.
- Chceme proceduru $Apply(O, O', \star)$, která vypočítá OBDD odpovídající složení funkcí f a f' logickým operátorem $\star$.

Operace restrikce

- $F_{x_i \leftarrow b}(x_1, \ldots, x_n) = F(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$
- Vytvoří boolovskou funkci s menším počtem proměnných.

Realizace na OBDD

- Pokud je kořen r označen proměnnou x_i, novým kořenem se stává vrchol
  - $low(r)$ pokud $b = 0$
  - $high(r)$ pokud $b = 1$
- Pokud vrchol v má hranu vedoucí do vrcholu t označeným proměnou x_i, tak se tato hrana přepojí do:
  - $low(t)$ pokud $b = 0$
  - $high(t)$ pokud $b = 1$
- OBDD se minimalizuje (vrcholy x_i jsou nedosažitelné).

Shannonova expanze

- Pro aplikaci libovolného binární logického operátoru lze využít tzv. Shannonovu expanzi:

$$F = (\neg x \wedge F_{x \leftarrow 0}) \vee (x \wedge F_{x \leftarrow 1})$$

- Pokud $F = f \star f'$, kde $\star$ je binární logická operace, pak

$$f \star f' = (\neg x \wedge (f_{x \leftarrow 0} \star f'_{x \leftarrow 0})) \vee (x \wedge (f_{x \leftarrow 1} \star f'_{x \leftarrow 1}))$$

Ostatní operace na OBDD

$Apply(O, O', \star)$

- Nechť v, v' jsou kořenové uzly O, O', označené proměnnými x, x'.
- Pokud v a v' jsou listy označené hodnotami h a h', pak vrať list označený hodnotou $h \star h'$.
- Jinak, pokud

$x = x'$ pak vrať nový vrchol w označený proměnnou x, kde

- $low(w) = Apply(low(v), low(v'), \star)$
- $high(w) = Apply(high(v), high(v'), \star)$

$x < x'$ pak vrať nový vrchol w označený proměnnou x, kde

- $low(w) = Apply(low(v), O', \star)$
- $high(w) = Apply(high(v), O', \star)$

$x' < x$ pak vrať nový vrchol w označený proměnnou x', kde

- $low(w) = Apply(O, low(v), \star)$
- $high(w) = Apply(O, high(v), \star)$

Operace negace a test na prázdnost

Pozorování

- Pokud *OBDD X* realizuje funkci F_X, tak *OBDD Y* realizující funkci $\neg F_X$ se vytvoří kopií *OBDD X* a přeznačením listů kopie duální hodnotou.

Test na prázdnost

- OBDD mají kanonický tvar.
- Kanonické OBDD reprezentující prázdnou množinu je vždy tvořeno pouze listem označeným hodnotou 0.

Test na přítomnost stavu v množině

- Vytvořím OBDD realizující jednoprvkovou množinu s dotazovaným stavem.
- Aplikuji operaci $\wedge$ na dané OBDD.
- Provedu test na prázdnost.

Symbolická reprezentace Kripkeho struktury

Kódování přechodové funkce KS pomocí OBDD

Pozorování

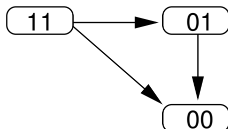
- Stav Kripkeho struktury $M = (S, T, I)$ je popsán n binárními proměnnými $a_1, \ldots, a_n$.
- Každou podmnožinu stavů Kripkeho struktury je možné zachytit pomocí OBDD s n proměnnými.
- Podobně přechodovou relaci $T \subseteq S \times S$ je možné kódovat boolovskou funkcí s $2n$ proměnnými.

Zjednodušení reprezentace OBDD

- Hrany vedoucí do listu 0 je možné vynechat.
- Neexistence hrany indikuje přechod do listu 0.

Příklad

- $M = (\{00, 01, 11\}\}, \{(11, 00), (11, 01), (01, 00)\}, I)$



- T lze popsat pomocí $F(a, b, a', b')$
- $F(a, b, a', b') =$
  $(a \wedge b \wedge \neg a' \wedge b') \vee (a \wedge b \wedge \neg a' \wedge \neg b') \vee (\neg a \wedge b \wedge \neg a' \wedge \neg b')$
- Zakreslete OBDD pro F je-li dáno $a < b < a' < b'$.

Pozorování

- Mějme $M = (S, T, I)$ a $OBDD_T(a, b, a', b')$.
- Mějme množinu stavů X zadanou pomocí $OBDD_X(a, b)$.
- Z $OBDD_T$ a $OBDD_X$ lze vytvořit $OBDD_{X'}(a', b')$, které reprezentuje množinu následníků množiny X, tj.

$$X' = \{v \in S \mid u \in X \wedge (u, v) \in T\}.$$

Následníci množiny stavů

Výpočet $OBDD_{X'}$ (intuitivně)

- $OBDD_{X'} = Apply(OBDD_T, OBDD_X, \wedge)$
- Modifikuj $OBDD'_X$ tak, aby na každé cestě byl vrchol a'.
- V $OBDD_{X'}$ smaž všechny vrcholy a a b.
- Postupně vol všechny vrcholy označené a' jako kořeny a vypočti k nim minimální OBDD.
- Množinu vypočtených OBDD spoj pomocí operace $\vee$.
- Výsledné OBDD minimalizuj.
- Přejmenuj čárkované proměnné na nečárkované.

Příklad

- Spočítejte reprezentaci následníků množiny $\{00, 11\}$.

Předchůdci množiny stavů

Podobně lze počítat předchůdce dané množiny (intuitivně).

- Přeznač proměnné v $OBDD_X$ na čárkované.
- $OBDD_{X'} = Apply(OBDD_T, OBDD_X, \wedge)$
- Modifikuj $OBDD'_X$ tak, aby na každé cestě byl vrchol a'.
- Vrcholy a', ze kterých neexistuje cesta do listu ohodnoceným 1 nahraď novým listem ohodnoceným 0.
- Ostatní vrcholy a' nahraď listem ohodnoceným 1.
- Smaž staré listy a ostatní vrcholy označené čárkovanou proměnnou a minimalizuj OBDD.

Příklad

- Spočítejte OBDD reprezentující množinu stavů $\{00\}$.
- Spočítejte OBDD reprezentující předchůdce dané množiny.

Symbolický přístup k verifikaci CTL

Připomenutí

Pozorování

- Znám-li pro každý stav platnost formulí φ a ψ, snadno odvodím platnost formulí $\neg\varphi$, $\varphi \vee \psi$, $EX\,\varphi$, ...

Idea algoritmu pro CTL Model Checking

- Je dána Kripkeho struktura $M = (S, T, I)$ a formule φ.
- Spočítám značkovací funkci $label : S \to 2^{\varphi}$, která o každém stavu $s \in S$ Kripkeho struktury M řekne jaké podformule formule φ platí v daném stavu.
- Platí, že $s_0 \models \varphi \iff \varphi \in label(s_0)$.
- Funkci *label* budu počítat postupně pro jednotlivé podformule formule φ, a to od nejjednodušších podformulí (atomické propozice) ke složitějším (až po podformuli φ).

Symbolický přístup

Myšlenka

- Budu si kompaktně pamatovat/budovat množiny stavů, ve kterých platí jednotlivé podformule verifikované CTL formule.
- Explicitní počítání funkce *label* nahradím manipulací s těmito kompaktními reprezentacemi.

Realizace

- Množiny stavů udržovány pomocí OBDD struktur.
- Výchozím bodem jsou OBDD pro jednotlivé AP.
- Podle struktury formule počítám OBDD pro jednotlivé podformule.
- Otestuji přítomnost iniciálního stavu v množině stavů splňující verifikovanou formuli.

Připomenutí syntax CTL

- $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\,\varphi \mid E[\varphi\, U\, \varphi] \mid EG\,\varphi$

Výpočet množiny stavů splňující CTL formuli

- Značení
  - $F(\psi)$ označme (funkci popisující) množinu stavů splňující ψ.
  - $Succ(X)$ označme všechny následníky množiny stavů X.
  - $Pred(X)$ označme všechny předchůdce množiny stavů X.
- Boolovské funkce pro atomické propozice
  - Atomická propozice se vyjadřuje o platnosti proměnných.
  - Atomická propozice je boolovská funkce.
- Výpočet logických operátorů ($\neg$, $\vee$)
  - $F(\neg\psi_1) = \neg(F\psi_1)$
  - $F(\varphi \vee \psi) = F(\varphi) \vee F(\psi)$

Temporální operátory $EX(\varphi)$, $E[\varphi\, U\, \psi]$ a $EG(\varphi)$

- Množina splňující $EX(\varphi)$
  - $F(EX(\varphi)) = Pred(F(\varphi))$
- Množina splňující $E(\varphi\, U\, \psi)$
  - $F(E(\varphi\, U\, \psi)) = X$,
    kde X je nejmenší pevný bod rekurzivního předpisu

$$X = F(\psi) \cup (F(\varphi) \cap EX(X))$$

- Množina splňující $EG\,(\varphi)$
  - $F(EG\,\varphi) = X$,
    kde X je největší pevný bod rekurzivního předpisu

$$X = F(\varphi) \cap EX(X)$$

Výpočet pevných bodů

Nejmenší pevný bod $f(x)$

```
proc LFP(f)
  X = ∅
  Xold = ∅
  do
     Xold = X
     X := f(X)
  while (X ≠ Xold)
end
```

Největší pevný bod $f(x)$

```
proc GFP(f)
  X = S
  Xold = S
  do
     Xold = X
     X := f(X)
  while (X ≠ Xold)
end
```

Ověřování modelu – shrnutí

Ověřování modelu – shrnutí

Enumerativní × symbolický přístup

- Enumerativní – orientován na "control-flow"
- Symbolický – orientován na "data-flow"

Výhody oproti testování

- Není třeba zdrojový kód (aplikovatelné ve fázi návrhu).
- Aplikovatelné na paralelní programy.

Výhody oproti statickým metodám

- Metoda je úplná, tj. vždy dává přesné výsledky.
- Možno verifikovat temporální vlastnosti.

Nevýhoda

- Problém stavové exploze.

IV113 Validace a verifikace

Ověřování modelu s využitím řešičů SAT a SMT (Bounded Model Checking)

Jiří Barnat

Připomenutí – SAT a SMT

Problém splnitelnosti – SAT

- Nalezení valuace boolovských proměnných formule výrokové logiky takové, že formule je v této valuaci pravdivá.

Satisfiability Modulo Theory – SMT

- Problém rozhodnout splnitelnost formule prvořádové logiky s rovností, predikáty a funkčními symboly kódující jednu či více zvolených teorií.

Typické teorie SMT

- Aritmetika neomezených celých a desetinných čísel.
- Aritmetika celých čísel omezené velikosti (bitové vektory).
- Teorie datových struktur (seznamy, pole, ...).

Připomenutí – Řešiče SAT a SMT

ZZZ aka **Z3**

- Nástroj vyvíjený v Microsoft Research.
- WWW interface — `http://www.rise4fun.com/Z3`
- Binární API pro použití v jiných aplikacích.

SMT-LIB

- Standardizace jazyka pro zadávání SMT dotazů.
- Volně dostupná knihovna s implementací SMT.

Připomenutí – Splnitelnost a platnost

Pozorování

- Formule je platná právě když její negace není splnitelná.

Důsledek

- Řešiče SAT a SMT lze využít jako nástroje pro dokazování platnosti formulovaných tvrzení.

Syntéza modelu

- Řešiče SAT nejen rozhodují splnitelnost formulí, ale v případě splnitelnosti vrací požadovanou valuaci proměnných, pro níž je formule pravdivá.
- Na rozdíl od dokazovacích nástrojů tak poskytují "protipříklad" v případě neplatnosti dokazovaného tvrzení.

Ověřování safety vlastností redukcí na problém SAT

Bounded Model Checking (BMC)

Hypotéza

- Je-li v systému chyba, pro její reprodukci stačí malý počet kontrolovaných kroků systému.

Myšlenka metody

- Používáme-li metodu ověřování modelu pro detekci chyb, je smysluplné zkoumat, zda k porušení specifikace dojde během prvních k kroků systému.

Literatura

- Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu: Symbolic Model Checking without BDDs. TACAS 1999: 193-207, LNCS 1579.
- Henry A. Kautz, Bart Selman: Planning as Satisfiability.Proceedings of the 10th European conference on Artificial intelligence (ECAI'92): 359-363, 1992, Kluwer.

Redukce problému BMC na problém SAT

Předpoklady

- Množinu prefixů délky k všech běhů Kripkeho struktury M lze kódovat boolovskou formulí $[M]^k$.
- Porušení vlastnosti typu *safety*, které se projeví po provedení k kroků systému, lze kódovat formulí $[\neg\varphi]^k$.

Redukce na problém SAT

- Ověřuje se splnitelnost formule $[M]^k \wedge [\neg\varphi]^k$.
- Splnitelnost indikuje existenci protipříkladu délky k.
- Nesplnitelnost formule prokazuje neexistenci protipříkladu délky k.

Předpoklady

- Mějme Kripkeho strukturu $M = (S, T, I)$ s iniciálním stavem $s_0 \in S$.
- Libovolný stav $s \in S$ lze reprezentovat jako bitový vektor délky n, tj. stav $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$.

Kódování M skrze boolovské formule

- $Init(s)$ – formule, která je splnitelná právě pro takovou valuaci proměnných $a_1, a_2, ..., a_n$, které popisují stav s_0.
- $Trans(s, s')$ – formule, která je splnitelná pro stavové vektory s, s', právě tehdy když valuace proměnných $a_1, a_2, ..., a_n, a'_1, a'_2, ..., a'_n$ popisuje stavy, mezi kterými existuje přechod, tj. $(s, s') \in T$.

Kódování konečných běhů M

Popis běhů systému délky k

- Běh délky k je tvořen $k+1$ stavy $s_0, s_1, \dots, s_k$.
- Množina všech běhů délky k struktury M je označena jako $[M]^k$ a je popsána následující formulí:

$$[M]^k \equiv Init(s_0) \wedge \bigwedge_{i=1}^{k} Trans(s_{i-1}, s_i)$$

Příklad $[M]^3 \wedge [\neg\varphi]^3$

- $Init(s_0) \wedge Trans(s_0, s_1) \wedge Trans(s_1, s_2) \wedge Trans(s_2, s_3) \wedge \neg\varphi(s_3)$

Úplnost metody BMC

Úplnost BMC pro detekci porušení safety vlastností

Problém – nedetekované porušení vlastnosti typu safety

- Porušení invariantu je cestou délky k nedosažitelné.
- Cesty kratší než k nejsou v $[M]^k$ kódovány.

Ohraničení k shora

- Pokud $k \geq d$, kde d je průměr grafu, všechny místa možného porušení invariantu jsou pokryta.
- Průměr grafu lze omezit konstantou 2^n, kde n je počet bitů stavového vektoru.

Řešení problému

- Realizace procedury BMC postupně pro $k \in [0, d]$.

Automatická detekce průměru grafu

Fakta

- Určení konstanty d uživatelem je nereálné.
- Bezpečné horní odhady jsou velmi vzdálené realitě.
- Chtěli bychom, aby samotná procedura verifikace detekovala, zda má smysl nadále zvyšovat k.

Kostra algoritmu pro úplný BMC

$k = 0$
while `(true)` **do**
  if (existuje protipříklad délky k)
    then return `"Invalid"`
  if (neexistuje protipříklad délky větší než k)
    then return `"Valid"`
  $k = k + 1$
od

Značení I

Předpoklady

- Kripkeho struktura $M = (S, T, I)$.
- Stavy popsány bitovými vektory fixní délky.
- *Trans* je SAT reprezentace binární relace T.

Cesta délky n

$$path(s_{[0..n]}) \quad \equiv \quad \bigwedge_{0 \leq i < n} Trans(s_i, s_{i+1})$$

Platnost tvrzení Q podél celé cesty

$$all.Q(s_{[0..n]})$$

Značení II

Cesta bez cyklů

$$loopFree(s_{[0..n]}) \quad \equiv \quad path(s_{[0..n]}) \wedge \bigwedge_{0 \leq i < j \leq n} s_i \neq s_j$$

Existence cesty délky n z s_0 do s_n

$$path_n(s_0, s_n) \quad \equiv \quad \exists s_1 \ldots s_{n-1}.path(s_{[0..n]})$$

Nejkratší cesta

$$shortest(s_{[0..n]}) \quad \equiv \quad path(s_{[0..n]}) \wedge \neg\Big(\bigvee_{0 \leq i < n} path_i(s_0, s_n)\Big)$$

Ekvivalentní formulace problému

Verifikace

- Chceme ukázat, že není dosažitelný stav z iniciální konfigurace, který by porušoval specifikaci φ, tedy chceme ukázat, že

$$\forall i. \forall s_0 \ldots s_i. \Big(Init(s_0) \wedge path(s_{[0..i]}) \implies \varphi(s_i) \Big)$$

Alternativně

- Chceme ukázat, že z chybového stavu není směrem zpět dosažitelný iniciální stav

$$\forall i. \forall s_0 \ldots s_i. \Big(\neg Init(s_0) \impliedby path(s_{[0..i]}) \wedge \neg \varphi(s_i) \Big)$$

Ekvivalentně

$$\forall i. \forall s_0 \ldots s_i. \neg \Big(Init(s_0) \wedge path(s_{[0..i]}) \wedge \neg \varphi(s_i) \Big)$$

Terminace procedury BMC – Acyklické cesty

Podmínka terminace v kostře algoritmu pro BMC

- Není delší acyklická cesta z počátečního stavu. tj. následující formule je nesplnitelná:

$$Init(s_0) \;\wedge\; loopFree(s_{[0..i+1]})$$

- **Platí i symetricky pro zpětnou dosažitelnost z chybových stavů.**

Řešení 1

- $\texttt{not SAT}\Big(\; loopFree(s_{[0..i+1]}) \;\wedge\; Init(s_0) \;\Big)$
  $\vee$
  $\texttt{not SAT}\Big(\; loopFree(s_{[0..i+1]}) \;\wedge\; \neg\varphi(s_{i+1}) \;\Big)$

Terminace procedury BMC – Acyklické cesty II

Vyšší účinost terminačního kritéria

- Při zpětné dosažitelnosti z $\neg\varphi$ stavů není třeba uvažovat cesty, které jdou přes další $\neg\varphi$ stavy.
- Symetricky platí i pro dopřednou variantu pro systémy, ve kterých je definováno více iniciálních stavů, tj. pro detekci úplnosti není třeba uvažovat cesty, které procházejí dalšími iniciálními stavy.

Řešení 2

- $\texttt{not SAT}\Big(\ \mathit{loopFree}(s_{[0..i+1]}) \wedge \mathit{Init}(s_0) \wedge \mathit{all}.\neg\mathit{Init}(s_{[1..i+1]})\ \Big)$
  $\vee$
  $\texttt{not SAT}\Big(\ \mathit{loopFree}(s_{[0..i+1]})\ \wedge\ \neg\varphi(s_{i+1})\ \wedge\ \mathit{all}.\varphi(s_{[0..i]})\ \Big)$

BMC, které nezačíná pro $k = 0$

Pozorování

- Pro malé hodnoty k, dotazy na řešiče SAT nevedou k nalezení protipříkladu ani k ukončení výpočtu.
- Chceme proceduru BMC začít s k větším než 0.

Reformulace testu na protipříklad

- Původní test na existenci protipříkladu pro dané k

$$\mathtt{SAT}\Big(Init(s_0) \wedge path(s_{[0..k]}) \wedge \neg\varphi(s_k)\Big)$$

  je nutno reformulovat, abychom neminuli protipříklady, jež jsou kratší než výchozí hodnota k.
- Nový test na přítomnost protipříkladu:

$$\mathtt{SAT}\Big(Init(s_0) \wedge path(s_{[0..k]}) \wedge \neg all.\varphi(s_{[0..k]})\Big)$$

k-indukce v BMC

Pozorování

- Testy lze reformulovat tak, aby připomínaly strukturu matematické indukce.
- TAUT je test na tautologii (nesplnitelnost negace).

Báze

- Test na přítomnost protipříkladu.

$$\mathtt{SAT}\Big(\neg\big(\ \mathit{Init}(s_o) \wedge \mathit{path}(s_{[0..i]}) \implies \mathit{all}.\varphi(s_{[0..i]})\ \big)\Big)$$

Indukční krok

- Test na úplnost.

$$\mathtt{TAUT}\Big(\neg \mathit{Init}(s_0) \Longleftarrow \mathit{all}.\neg \mathit{Init}(s_{[1..(i+1)]}) \wedge \mathit{loopFree}(s_{[0..i+1]})\Big)$$
$$\vee$$
$$\mathtt{TAUT}\Big(\ \mathit{loopFree}(s_{[0..i+1]}) \wedge \mathit{all}.\varphi(s_{[0..i]}) \implies \varphi(s_{i+1})\ \Big)$$

Acyklické versus nejkratší cesty v BMC

Pozorování

- Průměr grafu (d) je délka nejdelší z nejkratších cest mezi každými dvěma vrcholy grafu.
- Acyklická cesta v grafu může být výrazně delší než je průměr grafu.

BMC s nejkratšími cestami

- Algoritmus BMC je korektní, pokud se místo *loopFree* použije *shortest*.
- Predikát *shortest* ale vyžaduje použití kvantifikátorů, nejedná se tedy o čistou aplikaci SAT.

Pro více detailů viz ...

- Mary Sheeran, Satnam Singh, and Gunnar Stålmarck: Checking Safety Properties Using Induction and a SAT-Solver, FMCAD 2000, 108-125, LNCS 1954, Springer.

Ověřování modelu LTL metodou BMC

Pozorování 1

- LTL je dobře definováno pouze pro nekonečné běhy.
- Pro vyhodnocování LTL na konečných cestách použijeme tříhodnotovou logiku (platí, neplatí, nelze říci).
- Platnost některých LTL formulí nelze rozhodnout na žádné konečné cestě (např. $GF\ a$).

Pozorování 2

- Cykly tvořené malým počtem stavů jsou procedurou BMC vždy rozbaleny do acyklické cesty délky k.
- Umožníme kódovat cesty, jež mají tvar lasa.
- Tzv. (k, l)-cyklické cesty.

(k,l)-cyklické cesty

(k,l)-cyklické běhy

- Běh $\pi = s_0 s_1 s_2 \ldots$ Kripkeho struktury $M = (S, T, I, s_0)$ je (k, l)-cyklický pokud

$$\pi = (s_0 s_1 s_2 \ldots s_{l-1})(s_l \ldots s_k)^\omega,$$

  kde $0 < l \leq k$ a $s_{l-1} = s_k$.

Pozorování

- Pokud π je (k, l)-cyklický, pak π je též $(k + 1, l + 1)$-cyklický.
- Nahlížení konečné cesty délky jako (k, k)-cyklické je nekorektní (může vytvořit neexistující běh M).
- Každá cesta délky k je acyklická nebo je (k, l)-cyklická.

Sémantika LTL na konečných prefixech běhů

Sémantika LTL pro konečné prefixy

- Mějme π běh Kripkeho struktury M.
- Nechť je dáno k.
- $\pi = \pi^0$

$$\begin{aligned} \pi^i \models_{nl} X\,\varphi \quad &\text{iff} \quad i < k \wedge \pi^{i+1} \models_{nl} \varphi \\ \pi^i \models_{nl} \varphi\,U\,\psi \quad &\text{iff} \quad \exists j. i \leq j \leq k, \pi^j \models_{nl} \psi \text{ and} \\ &\qquad \forall m. i \leq m < j, \pi^i \models_{nl} \varphi \end{aligned}$$

Sémantika $\models_k$ pro LTL při BMC

- Pro (k, l)-cyklické cesty platí, že $\pi \models_k \varphi \iff \pi \models \varphi$.
- Pro necyklické cesty platí, že $\pi \models_k \varphi \iff \pi^0 \models_{nl} \varphi$.
- $\models_k \Longrightarrow \models_{k+1}$, $\models_k$ aproximuje $\models$

BMC pro LTL

Cíl

- Konstruujeme boolovskou formuli $[M, \varphi, k]$, která je splnitelná právě když Kripkeho struktura M má běh π takový, že $\pi \models_k \varphi$.
- $[M, \varphi, k] \equiv [M]^k \wedge [\varphi, k]$

Kódování

- $[M]^k$ kóduje všechny cesty délky k
- $[\varphi, k] \equiv _[\varphi, k]_0 \vee \bigvee_{l=1}^{k} {}_l[\varphi, k]_0$
- $_[\varphi, k]_0$ kóduje, že cesta je acyklická a $\models_{nl} \varphi$
- ${}_l[\varphi, k]_0$ kóduje, že cesta je (k, l)-cyklická a $\models \varphi$

Triky v BMC pro LTL

Fragment LTL-X

- Redukce počtu přechodů (redukce velikosti zadání SAT).
- Podobné principy jako redukce částečným uspořádáním.

Pro zájemce

- Keijo Heljanko: Bounded Model Checking for Finite-State Systems
  `http://users.ics.aalto.fi/kepa/qmc/slides-heljanko-2.pdf`
- Keijo Heljanko and Tommi Junttila: Advanced Tutorial on Bounded Model Checking
  `http://users.ics.aalto.fi/kepa/acsd06-atpn06-bmc-tutorial/lecture1.pdf`

Shrnutí pro BMC

Výhody BMC

Obecné

- Redukce na standardní problém SAT, vývoj v oblasti řešičů SAT se projevuje i na BMC přístupu.
- Často vrací protipříklady minimální délky (ne vždy).
- Boolovské formule mohou být kompaktnější než OBDD reprezentace.

Verifikace HW

- Díky k-indukci velmi úspěšná metoda.

Verifikace SW

- Dle Software Verification Competition (SV-COMP) je aktuálně BMC (rozšířené o využití SMT) mezi nejlepšími metodami pro verifikaci (spíše falsifikaci) software.

Nevýhody BMC

Obecné

- Obecně neúplná metoda.
- Velké instance SAT jsou stále neřešitelné.

Verifikace SW

- Problematická analýza dynamických datových struktur.
- Problematická analýza cyklů.
- Neefektivní pro úplnou aritmetiku (částečně zvládá SMT).

Nástroje a k zamyšlení ...

Nástroje

- CBMC – BMC pro ANSI-C.
- ESBMC – využívá SMT, nadstavba CBMC.
- LLBMC – BMC nad LLVM bitkódem.

K zamyšlení ...

- Čím se liší moderní SMT-BMC od symbolické exekuce?
- Koncepčně velmi podobné, rozlišení je v omezeném rozbalení cyklů a v jiném pořadí prohledávání (prořezaného) stromu symbolické exekuce.

IV113 Validace a verifikace

Lehký úvod do analýzy programů

Jiří Barnat

Analýza programů

Cíle programové analýzy

- Odvodit vlastnosti programů z jejich zdrojového kódu a ...
- ... využít je pro optimalizaci programů.
- ... využít je pro (alespoň částečnou) **verifikaci programů**.

Nerozhodnutelnost

- Všechny zajímavé vlastnosti programů zapsaných v obecném programovacím jazyce jsou nerozhodnutelné.
- Henry Gordon Rice (1953) – Riceovy věty.
- Alan Turing (1936) – Problém zastavení.

Nerozhodnutelné – Jak dál?

Abstrakce

- Zakrytí detailů za účelem zjednodušení analýzy.
- Snaha o korektní, byť neúplná řešení.

Využití abstrakce

- Zjednodušené modelovací jazyky pro popis programu. (Typicky vedou na konečně velký stavový prostor.)
- Vykonávání kódu při uvažované abstrakci – **abstraktní interpretace**.

Jiné cesty – připomenutí jiných přístupů

- Fixovaná, či jinak omezená množina vstupů (testování).
- Omezení zkoumaného prostoru (bounded MC).
- Praktická nerozhodnutelnost (řešiče SMT).

Datové a predikátové abstrakce

Datové abstrakce

Motivace

- Stavová exploze způsobená velkými datovými doménami.
- Redukce myšlenkou doménového testování, tj. nahrazení konkrétní velké datové domény abstrahovanou datovou doménou s menším počtem prvků.

Terminologie

- Abstrakce: mapování konkrétních stavů na abstraktní.
- Konkretizace: mapování abstraktních stavů na množiny konkrétních stavů.

Příklad datové abstrakce

- `Int -> { Even, Odd }`
- Konkrétní stav: $\langle$ `PC:12, A:15, B:0` $\rangle$
- Abstraktní stav: $\langle$ `PC:12, A:Odd, B:Even` $\rangle$

Abstraktní přechodový systém

Přechody v konkrétní a abstraktní sémantice

- Příkaz programu na řádku 12: `A := A+A`
- V konkrétní sémantice:
  $\langle$`PC:12, A:15, B:0`$\rangle \longrightarrow \langle$`PC:13, A:30, B:0`$\rangle$
- V abstraktní sémantice:
  $\langle$`PC:12, A:Odd, B:Even`$\rangle \longrightarrow \langle$`PC:13, A:Even, B:Even`$\rangle$

Nedeterminismus v abstraktním přechodovém systému

- Abstraktní stav: $\langle$`PC:13, A:Even, B:Even`$\rangle$
- Příkaz programu na řádku 13: `A := A div 2`
- $\langle$`PC:13, A:Even, B:Even`$\rangle \longrightarrow$

  $\langle$`PC:14, A:Even, B:Even`$\rangle$

  $\langle$`PC:14, A:Odd, B:Even`$\rangle$

Vztah abstraktního a konkrétního přechodového systému

Nad-aproximace (Over-Approximation)

- Každý běh konkrétního systému je obsažen v konkretizaci nějakého abstraktního běhu.
- Mohou existovat běhy, jež jsou obsaženy v konkretizaci nějakého abstraktního běhu, ale nejsou v původním konkrétním přechodovému systému.

Pod-aproximace (Under-Approximation)

- Každá běh obsažený v konkretizaci libovolného abstraktního běhu je během původního konkrétního přechodového systému.
- Mohou existovat běhy konkrétního přechodového systému, které nejsou obsaženy v konkretizaci žádného abstraktního běhu.

Verifikace aproximovaných přechodových systémů

Značení

- APS – abstraktní přechodový systém
- KPS – konkrétní přechodový systém

Verifikace nad-aproximace

- Absence chyby v APS dokazuje absenci chyby v KPS.
- Chyba v APS může a nemusí být chyba v KPS.
- Chyba v APS, která není chybou v KPS, se označuje jako "spurious error" (false positive, false alarm).

Verifikace pod-aproximace

- Chyba v APS dokazuje přítomnost chyby v KPS.
- Absence chyby v APS nedokazuje absenci chyby v KPS.
- Chyba v KPS, která není zachycena v APS, se označuje jako (false negative).

Příklad – konkrétní sémantika

Příklad

- Je dosažitelný chybový stav v následujícím programu?
- % označuje operaci modulo, `A` je celočíselná proměnná

Kód programu		Hodnota `A` v konkrétní sémantice po provedení příkazu vlevo	
1	`read(A);`	`[int]`	
2	`A = A % 2;`	`[0]`	`[1]`
3	`A = A + 1;`	`[1]`	`[2]`
4	`if (A==0)`	`<false>`	`<false>`
5	**error;**		
6	`else`		
7	`return;`	`<ret>`	`<ret>`

Příklad – datová abstrakce

Příklad

- Je dosažitelný chybový stav v následujícím programu?
- A je abstrahováno do domény {even,odd}

Kód programu		Hodnota A v abstrahované sémantice po provedení příkazu vlevo	
1	read(A);	[even]	[odd]
2	A = A % 2;	[even]	[odd]
3	A = A + 1;	[odd]	[even]
4	if (A==0)	<false>	<true/false>
5	**error;**		<**error**>
6	else		
7	return;	<ret>	<ret>

Predikátová abstrakce

- Predikáty – podmínkové výrazy o valuaci proměnných.
- Jiný způsob jak definovat abstraktní přechodový systém.
- Jedna možná konkrétní definice abstrakce:
  $\langle$čítač instrukcí + valuace zvolených predikátů$\rangle$

Míra abstrakce

- Množství predikátů ovlivňuje přesnost abstrakce.
- Málo predikátů $\rightsquigarrow$ velká nepřesnost, malá stavová exploze
- Více predikátů $\rightsquigarrow$ malá nepřesnost, velká stavová exploze

Příklad

Zadání

- Pro níže uvedený program a uvedené množiny predikátů nakreslete abstraktní přechodový systém, který vznikne použitím metody predikátové abstrakce.
- Ve vámi navrženém přechodovém systému rozhodněte, zda je některá z cest vedoucí do chybového programu realizovatelná.

```
1   read(A);
2   A = A % 2;
3   A = A + 1;
4   if (A==0)
5      error;
6   else
7      return;
```

a) $P1 \equiv A = 0$

b) $P1 \equiv A = 0$, $P2 \equiv A \geq 0$

Poznámky k predikátové abstrakci

Analýza abstraktních cest vedoucích k chybě

- Rozhodnutí o realizovatelnosti (jedná se o falešný alarm?)
- Odvození nových predikátů, které zpřesní abstrakci.

Problém velikosti abstraktního přechodového systému

- S počtem predikátů roste exponenciálně velikost abstraktního přechodového systému.

Možné řešení

- Predikáty svázané s konkrétní lokací v programu.

Metoda CEGAR

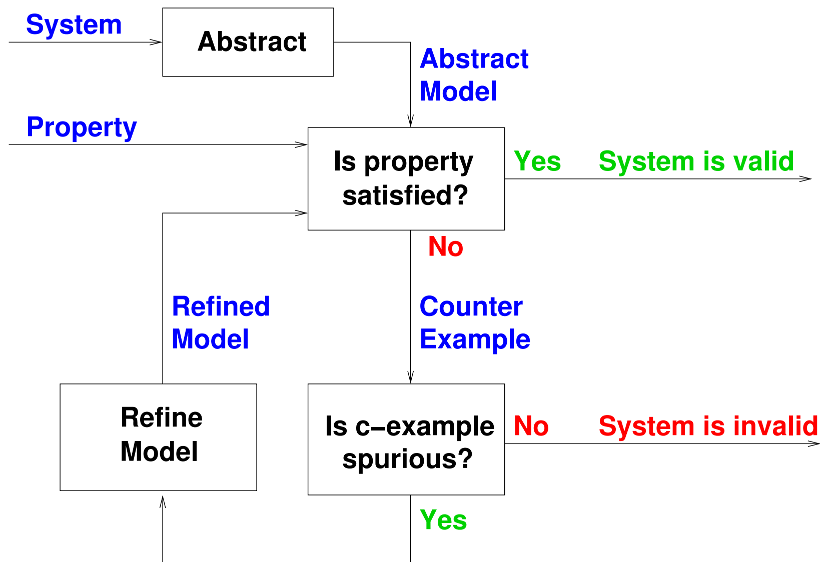
Princip metody CEGAR

- Systém je abstrahován metodou predikátové abstrakce pro iniciální množinu predikátů.
- Abstraktní přechodový systém (nad-aproximace) verifikován metodou ověřování modelu.
- V případě nalezení "spurious" protipříkladu je tento použit k odvození nových predikátů a zpřesnění abstrakce.
- Po zpřesnění abstrakce se postup opakuje.

Poznámky

- Odvozování nových vhodných predikátů je velmi složité.
- Odvozování predikátů v době běhu verifikace (on-the-fly).
- Berkeley Lazy Abstraction Software Verification Tool (BLAST).

Schéma metody CEGAR



Základy abstraktní interpretace

Programová analýza abstraktní interpretací

Reprezentace programu – Flow Graph

- "Speciální verze" grafu toku řízení (Control-Flow Graph).
- Každá hrana má buď právě jednu stráž (podmínku), nebo právě jeden efekt (přiřazení).

Cíl

- Vypočítat vlastnosti jednotlivých vrcholů flow-grafu.

Příklady cílů

- V jakém rozsahu hodnot se v daném místě programu může vyskytnout vybraná proměnná.
- Které proměnné jsou v daném místě programu živé.
- ...

Obecný postup výpočtu

Požadavky

- Doména možných řešení pro jednotlivé vrcholy grafu.
- Iniciální řešení asociované s každým vrcholem grafu.
- Definice toho, jakým způsobem ovlivňuje (aktualizuje) hrana mezi dvěma vrcholy řešení asociované k dotčeným vrcholům.
- Aktualizace řešení způsobené (opakovaným) zpracováním hrany flow grafu je monotónní funkce.

Výpočet

- Vyberu nezpracovanou hranu a aktualizuji řešení u přidružených vrcholů, pokud se řešení změnilo, označím hranu znovu jako nezpracovanou.
- Opakuji, dokud existují nezpracované hrany.

Příklad – výpočet živých proměnných

Konfigurace výpočtu

- Doména = potenční množina množiny všech proměnných.
- Iniciální hodnota asociovaná s vrcholy je $\emptyset$.
- Pro hranu z vrcholu u do vrcholu v je definován update řešení pro vrchol u takto:

$$V(u) = V(u) \cup \Big(V(v) \setminus \mathit{assigned}(u,v) \cup \mathit{used}(u,v)\Big),$$

  kde $V(x)$ označuje množinu řešení asociovaných s vrcholem x, $\mathit{assigned}(u,v)$ a $\mathit{used}(u,v)$ označují proměnné předefinované a použité hranou z u do v.

Pozorování

- V každém bodě výpočtu mám nějaké (aproximující) řešení.
- Dosažení pevného bodu není garance nejlepšího řešení.

Abstraktní interpretace

Pozorování

- Výše uvedený postup je ve své podstatě velmi obecný, vhodnou volbou abstrakce a dalších parametrů lze ověřovat mnoho různých věcí.
- Označuje se jako **abstraktní interpretace**.

Co lze parametrizovat

- Abstraktní doménu řešení.
- Směr aktualizace (po směru, proti směru, obojí).
- Definice update funkcí.
- Jak kombinovat hodnoty v místě spojení cest flow-grafu.
- Pořadí vyhodnocování nezpracovaných hran.
- Detekce časného ukončení.

Relevantní otázky abstraktní interpretace

Existuje pevný-bod?

- Struktura možných řešení asociovaných s jednotlivými vrcholy tvoří úplný svaz.
- Knaster-Tarskiho teorém říká, že na takové doméně má každá monotónní funkce pevný bod.

Terminuje výpočet?

- Pokud neexistuje nekonečná rostoucí posloupnost možných řešení, pak ano.
- V opačném případě nemusí.

Rozšíření (Widening)

- Pomocná transformace vypočítaných mezi-řešení tak, aby zachovalo korektnost, ale zabránilo existenci nekonečně rostoucí posloupnosti, respektive zkrátilo její délku.

Příklad

- Určení číselného intervalu, ve kterém budou hodnoty zpracovány přesně, mimo tento interval budou reprezentovány hodnotou $+\infty$ nebo $-\infty$.
- Posloupnost

  $[0,1] \subset [0,2] \subset [0,3] \subset [0,4] \subset [0,5] \subset [0,6]$ ...

  se pro interval přesnosti [0,3] změní na:

  $[0,1] \subset [0,2] \subset [0,3] \subset [0,+\infty]$

Pomocné techniky – Zúžení

Zúžení (Narrowing)

- Použití rozšíření vede na velmi nepřesné výsledky.
- Může být použit pouze dočasně k terminaci analýzy cyklů.
- Po analýze cyklu možno provést analýzu cyklu znovu a přesně, avšak s iniciální hodnotou získanou po dokončení analýzy cyklu s rozšířením.

Příklad

- Hodnota $[0,+\infty]$ se provedení zúžení dostane na reálnou hodnotu $[0,n]$.

Komentář

- Přesné a další použití technik rozšíření a zúžení je nad rámec tohoto kurzu.

Finální poznámky k abstraktní interpretaci

Další oblasti programové analýzy

- Mezi-procedurální analýza.
- Analýza paralelních programů.
- Generování invariantů.
- Analýza ukazatelů a dynamických datových struktur.
- ...

CPA checker

- The Configurable Software-Verification Platform
- `http://cpachecker.sosy-lab.org/`
- Vítěz mnoha kategoriích v Software Verification Competition.

Domácí úkol

Připomenutí

- Pro možnost udělení hodnocení stupněm A, je nutné vypracovat všechny domácí úlohy.

Zadání domácí úlohy

- Identifikujte ve slajdech místo, které by stálo za to doplnit nějakým vysvětlujícím obrázkem, tento obrázek vytvořte a nejpozději v den konání zkoušky pošlete emailem vyučujícímu spolu s identifikací místa umístění.
- Přednášející si vyhrazuje právo, odmítnout nevhodně, či "lacině" připravený obrázek.

Sekce

A to je konec ...

Shrnutí IV113 a reklama

IV113 – Přehled verifikačních přístupů

- Black-box testing.
- White-box testing a symbolická exekuce.
- Principy deduktivní verifikace.
- Model checking LTL a CTL.
- Bounded model checking.
- Úvod do programové analýzy.

IA159 – Formal Verification Methods

- Detaily vybraných verifikačních metod.
- Programová analýza.
- Verifikace nekonečně stavových systémů.

IV101 – Seminář z verifikace

- Použití verifikačních nástrojů.

Závěrem

Zkoušky

- Bez pomocných materiálů na veškerý odpřednášený obsah.
- Nutno se přihlásit skrze IS.

Výzva

- Prosím o zpětnou vazbu skrze studentskou anketu.
- Uvítám náměty a obrázky na doplnění slajdů.

Děkuji za pozornost!