Predikátová abstrakce

Jiří Hýsek

19. února 2008

Abstrakt

Jedním z největších problémů metod formální verifikace založených na procházení stavového prostoru je tzv. stavová exploze. Reálné systémy mívají obrovské a často nekonečné stavové prostory. Tato práce se soustředí na jeden ze způsobů, jakým se tento problém překonává, na predikátovou abstrakci. V práci se zabývám verifikační technikou známou jako *model checking*. Jsou popsány dvě techniky predikátové abstrakce využité pro účely model checkingu.

Obsah

1 Úvod

V dnešní době jsou hardwarové a softwarové systémy stále častěji používány v oblastech, kde nejsou přípustné chyby: bankovní aplikace, telefonní sítě, řídicí systémy letadel, lékařské přístroje, autonomní vesmírné sondy a podobně. Rozšířenost počítačových systémů, jejich propojení s běžným životem a nároky na jejich spolehlivost jsou den ode dne vyšší. Proto se výzkumu formálních metod verifikace dostává stále větší pozornosti.

V této práci jsou nejdříve velmi stručně vysvětleny základní pojmy z oblasti formální verifikace, zejména model checkingu. Dále je popsáno několik často využívaných technik redukce stavového prostoru a hlavním těžištěm práce je pak popis dvou technik predikátové abstrakce.

1.1 Základní pojmy

Verifikace je proces ověření, zda daný systém splňuje specifikaci korektnosti. Například zda se v daném paralelním systému nevyskytuje deadlock. Poskytuje odpovědi typu ano/ne případně doplněné o diagnostické informace, například za jaké situace se systém nechová podle specifikace. Formální verifikace je založena na formálních, matematických základech. Na rozdíl od jiných technik, jako je například testování, je formální verifikace schopna dokázat správnost chování systému vzhledem k dané specifikaci. Ideální případ verifikace je plně automatický, je zaručeno, že vždy skončí a je *spolehlivý* a *úplný*[1].

Je-li metoda *spolehlivá* (sound), pak v případě, že výsledkem verifikace je výrok, že systém je vzhledem k dané specifikaci korektní, pak je skutečně korektní. Taková metoda pak umožňuje nalézt *všechny* chyby v systému vzhledem k dané specifikaci.

Je-li metoda *úplná* (complete) a výsledkem verifikace je, že systém korektní není, pak je v něm skutečně vzhledem k dané specifikaci chyba. Taková metoda nezpůsobuje žádné falešné poplachy (false alarms).

2 Model checking

Model checking je technika verifikace konečně stavových systémů. Je to způsob automatizovaného ověření, zda model systému (nebo i systém sám) splňuje určitou specifikaci korektnosti. Ta je typicky vyjádřena v některé z temporálních logik jako např. LTL, CTL, CTL^*nebo $\mu-calculus$, ale může jít i o jednodušší specifikaci, jako například použití *assert, end-state* nebo *progress* návěští známé z jazyka *Promela* [4].

Uvažujme, že specifikace je dána formulí temporální logiky. Problém model checkingu pak lze popsat jako hledání množiny stavů v modelu systému, ve kterých platí daná formule. Model systému bývá typicky reprezentován Kripkeho strukturou.

[1] Česká terminologie není ustálena, proto budu v závorce doplňovat anglické názvy vlastností – *soundness* a *completeness* – aby se tato slova nedala podle kontextu vyložit jinak.

Formálně vyjádřeno máme *Kripkeho strukturu* $M = (S, S_0, R, L)$, kde S je množina stavů, S_0 množina počátečních stavů, $R \subseteq S \times S$ je přechodová relace a *labelling* funkce $L : S \to 2^{AP}$, kde AP je množina atomických výroků. Dále máme formuli temporální logiky φ, jež specifikuje požadované chování. Cílem model checkingu je nalézt množinu stavů systému, ve kterých platí formule φ:

$$S_\varphi = \{s \in S \mid M, s \models \varphi\}.$$

Jestliže $S_0 \cap S_\varphi \neq \emptyset$, pak daný systém splňuje specifikaci vyjádřenou formulí φ.

2.1 Problém stavové exploze

Model checking je sám o sobě prostý algoritmus. Ovšem dosažení požadovaných vlastností, zejména spolehlivosti (soundness) verifikačního procesu, je ve většině reálných případů nesrovnatelně obtížnější úkol. Model checking pracuje na principu prozkoumávání všech možných stavů, ve kterých se systém může nacházet. Reálné systémy však mají tendenci mít obrovský a často nekonečný stavový prostor. Jeho prostým vygenerováním a naivním prozkoumáváním jej nejsme s dnešními technickými možnostmi schopni v únosném čase a za použití dostupného množství paměti verifikovat.

Je to jedna z největších překážek, kterou metody formální verifikace založené na prohledávání stavového prostoru musí řešit. Se *stavovou explozí*, jak je tento problém nazýván, se často vyrovnáváme i za cenu zlevnění z požadavků na úplnost (completeness) a s tím související plnou automatizaci metody – chyby, které verifikační proces objeví, nemusí být reálné nebo může vrátit výstup typu "nevím" a rozhodnout musí člověk.

Byla představena spousta různých přístupů k redukci stavového prostoru, je to však stále živá oblast výzkumu.

2.2 Techniky redukce stavového prostoru

V této kapitole se stručně zmíním o některých technikách redukce stavového prostoru. Protože by podrobný popis každé z nich vydal na samostatnou práci, uvedu pouze jejich myšlenku. Od kapitoly 3 se budeme podrobněji zabývat konkrétními technikami patřící do jedné skupiny redukčních metod – predikátové abstrakce.

2.2.1 Předzpracování

Metody patřící do této kategorie jsou založeny na předzpracování modelu verifikovaného systému takovým způsobem, že snižují velikost stavového prostoru a chování upraveného modelu se vzhledem k verifikované vlastnosti nezmění. Předzpracování je často (ale ne nutně vždy) prováděno manuálně již během modelování systému. Jako příklad si uveďme dvě z používaných metod.

Eliminace zbytečných hodnot proměnných. V modelu se mohou vyskytovat proměnné, jejichž konkrétní hodnota je důležitá pouze v určitých fázích výpočtu.

Počet stavů můžeme snížit zahrnutím více různých hodnot do jednoho stavu. Například v době, kdy hodnota proměnné již není pro verifikovanou vlastnost důležitá, můžeme v modelu uvažovat, že její hodnota odpovídá konstantě "undefined" nezávisle na tom, jakou hodnotu ve skutečnosti má.

Zvyšování hrubosti atomicity. Při modelování je možné dosáhnout menšího stavového prostoru tím, že považujeme co největší posloupnosti příkazů za atomické. Atomická posloupnost příkazů nemůže být prokládána akcemi jiných procesů. V rámci atomické posloupnosti tedy akce, které mohou způsobit paralelně běžící procesy, není třeba uvažovat. Zvyšování atomicity může být podporováno modelovacími konstrukcemi, které umožní seskupit sekvenci přechodů nebo příkazů do jednoho atomicky prováděného bloku. Příkladem mohou být konstrukce *atomic*{} nebo *d_step*{} v jazyce *Promela* [4].

2.2.2 Efektivní reprezentace stavů

Metody patřící do této skupiny reprezentují stavový prostor paměťově nenáročným způsobem. Zároveň je požadováno, aby bylo možné z reprezentace stavového prostoru efektivně informace získávat. Opět si stručně popišme několik příkladů technik takovéto reprezentace stavového prostoru.

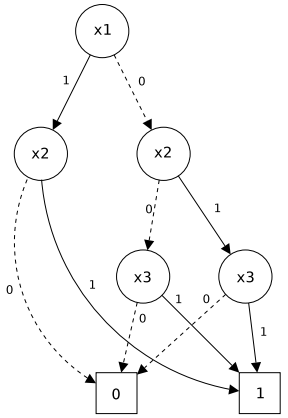
Binární rozhodovací diagramy. Uspořádaný binární rozhodovací diagram [1] (OBDD) je orientovaný acyklický graf, jehož uzly mají dva nebo žádného následníka. Nekoncové uzly jsou označeny Booleovskými proměnnými a jejich odchozí hrany hodnotami "0" nebo "1". Koncové uzly (listy) jsou označeny buď "0" nebo "1". OBDD má jeden kořen (vrchol, do něhož nevedou žádné hrany) a je definováno pevné uspořádání proměnných na cestách vedoucích z kořene. OBDD uchovává množinu vektorů s Booleovskými hodnotami pevné délky. S ohledem na pevné uspořádání proměnných tato množina reprezentuje Booleovskou formuli. Příklad OBDD vidíme na obrázku 1.

OBDD mohou být významně redukovány například spojováním vrcholů, které jsou kořeny izomorfních podstromů. Jejich výhodami je tedy paměťově efektivní reprezentace Booleovských formulí a zároveň je s nimi možné efektivně manipulovat. Využívají se v tzv. *symbolickém model checkingu.* Podrobnější informace o OBDD i symbolickém model checkingu je možné nalézt v [1].

Redukování symetrie stavových prostorů. Model systému často jeví známky symetrie. Například může obsahovat více komponent, které se chovají určitým způsobem ekvivalentně. Pak nemusí být vždy nutné komponenty rozlišovat. Symetrie v modelu mohou být specifikovány manuálně nebo získány automaticky ze struktury modelu popsaném v některém z modelovacích jazyku, jako například *P/T Petriho sítě* [8].

Metoda supertrace. Tato metoda (někdy nazývána *bit-state hashing*) [9] je založena na reprezentaci stavového prostoru jako bitový vektor určité délky. Při generování stavového prostoru je nově nalezený stav interpretován jako řetězec bitů, nad kterým je vyhodnocena hashovací funkce a je získán index do vektoru. Podle hodnoty bitu v bitovém vektoru se určuje, zda byl již daný stav navštíven. Ovšem s tímto přístupem může dojít ke kolizím stavů, což způsobí, že část stavového prostoru nebude prozkoumána. Proto je to technika určena pouze

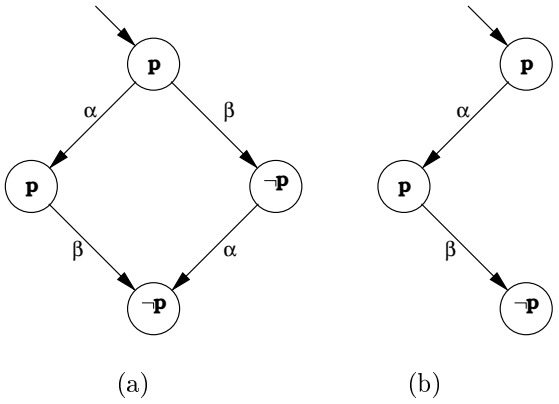
pro částečnou verifikaci. Tato metoda je prakticky implementována například v model checkeru SPIN [4].



Obr. 1: Binární rozhodovací diagram.

2.2.3 Partial-order redukce

Častým předmětem verifikace jsou paralelní systémy. S počtem souběžně běžících procesů roste exponenciálně i stavový prostor celého systému. Generují se všechny kombinace pořadí provádění jednotlivých kroků. Pokud jsou na sobě tyto kroky nezávislé, tedy pořadí jejich provádění nemá na funkci systému vliv, pak není nutné, aby se brala v úvahu všechna možná proložení procesů.



(a) (b)

Obr. 2: Příklad (a) úplného stavového prostoru (b) a jeho redukované podoby.

Toho využívá redukční technika známá jako *partial order redukce* [1]. Obrázek 2a znázorňuje jednoduchý stavový prostor dvou triviálních procesů. Jsou-li přechody α a β nezávislé, stačí ve stavovém prostoru pouze jedno proložení provádění přechodů. Redukovaný stavový prostor je znázorněn na obrázku 2b. Jsou přesně definovány postačující podmínky, které určují, které z přechodů nejsou

závislé a které hrany můžeme ze stavového prostoru vynechat, aniž bychom tím pominuli některé chování systému, důležité pro verifikovanou vlastnost.

Tato redukční technika se s výhodou využívá v nejrozšířenějších model checkerech, jako jsou například SPIN [13] či Java PathFinder [12].

2.2.4 Abstrakce

Abstrakční metody redukují stavový prostor tím, že "shluknou" více konkrétních stavů, které jsou vzhledem k nějaké vlastnosti ekvivalentní, do jednoho abstraktního. Verifikace pak probíhá nad abstraktním stavovým prostorem. Tento prostor je pak pouze aproximací původního. Díky tomu není verifikační proces úplný (complete) nebo spolehlivý (sound), podle typu aproximace. Což může způsobovat falešné poplachy nebo naopak pominout chyby systému vzhledem ke specifikaci. Protože tato práce pojednává o určitém typu abstrakce, budeme se jí podrobněji věnovat v následující kapitole.

3 Predikátová abstrakce

3.1 Základní pojmy

Program P má konečný počet *proměnných* $V = \{v_1, ..., v_n\}$, každá má přiřazenou konečnou *doménu* D_{v_i}. Množina všech možných stavů programu je pak dána kartézským součinem domén $D = D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$.

Výrazy jsou složeny z proměnných z V, konstant z D_{v_i} a funkčních symbolů běžným způsobem, např. $v_1 + 3$. *Atomické formule* se sestávají z výrazů a relačních symbolů, např. $v_1 + 3 < 5$. *Predikáty* jsou složeny z atomických formulí a operátorů negace ($\neg$), konjunkce ($\wedge$) a disjunkce ($\vee$).

Každá proměnná v_i má přiřazen *přechodový blok* (transition block), který definuje její počáteční hodnotu $init(v_i)$ a přechodovou relaci $next(v_i)$.

$$
\begin{array}{l}
\texttt{init}(v_i)\texttt{:=}I_i \\
\texttt{next}(v_i)\texttt{:= case} \\
\qquad\qquad C_i^1 : A_i^1 \\
\qquad\qquad C_i^2 : A_i^2 \\
\qquad\qquad \ldots \\
\qquad\qquad C_i^k : A_i^k
\end{array}
$$

Obr. 3: Obecný přechodový blok.

Na obrázku 3 vidíme obecný přechodový blok proměnné v_i, $I_i \in D_{v_i}$ je počáteční výraz pro proměnnou v_i, každá podmínka C_i^j je predikát a A_i^j je výraz. Sémantika je podobná příkazu *case* z běžných programovacích jazyků. Najde se nejmenší j takové, že podmínka C_i^j je platná a v dalším stavu přiřadíme proměnné v_i hodnotu výrazu A_i^j.

3.2 Abstrakce

Zavedením relace ekvivalence na doménách proměnných můžeme stavový prostor redukovat. Nový abstraktní stavový prostor se skládá z tříd ekvivalencí. Program P koresponduje s Kripkeho strukturou $M = (S, S_0, R, L)$, abstraktní Kripkeho strukturu označíme $\hat{M} = (\hat{S}, \hat{S_0}, \hat{R}, \hat{L})$. Rozlišujeme dva typy abstrakce.

- existenční abstrakce (may abstraction)
- univerzální abstrakce (must abstraction)

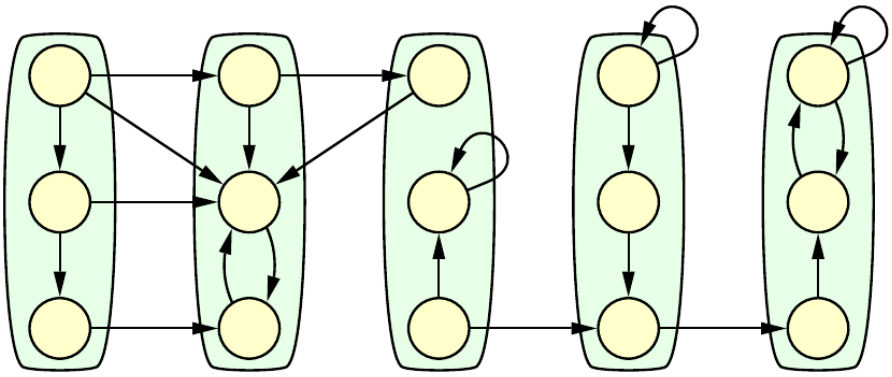
Existenční abstrakce je nad-aproximace původního stavového prostoru. Tato abstrakce neporušuje soundness (tedy nezpůsobuje false positives), ale může způsobit falešné alarmy – nalezené chyby se mohou vyskytovat pouze v abstraktním systému, nikoli však v původním. Ověřované vlastnosti systému jsou udávány jako formule temporální logiky ACTL, což je podmnožina CTL, ve které neexistují existenční kvantifikátory. V abstraktní Kripkeho struktuře existuje přechod mezi stavy, pokud v konkrétní struktuře přechod existuje alespoň z jednoho stavu, který spadá do daného abstraktního stavu. Přesněji vyjádřeno

$$\hat{R} = \{(\hat{s_1}, \hat{s_2}) | \hat{s_1}, \hat{s_2} \in \hat{S} \wedge \exists s_1, s_2 \in S : \alpha(s_1) = \hat{s_1} \wedge \alpha(s_2) = \hat{s_2} \wedge (s_1, s_2) \in R\}.$$

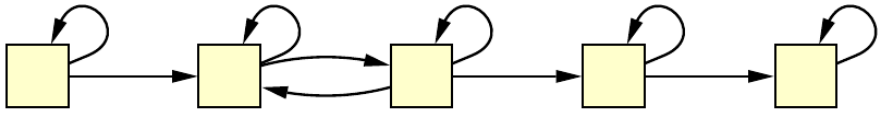
Univerzální abstrakce je naopak pod-aproximace stavového prostoru. Nezpůsobuje falešné alarmy, ale porušuje soundness. Ověřované vlastnosti jsou formule logiky ECTL, což je analogicky s předchozím podmnožina CTL, ve které neexistují univerzální kvantifikátory. Přechod v abstraktní struktuře je tehdy, pokud mezi všemi odpovídajícími konkrétními stavy je přechod. Tedy

$$\hat{R} = \{(\hat{s_1}, \hat{s_2}) | \hat{s_1}, \hat{s_2} \in \hat{S} \wedge \forall s_1 \in S : \alpha(s_1) = \hat{s_1}. \exists s_2 \in S : \alpha(s_2) = \hat{s_2} \wedge (s_1, s_2) \in R\}.$$

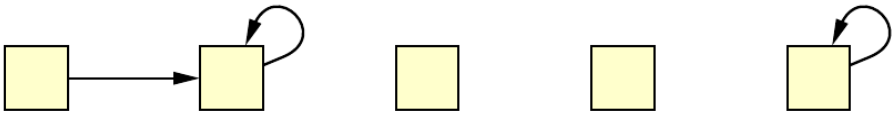
Oba způsoby abstrakce máme názorně zobrazeny na obrázcích 5 a 6. Obrázek 4 ukazuje konkrétní stavový prostor a zelenými oblastmi jeho rozložení podle tříd ekvivalencí. Obrázek 5 ukazuje existenční abstrakci, na obrázku 6 je znázorněna univerzální abstrakce.



Obr. 4: Rozdělení konkrétního stavového prostoru na třídy ekvivalence.



Obr. 5: Existenční abstrakce (nad-aproximace) stavového prostoru z obrázku 4.



Obr. 6: Univerzální abstrakce (pod-aproximace) stavového prostoru z obrázku 4.

3.2.1 Abstrakce pomocí abstrakční funkce

Zavedeme abstrakční funkci $h : S \to \hat{S}$. Jejím jádrem je relace $\pi_h = \{(s_1, s_2) | s_1, s_2 \in S \wedge h(s_1) = h(s_2)\}$. Nová množina stavů abstraktní struktury je tedy

$$\hat{S} = S/\pi_h = \{[s] | s \in S\}$$

a množina počátečních stavů

$$\hat{S_0} = \{[s] | s \in S \wedge s \in S_0\}.$$

Příklad 3.2.1. Uvažujme program s jednou proměnnou x, $D_x = \{1, 2, ..., 12\}$. Zavedeme abstrakční funkci $h : x \in D_X \mapsto \lfloor (x-1)/3 \rfloor + 1$. Získáme třídy ekvivalence
$\hat{1} = \{1, 2, 3\}$,
$\hat{2} = \{4, 5, 6\}$,
$\hat{3} = \{7, 8, 9\}$,
$\hat{4} = \{10, 11, 12\}$.
Abstraktní stavový prostor je $\hat{S} = \{\hat{1}, \hat{2}, \hat{3}, \hat{4}\}$.

3.2.2 Predikátová abstrakce

Predikátová abstrakce využívá množinu predikátů $P = \{p_1, ..., p_n\}$ nad množinou proměnných V. Abstraktní stavový prostor je pak $S_P = \{0,1\}^n$, jeho velikost je tedy 2^n. Každý abstraktní stav $s_P \in S_P$ koresponduje s ekvivalenční třídou v S, jejíž prvky mají stejné ohodnocení predikátů.

Příklad 3.2.2. Uvažujme program se dvěma proměnnými x a y. Domény D_x a D_y jsou stejné a rovnají se množině $\{0,1,2\}$. Množina predikátů je $P = \{(x = y), (x < y), (y = 2)\}$. Velikost stavového prostoru $|S_P| = 2^3 = 8$. Jednotlivé třídy ekvivalencí (abstraktní stavy) jsou následující.

$$
\begin{array}{ll}
\hat{1} = \{(0,0),(1,1)\}, & h((0,0)) = h((0,1)) = (1,0,0) \\
\hat{2} = \{(0,1)\}, & h((0,1)) = (0,1,0) \\
\hat{3} = \{(0,2),(1,2)\}, & h((0,2)) = h((1,2)) = (0,1,1) \\
\hat{4} = \{(2,2)\}, & h((2,2)) = (1,0,1) \\
\hat{5} = \{(1,0),(2,0),(2,1)\}, & h((1,0)) = h((2,0)) = h((2,1)) = (0,0,0)
\end{array}
$$

Ostatní tři stavy $(1,1,0), (1,1,1), (0,0,1)$ jsou neplatné – nemohou nikdy nastat.

3.3 Counterexample-Guided Abstraction Refinement

Technika zvaná *counterexample-guided abstraction refinement* (CEGAR) byla představena v [3]. Tato kapitola je výtahem z [3], nezachází do takových podrobností, zaměřuje se spíše na vysvětlení principu.

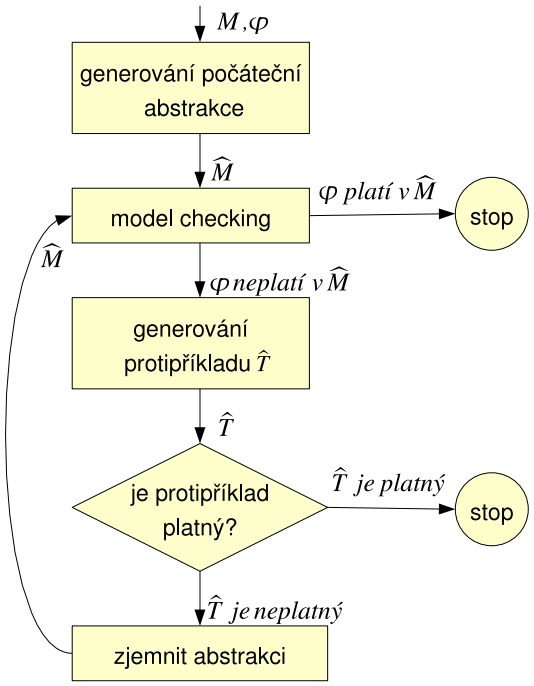
Cílem verifikace s využitím této techniky je zjistit, zda daná formule φ temporální logiky $ACTL^*$ platí v Kripkeho struktuře M, která odpovídá danému programu P. Metoda se skládá z několika kroků:

1. *Generování počáteční abstrakce*
   Z přechodových bloků jednotlivých proměnných vygenerujeme počáteční abstrakci. Podrobněji se ke generování abstrakce vrátíme v kapitole 3.3.1.

2. *Model checking abstraktní struktury*
   Mějme abstraktní Kripkeho strukturu $\widehat{M}$ podle abstrakce h. Ověříme, zda platí $\widehat{M} \models \varphi$. Pokud to model checking potvrdí, pak platí i $M \models \varphi$. Naopak pokud objeví protipříklad $\widehat{T}$, je nutné zjistit, zda nejde o protipříklad způsobený nad-aproximací. Pokud jde o platný protipříklad, tedy pokud platí i v M, pak to nahlásí uživateli a skončí. V opačném případě pokračuje bodem 3. Způsob identifikace neplatných protipříkladů bude rozveden v kapitole 3.3.2.

3. *Zjemnění abstrakce*
   Abstrakci zjemníme rozdělením jedné třídy ekvivalence tak, že zjemněná struktura $\widehat{M}$ již protipříklad $\widehat{T}$ nepřipustí. Způsob rozdělení bude nastíněn v kapitole 3.3.3.

Popsaný postup je přehledně znázorněn na obrázku 7.



Obr. 7: Znázornění model checkingu se zjemňováním abstrakce na základě protipříkladu (CEGAR).

3.3.1 Generování počáteční abstrakce

Uvažujme daný program P s n proměnnými $\{v_1, .., v_n\}$. Množinu atomických formulí programu P budeme označovat $Atoms(P)$. Je-li $f \in Atoms(P)$ atomická formule, pak $var(f)$ je množina proměnných, které se ve formuli f vyskytují, např. $var(x = y) = \{x, y\}$. Pro množinu atomických formulí U je $var(U)$ sjednocení všech množin $var(f)$ pro každé $f \in U$, tedy $\bigcup_{f \in U} var(f)$.

Formule f_1 a f_2 *interferují* tehdy a pouze tehdy, když $var(f_1) \cap var(f_2) \neq \emptyset$. Zaveďme relaci ekvivalence $\equiv_I$ nad množinou $Atoms(P)$, která je reflexivním a tranzitivním uzávěrem relace interference. Ekvivalenčním třídám vzniklých rozkladem $Atoms(P)/\equiv_I$ říkáme *shluky formulí*. Proměnná v_i se nemůže vyskytovat zároveň ve formulích vyskytující se ve více různých shlucích.

Shluky formulí indukují relaci ekvivalence $\equiv_V$ na množině proměnných V. $v_i \equiv_V v_j$ tehdy a pouze tehdy, pokud se vyskytují ve formulích patřících do jednoho shluku. Analogicky třídám ekvivalence podle $\equiv_V$ říkáme *shluky proměnných*. Např. uvažujme množinu formulí $FC_i = \{v_1 > 3, v_1 = v_2\}$. Odpovídající shluk proměnných je $VC_i = \{v_1, v_2\}$.

Uvažujme množinu shluků formulí $\{FC_1, ..., FC_m\}$ a odpovídající množinu shluků proměnných $\{VC_1, ..., VC_m\}$. Počáteční abstrakci $h = (h_1, ..., h_m)$ sestrojíme následovně:

- Pro každé h_i stanovíme doménu D_{VC_i} odpovídajícího shluku proměnných VC_i (kartézský součin jednotlivých domén D_v pro každé $v \in VC_i$).
- Pro každý shluk proměnných $VC_i = \{v_{i_1}, ..., v_{i_k}\}$ je odpovídající abstrakce h_i definována na D_{VC_i} následovně

  $h_i(d_1, ..., d_k) = h_i(e_1, ..., e_k)$ iff $\forall f \in FC_i : (d_1, ..., d_k) \models f \Leftrightarrow (e_1, ..., e_k) \models f$

  jinými slovy dvě obecně různá ohodnocení proměnných jsou ve stejné třídě ekvivalence, pokud nejsou rozlišitelné žádnou z atomických formulí $f \in FC_i$, tzn. pokud vyhodnocení formule f po dosazení hodnot $d_1, ..., d_k$ má stejnou pravdivostní hodnotu, jako po dosazení $e_1, ..., e_k$.

Příklad 3.3.1. Uvažujme program P s proměnnými x, y, $D_x = D_y = \{0, 1, 2\}$ a $reset$, $D_{reset} = \{TRUE, FALSE\}$. Přechodové bloky máme zobrazeny na obrázku 8.

- Množina atomických formulí je $Atoms(P) = \{(reset = TRUE), (x = y), (x < y), (y = 2)\}$.
- Jsou zde dva shluky formulí: $FC_1 = \{(x = y), (x < y), (y = 2)\}$, $FC_2 = \{(reset = TRUE)\}$,
- odpovídající shluky proměnných jsou $VC_1 = \{x, y\}$ a $VC_2 = \{reset\}$.
- Sestrojíme abstrakci $h = (h_1, h_2)$:
  - h_1: Odpovídající doména $D_{VC_1} = \{0, 1, 2\} \times \{0, 1, 2\}$, ekvivalenční třídy vypadají následovně:
    $\hat{0} = \{(0, 0), (1, 1)\}$, protože platí $h_1(0, 0) = h_1(1, 1)$
    $\hat{1} = \{(0, 1)\}$
    $\hat{2} = \{(0, 2), (1, 2)\}$, protože platí $h_1(0, 2) = h_1(1, 2)$
    $\hat{3} = \{(1, 0), (2, 0), (2, 1)\}$, protože $h_1(1, 0) = h_1(2, 0) = h_1(2, 1)$
    $\hat{4} = \{(2, 2)\}$
    Tedy abstrakce $h_1 : \{0, 1, 2\}^2 \mapsto \{\hat{0}, \hat{1}, \hat{2}, \hat{3}, \hat{4}\}$.
  - h_2: Doména $D_{VC_2} = \{TRUE, FALSE\}$, tudíž h_2 je pouze funkce identity. Tedy $h_2(reset) = reset$.
- Pro výpočet abstraktního modelu použijeme existenční abstrakci.

Konkrétní stavový prostor by měl počet stavů daný velikostí kartézského součinu domén všech proměnných, tedy $|D_x \times D_y \times D_{reset}| = 18$. Abstraktní stavový prostor má velikost $|D_{VC_1} \times D_{VC_2}| = 10$.

```
init(x) := 0                    init(y) := 0
next(x) := case                 next(y) := case
   reset = TRUE : 0                reset = TRUE : 0
   x < y : x + 1                 (x=y)∧¬(y=2) : y + 1
   x = y : 0                     (x=y) : 0
   else: x                         else: y
```

Obr. 8: Příklad přechodových bloků proměnných x a y.

3.3.2 Model checking na abstraktním modelu

Je dána $ACTL^*$ specifikace φ a abstraktní Kripkeho struktura $\widehat{M}$ sestrojená odpovídající abstrakční funkcí h. Technika popisovaná v [3] používá standardní symbolický model checking k ověření, zda abstraktní Kripkeho struktura $\widehat{M}$ splňuje specifikaci φ. Pokud ano, pak φ díky nad-aproximaci splňuje i konkrétní Kripkeho struktura M. V opačném případě předpokládáme, že model checker vygeneroval protipříklad $\widehat{T}$ a je třeba ověřit, zda je platný (tedy zda se vyskytuje i v M). $\widehat{T}$ může být buď konečná cesta nebo cyklus v $\widehat{M}$.

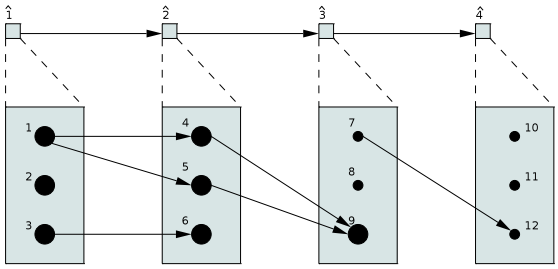
Identifikace neplatných protipříkladů v cestách
Nejprve se budeme zabývat případem, kdy protipříklad $\widehat{T}$ je cesta $\langle \widehat{s_1}, ..., \widehat{s_n} \rangle$. Množinu konkrétních stavů odpovídající abstraktnímu stavu $\hat{s}$ budeme značit $h^{-1}(\hat{s})$. Tzn. $h^{-1}(\hat{s}) = \{s | h(s) = \hat{s}\}$. h^{-1} rozšíříme i na sekvence tak, že $h^{-1}(\widehat{T})$ je množina všech konkrétních cest odpovídajících abstraktní cestě $\widehat{T}$, tedy

$$h^{-1}(\widehat{T}) = \{\langle s_1, ..., s_n \rangle | \bigwedge_{i=1}^{n} h(s_i) = \widehat{s_i} \wedge s_1 \in S_0 \wedge \bigwedge_{i=1}^{n-1} (s_i, s_{i+1}) \in R\},$$

kde S_0 je množina počátečních stavů konkrétní Kripkeho struktury a R její přechodová relace. Je to tedy množina konkrétních cest, které začínají v počátečním stavu. Je-li $h^{-1}(\widehat{T})$ prázdná, je zřejmé, že protipříklad je neplatný, a pokračuje se zjemněním abstrakce. V opačném případě je protipříklad reálný, tudíž se nahlásí uživateli a verifikace může být ukončena.

Lemma 1. *Následující tři věty jsou ekvivalentní:*
(i) Cesta $\widehat{T}$ odpovídá konkrétnímu (platnému) protipříkladu.
(ii) Množina konkrétních cest $h^{-1}(\widehat{T})$ je neprázdná.
(iii) $\forall 1 \leq i \leq n$: $S_i \neq \emptyset$

Příklad 3.3.2. Uvažujme program z příkladu 3.2.1. Přechody mezi stavy jsou znázorněny šipkami na obrázku 9. Menší body znázorňují nedosažitelné stavy. Předpokládejme, že jsme obdrželi protipříklad $\widehat{T} = \langle \hat{1}, \hat{2}, \hat{3}, \hat{4} \rangle$. Tento protipříklad je neplatný. S použitím terminologie z lemmatu 1 máme množiny $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$ a $S_4 = \emptyset$. Tedy i $h^{-1}(\widehat{T}) = \emptyset$ a protipříklad v konkrétním systému neexistuje.

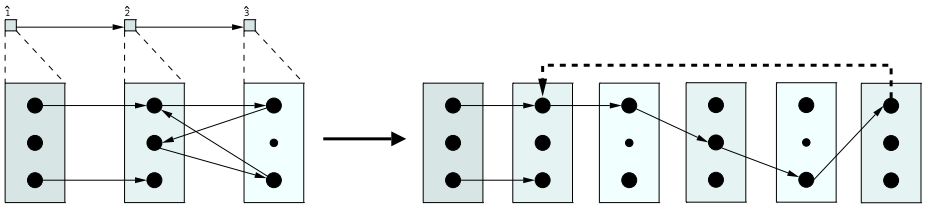


Obr. 9: Abstraktní protipříklad (cesta) a odpovídající konkrétní stavy v dané cestě.

Identifikace neplatných protipříkladů v cyklech
Nyní předpokládejme, že protipříklad $\widehat{T}$ obsahuje smyčku. Pak protipříklad rozvineme, jak je znázorněno na obrázku 10. V určité fázi se rozvíjení stane periodické. Stačí nám rozvinout jeden cyklus (na obrázku znázorněn tlustou čárkovanou šipkou). Rozvinutý protipříklad značíme $\widehat{T}_{unwind}$.

Teorém 1: *Následující dvě věty jsou ekvivalentní:*
(i) $\widehat{T}$ *odpovídá konkrétnímu protipříkladu.*
(ii) $h^{-1}(\widehat{T}_{unwind}) \neq \emptyset$.

Z teorému 1 můžeme usoudit, že algoritmus pro identifikaci neplatného protipříkladu, který je cestou, je možné využít i pro případ, kdy protipříklad obsahuje cykly.



Obr. 10: Rozvinutí protipříkladu obsahující smyčku.

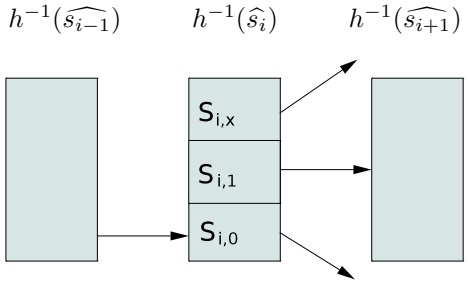
3.3.3 Zjemňování abstrakce

Uvažujme, že protipříklad $\widehat{T} = \langle \widehat{s_1}, ..., \widehat{s_n} \rangle$ je cesta. V této fázi již můžeme říci, že $\widehat{T}$ neodpovídá reálnému protipříkladu, tudíž podle lemmatu 1 (iii) existuje množina konkrétních stavů $S_i \subseteq h^{-1}(\widehat{s_i})$ kde $1 \leq i \leq n$, takových, že ze žádného z nich nevede přechod do žádného z množiny konkrétních stavů $h^{-1}(\widehat{s_{i+1}})$. Např. na obrázku 9 existuje v protipříkladu přechod ze stavu $\hat{3}$ do stavu $\hat{4}$. Množina dostupných konkrétních stavů $S_3 = \{9\}$, ale ze stavu 9 nevede žádná hrana do některého ze stavů $\{10, 11, 12\}$ ($h^{-1}(\hat{4}) = \{10, 11, 12\}$). Protože v abstraktním struktuře hrana mezi $\hat{3}$ a $\hat{4}$ existuje, musí být v $h^{-1}(\hat{3})$ alespoň jeden stav, ze

kterého vede hrana do některého ze stavů $h^{-1}(\widehat{4})$. Aby po zjemnění protipříklad neplatil, je nutné takové stavy odlišit. Rozdělíme tedy původní množinu konkrétních stavů $h^{-1}(\widehat{s_i})$ na tři podmnožiny $S_{i,0}$, $S_{i,1}$, $S_{i,x}$.

- $S_{i,0} = S_i$ ... zde bude množina stavů, do kterých vede hrana z S_{i-1},
- $S_{i,1} = \{s \in h^{-1}(\widehat{s_i}) | \exists s' \in h^{-1}(\widehat{s_{i+1}}), (s, s') \in R\}$ ... množina stavů z $h^{-1}(\widehat{s_i})$, ze kterých vede hrana do některého ze stavů $h^{-1}(\widehat{s_{i+1}})$,
- $S_{i,x} = h^{-1}(s_i) \backslash (S_{i,0} \cup S_{i,1})$ ... množina zbývajících stavů.

Rozdělení máme přehledně znázorněno na obrázku 11. K dosažení rozdělení stavů se upravuje relace ekvivalence tak, aby došlo ke správnému rozdělení ekvivalenční třídy $\widehat{s_i}$. Podrobnější informace k implementaci je možné nalézt v [3].



Obr. 11: Rozdělení do množin $S_{i,x}$, $S_{i,1}$ a $S_{i,0}$.

3.4 Lazy abstrakce

Problémem model checkingu typu "abstrakce – verifikace – zjemnění na základě protipříkladu", jak jsme si jej popsali v předchozí kapitole, je vysoká výpočetní náročnost. V [2] je představena technika, která zavádí určité optimalizace a přináší některé další výhody. Této technice budeme dále říkat *lazy abstrakce*, nepřijde mi vhodné zde zavádět nezažitou českou terminologii, i když bychom o ni mohli mluvit jako o "líné abstrakci". Následující kapitoly jsou stručným výtahem z [2] a opět není cílem jít do přílišné hloubky a formálních detailů. Zájemce odkazuji na zmíněný článek.

Lazy abstrakce ve stručnosti pracuje následovně. První dva kroky (abstrakce, verifikace) zůstanou. Ve třetím kroku, kdy víme, že protipříklad není konkrétní, najdeme abstraktní stav, ve kterém se protipříklad od konkrétního odlišuje, a zjemňujeme přímo abstraktní model od toho stavu. Ostatní stavy v případě, se jich zjemnění abstrakce nijak nedotýká, zůstávají a není třeba je vytvářet znovu. Různé části modelu mohou mít různý stupeň abstrakce.

Potom se verifikuje stavový prostor, který byl zjemněním postihnut. Stavy, na které s upravenou částí stavového prostoru nesouvisí, není nutné znovu procházet. Například za předpokladu, že je stavový prostor na počátku rozdělen na dva nepropojené podgrafy, pak po úpravě jednoho z nich, není nutné verifikovat i druhý.

3.4.1 Lazy abstrakce pracující s programy v C

V této kapitole si neformálně popíšeme způsob verifikace a pro názornost ukážeme postup na konkrétním příkladu programu v jazyce C.

Budeme pracovat s tzv. *control flow automatem* (CFA), což je orientovaný graf, jehož uzly odpovídají řídicím lokacím v daném programu a hrany přechodům mezi lokacemi. Hrany jsou popsány buď tzv. *základním blokem* příkazů, které se provedou při přechodu z výchozí lokace do cílové, nebo *podmínkou* (*assume predicate*), která pro provedení přechodu musí být vyhodnocena jako *true*.

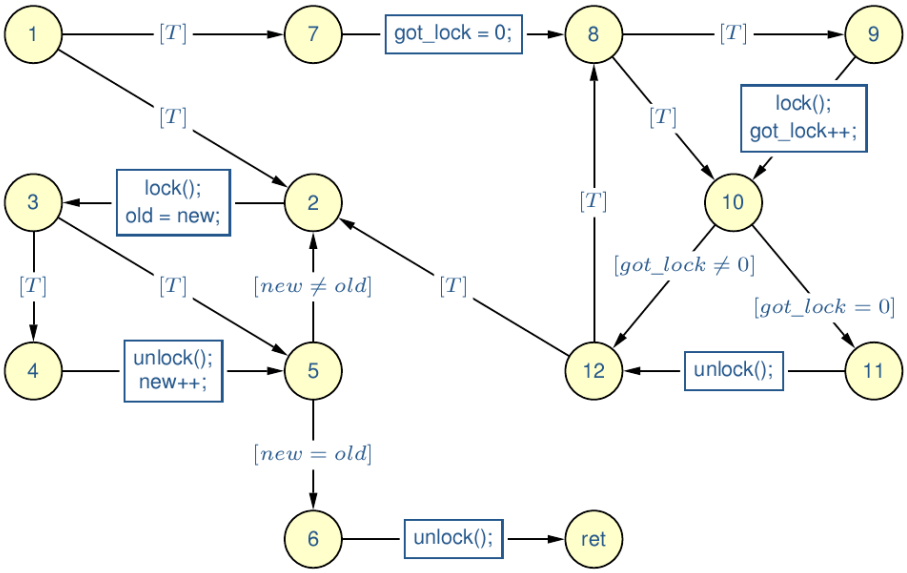
```
example() {
  1: if (*) {
  7:     do {
             got lock = 0;
  8:         if (*) {                    lock() {
  9:             lock();                     if (LOCK == 0) {
                 got lock++;                     LOCK = 1;
             }                               } else {
 10:         if (got lock) {                     ERROR
 11:             unlock();                   }
             }                             }
 12:     } while (*);
     }                                     unlock() {
  2: do {                                      if (LOCK == 1) {
         lock();                                  LOCK = 0;
         old = new;                           } else {
  3:     if (*) {                                 ERROR
  4:         unlock();                        }
             new++;                        }
         }
  5: } while (new != old);
  6: unlock();
  }
```

Obr. 12: Ukázkový program v C – funkce example(), lock() a unlock().

Na obrázku 13 vidíme CFA funkce *example()*, kterou vidíme na obrázku 12. Návěští v programu korespondují s vrcholy automatu. Hrany s obdélníky jsou popsány základními bloky, podmínky pro provedení hrany jsou vyznačeny predikátem v hranatých závorkách. $[T]$ znamená *true*, tedy podmínka je splněna vždy. $if\ (*)$ znamená větvení programu závislé na predikátech, které nejsou modelovány (může jít např. o kontroly výsledků IO akcí a podobně). V takovém případě předpokládáme, že predikát může být vyhodnocen jak na *true* tak i na *false*.

Model checking se provádí na CFA. Pro zjednodušení uvažujme, že *lock*() a *unlock*() jsou atomické operace. Při zavolání funkce *lock*() se globální proměnná $LOCK$, má-li hodnotu 0, nastaví na 1. Pokud již měla hodnotu 1, přejde systém

do stavu *ERROR*. Obdobně pracuje i funkce *unlock*().



Obr. 13: CFA funkce *example()*.

Dopředné hledání
První fází algoritmu je dopředné hledání. Jde o procházení control flow automatu stylem depth-first search. Během toho se vytváří prohledávací strom odpovídající procházení automatu, jak můžeme vidět na obrázku 15a. Vrcholy jsou označeny formulemi, takzvanými *dosažitelnými regiony* (reachable regions), které reprezentují to, co doposud známe o daném stavu programu vzhledem k množině uvažovaných predikátů. Každý další dosažitelný region získáme z regionu rodičovského stavu a příkazů na hraně mezi nimi. Dosažitelný region je popsán konjunkcí těchto predikátů.

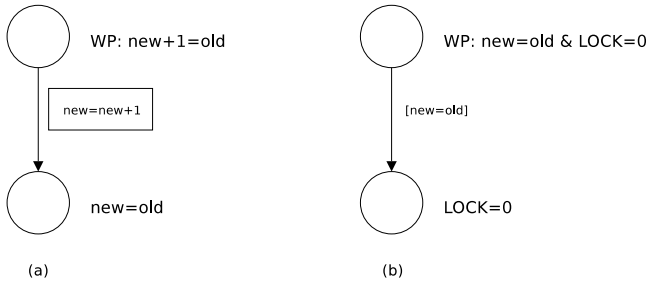
Příklad 3.4.1. Držme se příkladu, jehož CFA vidíme na obrázku 13. Začneme ve stavu 1. Jediná informace, kterou o stavu máme, je, že platí predikát $LOCK = 0$. Hrana mezi stavy 1 a 2 může být vždy provedena (díky podmínce $[T]$, která vždy platí) a nemá na proměnnou $LOCK$ žádný vliv, proto i ve stavu 2 platí $LOCK = 0$. Při přechodu mezi stavy 2 a 3 se volá příkaz *lock*() a $old = new$. Dosažitelný region stavu 2 neobsahuje žádnou informaci o proměnných *new* a *old*, proto tento příkaz na náš predikát nemá vliv. Kdežto volání *lock*() nastaví $LOCK$ na 1, stav 3 tedy označíme predikátem $LOCK = 1$. Přechod mezi 3 a 4 opět nemá vliv, přechod mezi 4 a 5 nastaví LOCK na hodnotu 0, ve stavu 5 tedy platí $LOCK = 0$. Ze stavu 5 do 6 můžeme přejít v případě, že $new = old$, ovšem o těchto proměnných nemůžeme nic usoudit, proto do stavu 6 přejdeme a stále platí, že $LOCK = 0$. Při přechodu ze stavu 6 do stavu *ret* se volá *unlock()*, ale protože víme, že $LOCK = 0$, pak se tato akce vyhodnotí

jako chybné použití funkce unlock() a přejde se do stavu $ERROR$.

V případě, že v dopředném běhu narazíme na chybový stav, musíme ověřit, zda je skutečně o reálnou chybu, nebo je způsobena příliš hrubou abstrakcí. K tomu je určena tzv. *zpětná analýza protipříkladu* (backwards counterexample analysis).

Zpětná analýza protipříkladu
Fázi zpětné analýzy ukazuje obrázek 15b. Vygenerovaný strom procházíme od chybového stavu pozpátku a každému stavu přiřazujeme tzv. *chybový region* (bad region) B. Na obrázku ho označuje predikát ve složených závorkách. Chybový stav má vždy hodnotu $B = true$.

Dále budeme pracovat s tzv. *nejslabším předpokladem* (weakest precondition). Formule P' je nejslabší předpoklad formule P vzhledem k akci A pokud platí P' před a P po provedení akce A. Příklady nejslabších předpokladů vidíme na obrázku 14.



Obr. 14: Nejslabší předpoklad WP. (a) Přechod obsahuje základní blok. (b) Přechod obsahuje podmínku.

Při zpětném procházení přiřazujeme stavům chybový region. Chybový region rodičovského stavu B_{parent} je nejslabší předpoklad vzhledem k chybovému regionu aktuálního stavu B a označení AP přechodu mezi těmito stavy, tedy $B_{parent} = wp(B, AP)$.

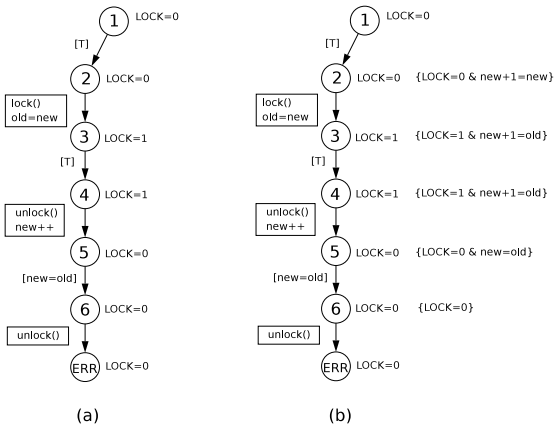
Při postupu směrem zpátky z chybového stavu ve vygenerovaném stromu hledáme stav, jehož chybový region má prázdný průnik s dosažitelným regionem, což indikuje, že není možné touto cestou dojít k chybovému stavu a protipříklad je tedy neplatný. Stav, ve kterém tuto skutečnost zjistíme, nazveme *pivot*.

Příklad 3.4.2. Uvažujme strom s chybovými regiony podle obrázku 15b. Stavy, které ve zdrojovém kódu z návěští 6 mohou po zavolání unlock() skončit v chybovém stavu jsou právě takové, kde platí $LOCK = 0$. Chybový region stavu 6 tedy bude $LOCK = 0$. Podobně chybový region stavu 5 bude nejslabší podmínka predikátu $LOCK = 0$ vzhledem k podmínce na přechodu $[new = old]$, což je predikát $LOCK = 0 \wedge new = old$. Podobně na základě nejslabší podmínky budeme postupně určovat chybové regiony dalších stavů ve stromu. Až dojdeme

ke stavu 2, jehož chybový region je $LOCK = 0 \wedge new + 1 = new$. Toto je první stav, kde je průnik dosažitelného regionu a chybového regionu prázdný (tzn. sjednocení obou predikátů je nesplnitelné). Stav 2 je tedy označen jako pivot.

Pro kontrolu prázdnosti průniku regionů se využívá *theorem prover*, který zjistí, zda je formule vzniklá sjednocením chybového a dosažitelného regionu splnitelná. Pokud není, poskytne *důkaz nesplnitelnosti*, ze kterého zjistíme, které predikáty jsou důležité pro zjemnění abstrakce. V případě příkladu 3.4.2 by šlo o predikát $old = new$. Abstrakci dále budeme zjemňovat od pivotu dopředu, tudíž predikát $old = new$ přidáme do reachable regionu podstromu stavu 2 a znovu analyzujeme celý podstrom.

Pokud zpětná analýza prošla až do kořene stromu bez toho, aby narazila na nesplnitelnou konjunkci formulí, potom je nalezená chyba reálná.



Obr. 15: (a) Dopředné hledání. (b) Zpětná analýza protipříkladu.

4 Závěr

Predikátová abstrakce se jeví jako nadějná technika v boji se stavovou explozí. V této práci jsme si zavedli základní pojmy týkající se model checkingu a redukce stavových prostorů a vysvětlili dva přístupy k predikátové abstrakci.

Oba z přístupů používají schema "abstrakce – verifikace – zjemnění abstrakce na základě protipříkladu", druhý z nich, lazy abstrakce, se snaží omezit výpočetně náročné operace, jakou jsou generování abstraktního modelu a samotný model checking, zjemňováním pouze potřebné části stavového prostoru a verifikací pouze stavů, na které zjemnění abstrakce mělo vliv.

4.1 Některé nástroje využívající predikátovou abstrakci

SLAM je sada nástrojů pro model checking programů psaných v jazyce C. Je vyvíjen v Microsoftu a je využíván k verifikaci ovladačů. Množství dalších

informací je možné nalézt na domovské stránce projektu [10].

BLAST je stejně jako SLAM model checker programů psaných v jazyce C. Jeho hlavním vylepšením je využití myšlenky popisované lazy abstrakce. Využívá externí theorem prover a můžeme jej nalézt na [11].

Jako poslední bych uvedl nástroj vyvíjený ve spolupráci s NASA a to *Java PathFinder*. Slouží k verifikaci bytecodu programů napsaných v jazyce Java. Podle [12] byl tento nástroj také využit při hledání chyb v reálné vesmírné lodi.

Reference

[1] E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking. MIT Press, 2000.

[2] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In Proc. POPL, pages 5870. ACM, 2002.

[3] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Computer Aided Verification, pages 154-169. Springer, 2000.

[4] G. J. Holzmann. The model checker SPIN. IEEE Trans. on Softw. Eng., 23(5):279–295, May 1997.

[5] A. Valmari. The State Explosion Problem. Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, str. 429-528.

[6] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[7] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. ACM Trans. Program. Lang. Syst. 27, 2 (Mar. 2005), str. 314-343. 2005

[8] K. Schmidt. How to Calculate Symmetries of Petri Nets. Technical Report MATH-AL-8-1997, Technical University of Dresden, Dresden, Germany, September 1997.

[9] G. J. Holzmann. An Analysis of Bit-State Hashing. In Proceedings of IFIP/WG6.1 Symposium on Protocol Specification, Testing, and Verification, str. 300-314, Varšava, Polsko, 1995. Chapman and Hall.

[10] Domovská stránka nástroje SLAM, dostupná na URL: http://research.microsoft.com/slam/ (únor 2008)

[11] BLAST: Berkeley Lazy Abstraction Software Verification Tool. Domovská stránka nástroje BLAST, dostupná na URL: http://mtc.epfl.ch/software-tools/blast/ (únor 2008)

[12] Domovská stránka nástroje Java PathFinder, dostupná na URL: http://javapathfinder.sourceforge.net/ (únor 2008)

[13] On-The-Fly, LTL Model Checking with SPIN. Domovská stránka nástroje SPIN, dostupná na URL: http://spinroot.com/spin/whatispin.html (únor 2008)