Abstraktní regulární model checking a separační logika jako vybrané metody verifikace pointerových programů

Lukáš Holík

15. února 2007

Abstrakt

Hlavním tématem této semestrálné práce jsou dvě metody model checkingu procesů s nekonečným stavovým prostorem, které jsou zároveň polem mé práce zde na FIT – abstraktní a abstraktní stromový model checking a separační logika. V úvodu jsem věnoval menší prostor přehledu problematiky, jádro a hlavní část patří abstraktnímu regulárnímu model checkingu, který je nynější oblastí mé práce. Poslední, nakonec jen krátkou část věnuji separační logice, jejímuž studiu se teprve věnuji.

Obsah

1 Úvod

1.1 Verifikace nekonečně stavových procesů

Verifikace konečně stavových procesů je dnes již poměrně dobře zavedená. V praxi se ale setkáváme s procesy, které konečně stavové nejsou. Může to být způsobeno jednak použitím neohraničených struktur, jako jsou jako fronty, zásobníky, neomezené čítače, obecné dynamické datové struktury, nebo rozlišováním stavů podle například času. Případně se může jednat o třídu konečných systémů, kterých je ale nekonečně mnoho (například systémy s pevně ohraničeným zásobníkem). Všechny podobné varianty se vymykají ze záběru tradičního konečně stavového model checkingu a je třeba hledat pro ně metody založené na jiných principech.

1.2 Verifikace pointerových programů

Budu se zde zabývat některými možnostmi automatické verifikace programů využívajících neohraničené rekurzivní datové struktury, typicky půjde o programy postavené na práci s ukazateli. Pointerové programy se obecně nelehko píší, často jsou komplikované a těžko srozumitelné. Zároveň jsou používané programovací techniky velice náchylné k nepříjemným chybám. Přesto je programování s pointery oblíbené a používané. Kvalitní techniky automatické kontroly tokových programů jsou tedy silně žádoucí. Obecně je tento problém – verifikace programů s ukazateli, nerozhodnutelný. Musíme se zde totiž vypořádat s nekonečným stavovým prostorem. Je třeba hledat vhodné

způsoby konečné reprezentace nekonečného stavového prostoru a konstruovat nad nimi algoritmy a heuristiky, schopné vracet co nejpřesnější výsledky. Podobně i časová a prostorová složitost používaných technik odpovídá komplikovanosti problému a je vždy silně netriviální.

V důsledku obtížnosti a atraktivnosti tohoto problému se metody verifikace pointerových programů staly velice živým polem výzkumu a bylo navrženo velké množství technik a rozdílných přístupů. Budu se zde zabývat dvěma poměrně úspěšnými technikami, a to abstraktním regulárním (stromovým) model checkingem a separační logikou. Nejdříve ale uvedu stručný přehled nejdůležitějších alternativních přístupů.

2 Některé alternativní přístupy

V prvé řadě je třeba zmínit často citované přístupy spojené s nástroji PALE a TVLA, což jsou pravděpodobně první nástroje, které se blíží možnosti využití k řešení reálných problémů. Poté velice krátce zmíním příklady dalších alternativ.

2.1 TVLA (Three Valued Logic Analyser)

Základem TVLA [SRW02] je troj-hodnotová Kleeneho výroková logika s tranzitivním uzávěrem na grafech [Kle87]. Datové struktury jsou popsány takzvanými jádrovými predikáty, které zachycují základ, jádro sémantiky struktury, a instrumentačními predikáty, které jsou definovány nad jádrovými a jemněji vyjadřují rozdíly mezi podtřídami datových struktur.

Konečná reprezentace nekonečné třídy grafů paměťových struktur je zde dosažena použitím takzvaných sumárních uzlů, které reprezentují jeden nebo více výskytů podstruktur, splňujících stejné abstrakční predikáty. Jedná se o takzvanou kanonickou abstrakci. V případě aplikace abstrakce, kdy více uzlů grafu je spojeno do jednoho sumárního, je možné, že některé predikátů ztratí svou definitivní hodnotu. Sumární uzel totiž může obsahovat uzly, pro které je predikát platný, i uzly, pro které platný není. Zde se užívá ona „třetí“ hodnota Kleeneho logiky interpretovaná jako „není známo“ nebo „možná“.

Sémantika konkrétního příkazu programu je pak vyjádřena modifikační predikátovou formulí. Ty jsou odvozovány někdy automaticky, ale z velké části musí mýt poskytnuty uživatelem. Metody jejich automatického odvození jsou v současné době předmětem výzkumu. Modifikace struktury podle modifikační formule je založena na mechanismu materializace, kdy jsou sumární uzly do jisté míru „rozbaleny“ – materializovány. Na získané struktuře je provedena modifikace a na výsledek je znova aplikována abstrakce.

TVLA byl testován na různých algoritmech (například třídících) nad různými lineárními a stromovými strukturami. Plně uspokojivý však zatím není ani rozsah zvládaných struktur, ani škála vlastností, které je možno verifikovat.

2.2 PALE (Pointer Assertinon Logic Engine)

Nástroj PALE [MS01] umožňuje poloautomatickou verifikaci programů s pointerovými strukturami. Uživatel totiž musí poskytnout invarianty cyklů. S fragmenty kódu bez cyklů se pak PALE vypořádá plně automaticky. Poměrně rozsáhlá třída pointerových struktur je zde pokryta použitím takzvaných grafových typů [KS93]. Jedná se o reprezentaci struktury pomocí specifikace její stromové kostry. Zbylé pointery jsou vyjádřeny směrovacími formulemi, které specifikůjí cíl pointeru popsáním cesty k němu stromovou kostrou. Problémem zde je, že tyto formule musí mýt deterministické, a navíc význam next-pointeru je dopředu fixován.

Program, invarianty cyklů a vstupně výstupní podmínky jsou zadávány jako formule monadické druhořádové logiky nad grafy. PALE pak zkonstruuje formuli popisující efekt programu a platnost podmínek, formuli zkombinuje s formulemi invariantů cyklů a formulemi vstupně výstupních podmínek a vyhodnotí platnost výsledku. Formule jsou rozhodovány cestou transformace na příslušné formule WS2S (weak monadic scond-order theory of two succesors), které jsou pak rozhodovány cestou využívající stromové automaty. Celá rozhodovací procedura je netriviální.

Přístup byly testován nad datovými strukturami typu seznamů a stromů, ukázal se být poměrně silný a obecný, ale hlavní nevýhodou zůstává nutnost poskytnutí dodatečné, často ne snadno odvoditelné, informace člověkem.

2.3 Příklady dalších alternativních přístupů

Tato sekce si naklade za cíl poskytnou vyčerpávající výčet přístupů. Cílem je spíše poskytnutí představy o jejich rozmanitosti spolu se základními informacemi o několika zajímavých reprezentantech.

- **Predikátová abstrakce**
  Predikátová abstrakce [GS97] je postavena na technice převedení programu na booleovskou verzi, kde booleovské proměnné zachycují pravdivost predikátů vztahujících se ke stavu původního programu. Tento přístup nebyl původně navržen pro programy s dynamickými datovými strukturami, ale pokusy byly následně publikovány v [DN03], [BPZ05] a dalších.

- **Prvořádové odvozování**
  Byly publikovány práce založené na prvořádové axiomatizaci různých variant dosažitelnosti pro dynamické struktury, vhodné pro automatickou dedukci. Práce [LQ06] se poměrně dobře vypořádává se jedno i obousměrně zřetězenými seznamy a jejich cyklickými variantami. Metoda byla úspěšně použita k verifikaci praktických operací nad seznamy, jako je reverze, smazání a přidání prvku, spojení a sjednocení (množin implementovaných jako seznamy). Výsledků je dosahováno plně nebo téměř plně automaticky. Metoda může být navíc snadno zobecněna tak, aby dovolovala verifikovat jednoduché podmínky s omezenou celočíselnou aritmetikou.

- **Alias logika**
  Alias logika [BIL03] je interpretována nad modelem, kde halda není reprezentována jako graf, ale jako soubor regulárních jazyků. Paměťový uzel je popsán regulárním jazykem cest, vedoucích do něj a z něj haldou, která je zde v podstatě konečným automatem s jediným koncovým stavem, který odpovídá danému uzlu.

- **Čítačové automaty**
  Práce [BFL06] uvádí metodu verifikace programů manipulujících seznamy jejich kódováním do čítačových automatů. Stav čítačového automatu kóduje současný stav haldy (shape-grafu haldy) se všemi nepřerušenými pointerovými řetězci kódovanými v jednom uzlu. Čítače kódují délky řetězců. Touto metodou se podařilo verifikovat vlastnosti týkající se uspořádání prvků seznamů nebo i terminace programu. Slibné je spojení čítačových automatů a regulárního model checkingu.

- **Analýza založená na gramatikách**
  Konkrétně [Ven04] se jedná o použití gramatik bezkontextových. Graf paměti je vyjádřen s využitím sumárních uzlů, ke kterým jsou přiřazeny neterminály gramatik. Struktura skrývající se za sumárním uzlem pak odpovídá derivačnímu stromu gramatiky s počátečním neterminálem příslušným uzlu.

- **Časové známky**
  Analýza pomocí časových známek [Ven04] je založena na označování nově alokovaných paměťových objektů celočíselnými známkami, jejichž hodnota je odvozena ze současného stavu proměnných programu. Následně jsou aplikovány metody analýzy programů s numerickými daty a pomocí nich ověřovány vlastnosti relací mezi časovými známkami.

3 Regulární model checking

Metody regulárního model checkingu jsou založeny na myšlence reprezentace stavového prostoru procesu nějakým typem konečného automatu. Slovo (případně term) je kódováním stavu paměti a jazyk určitého automatu odpovídá množině konfigurací paměti. Samozřejmě je takto možné popsat jen ty třídy konfigurací paměti, které odpovídají regulárním množinám. Základem je odpovědět na dvě otázky:

1. Jakým způsobem bude zachycena sémantika programového kódu měnícího stav paměti?

2. Jak zaručit, aby byl verifikační výpočet konečný?

Přístupů je opět řada, jmenujme [BP96], [AJ96], [KMM+97]. Budu se zde soustředit na variantu, která je poměrně úspěšná, a to na abstraktní regulární model checking, posléze na zobecněnou variantu – abstraktní regulární stromový model checking. První technika kóduje stavy paměti formou klasických konečných automatů, druhá ji v podstatě kopíruje, ale konfigurace paměti je kódována konečným automatem stromovým, který má mnohem větší vyjadřovací schopnosti. Přechody mezi stavy paměti se zde dějí takzvanými transducery. Aby se docílilo terminování výpočtu v co možná nejvíce případech a v co nejkratším čase, jsou vyvíjeny akcelerační techniky, zejména abstrakční techniky, které množinu kódovaných konfigurací paměti vhodně nadaproximují.

3.1 Abstraktní regulární model checking

Základní vhled do problematiky je možno najít například v [BHV04].

3.1.1 Základní pojmy

Začněme se základními definicemi nedeterministického konečně stavového automatu a konečně stavového transduceru.

Definice 3.1 Konečně stavový automat *je pětice* $M = (Q, \Sigma, \delta, q_0, F)$ *kde:*

- Q *je konečná množina stavů*
- Σ *je konečná abeceda*
- $\delta : Q \times \Sigma \to 2^Q$ *je přechodová fuke*
- $q_0 \in Q$ *počáteční stav*

- $F \subseteq Q$ *množina koncových stavů.*

Přechodová relace $\xrightarrow[M]{} \subseteq Q \times \Sigma^* \times Q$ *automatu* M *je definována jako nejmenší relace splňující:*

1. $\forall q \in Q : q \xrightarrow[M]{\varepsilon} q$,
2. *jestliže* $q' \in \delta(q, a)$, *pak* $q \xrightarrow[M]{a} q'$
3. *jestliže* $q \xrightarrow[M]{w} q'$ *a zároveň* $q' \xrightarrow[M]{a} q''$, *pak* $q \xrightarrow[M]{wa} q''$, *kde* $a \in \Sigma, w \in \Sigma^*$.

Automat je deterministický, *právě když platí* $\forall q \in Q\ \forall a \in \Sigma : |\delta(q, a)| \leq 1$.

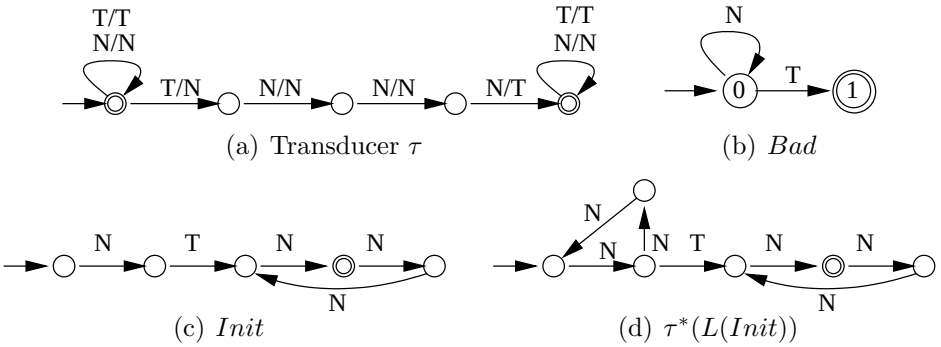
Jazyk *rozpoznávaný konečným automatem* $M = (Q, \Sigma, \delta, q_0, F)$ *ze stavu* $q \in Q$ *je definován jako* $L(M, q) = \{w \in \Sigma^* \mid \exists q_F \in F : q \xrightarrow[M]{w} q_F\}$. *Jazyk* $L(M)$ *automatu* M *je potom roven* $L(M, q_0)$. *Množina* $L \subseteq \Sigma^*$ *je* regulární množinou, *právě když existuje konečný automat* M *takový, že* $L = L(M)$. *Definujeme také* zpětný jazyk *jako* $\overleftarrow{L}(M, q) = \{w \mid q_0 \xrightarrow[M]{w} q\}$ *a dále* dopředný/zpětný jazyk do určité délky*:* $L^{\leq n}(M, q) = \{w \in L(M, q) \mid |w| \leq n\}$ *a podobně* $\overleftarrow{L}^{\leq n}(M, q)$. *Dále definujeme* dopředné/zpětné stopové jazyky *stavů* $T(M, q) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^* : ww' \in L(M, q)\}$ *a podobně* $\overleftarrow{T}(M, q)$. *Nakonec definujeme dopředné/zpětné jazyky* $T^{\leq n}(M, q)$ *a* $\overleftarrow{T}^{\leq n}(M, q)$ stop do určité délky.

Definice 3.2 Konečně stavový transducer *nad abecedou* Σ *je pětice* $\tau = (Q, \Sigma, \delta, q_0, F)$ *kde:*

- Q *je konečná množina stavů*
- Σ *je konečná vstupní/výstupní abeceda*
- $\delta : Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon \to 2^Q$ *je přechodová funkce*
- $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $q_0 \in Q$ *je počáteční stav*
- $F \subseteq Q$ *je množina koncových stavů*

Konečně stavový transducer nazveme délku zachovávající, *pokud jeho přechody neobsahují* ε. *Přechodová relace* $\xrightarrow[\tau]{} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ *je definována jako nejmenší relace splňující:*

1. $q \xrightarrow[\tau]{\varepsilon/\varepsilon} q$ *pro každé* $q \in Q$



(a) Transducer τ (b) *Bad*

(c) *Init* (d) $\tau^*(L(Init))$

Obrázek 1: Transducer τ modeluje protokol předávání tokenu. Automaty popisují iniciální konfigurace, nežádoucí – chybné konfigurace a dosažitelné konfigurace systému.

2. *pokud* $q' \in \delta(q, a, b)$, *pak* $q \xrightarrow[\tau]{a/b} q'$

3. *pokud* $q \xrightarrow[\tau]{w/u} q'$ *a zárovň* $q' \xrightarrow[\tau]{a/b} q''$, *potom* $q \xrightarrow[\tau]{wa/ub} q''$, *kde* $a, b \in \Sigma_\varepsilon, w, u \in \Sigma^*$.

Konečně stavový transducer $\tau = (Q, \Sigma, \delta, q_0, F)$ *reprezentuje relaci* $\varrho(\tau) = \{(w, u) \in \Sigma^* \times \Sigma^* \mid \exists q_F \in F : q_0 \xrightarrow[\tau]{w/u} q_F\}$. *Relace* $\varrho \subseteq \Sigma^* \times \Sigma^*$ *je* regulární, *právě když existuje konečně stavový transducer* τ *takový, že*$\varrho = \varrho(\tau)$. *Buď* $L \subseteq \Sigma^*$ *množina a* $\varrho \subseteq \Sigma^* \times \Sigma^*$ *relace. Značíme* $\varrho(L)$ *množinu* $\{w \in \Sigma^* \mid \exists w' \in L : (w', w) \in \varrho\}$.

3.1.2 Základní schéma ARMC

Nyní co a jak je možné pomocí ARMC verifikovat. Jak bylo zmíněno dříve, kódujeme množinu možných stavů systému jako konečný automat. Pro lepší představu budu citovat příklad z [BHV04]. Jde o jednoduchý parametrizovaný systém, který modeluje token–passing protokol. Systém sestává z parametrického počtu procesů seřazených v poli. Proces buď drží nebo nedrží token. Každý proces může poslat token svému třetímu pravému sousedu. V iniciální konfiguraci na obrázku, kterou popisuje automat *Init*, drží token druhý proces a počet procesů je násobkem tří. Chceme dokázat, že není dosažitelná konfigurace, kdy by token držel poslední proces, kterouž situaci popisuje automat *Bad*.

Vlastnosti, které se chceme pokoušet verifikovat, jsou primárně vlastnosti dosažitelnosti. Pro daný systém s přechodovou funkcí modelovanou transdu-

cerem τ, regulární množinou iniciálních konfigurací $Init$ a množinou nežádoucích konfigurací Bad chceme ověřit, jestli platí že

$$\tau^*(L(Init)) \cap (Bad) = \emptyset).$$

Komplikovanější vlastnosti je možno transformovat na dosažitelnost konstrukcí příslušného automatu a jeho následným sjednocení s verifikovaným systémem. V našem příkladu je $\tau(L(Init))$ znázorněn na obrázku a verifikovaná vlastnost zřejmě platí. Poznamenejme, že množina $\tau^*(L(Init))$ nemusí být regulární a dokonce ani spočetná. V dalším nám půjde o zkonstruování její regulární nadaproximace L takové, že $L \supseteq \tau^*(L(Init))$ a na ní ověřit, že $L \cap L(Bad) = \emptyset$.

3.1.3 Abstrakce

Jak bylo řečeno, klíčovou technikou pro ARMC je technika abstrakce – nadaproximace množin dosažitelných stavů. Definujeme tedy obecné abstrakční schéma takto:

Definice 3.3 *Nechť je Σ konečná abeceda a $\mathbb{M}_\Sigma$ množina všech konečných automatů nad Σ. Pro třídu $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$, je zobrazení α $\mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$* abstrakcí, *pokud splňuje: $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. Nazveme α* konečnou *právě když její definiční obor $\mathbb{A}_\Sigma$ konečný.*

Pro transducer τ a abstrakci α zavedeme pojem abstrakční přechodová funkce τ_α *takto: Pro každý automat $M \in \mathbb{M}_\Sigma$, $\tau_\alpha(M) = \alpha(\hat{\tau}(M))$ kde $\hat{\tau}(M)$ je minimální deterministický automat ekvivalentní s $\tau(L(M))$.*

Nyní je možné iterativně vypočítat sekvenci $(\tau_\alpha^i(M))_{i\geq 0}$. Jelikož předpokládáme, že $\iota \subseteq \tau$, je zřejmé, že pokud α je konečná, existuje $k \geq 0$ takové, že $\tau_\alpha^{k+1}(M) = \tau_\alpha^k(M)$. Definice α implikuje $L(\tau_\alpha^k(M)) \supseteq \tau^*(L(M))$. Tedy celkem můžeme vypočítat nadaproximaci množiny dosažitelných konfigurací $\tau^*(L(M))$ v konečném počtu kroků.

Pokud v takto nadaproximované množině konfigurací zjistíme průnik s množinou nežádoucích konfigurací (přesněji v nějakém kroku iterativního výpočtu zjistíme, že $L(\tau_\alpha^k(M)) \cap Bad \neq \emptyset$), může to znamenat dvě věci:

1. Nalezený průnik má *neprázdný* průnik s přesnou množinou dosažitelných konfigurací, našli jsme tedy protipříklad a můžeme výpočet ukončit s tím, že verifikovaná vlastnost splněna není.

2. Nalezený průnik má *prázdný* průnik s přesnou množinou dosažitelných konfigurací. Použitá abstrakce je tedy pro tuto konkrétní instanci problému příliš hrubá, nedokáže rozlišit „dobré“ a „špatné“ konfigurace. Je tedy třeba ji nějakým způsobem zjemnit.

Budeme se nyní věnovat detekci a řešení druhého případu. Klíčové jsou tedy dva kroky.

1. Identifikace „falešnosti“ protipříkladu:
   Situace je tedy taková, že pro nějaké $k \in \mathbb{N}$ máme $L(\tau_\alpha^k(M)) \cap Bad = Spurious \neq \emptyset$. Protipříklad je zřejmě falešný právě tehdy, když není dosažitelný z počáteční konfigurace. Zpustíme zpětný *přesný* výpočet, který bude inicializován množinou *Spurious*. Pokud se dostaneme k neprázdnému průniku s počáteční konfigurací, protipříklad byl pravý. Pokud se ale jako výsledek některého mezikroku dostaneme prázdnou množinu, protipříklad je falešný, vznikl nepřesností aproximace.

2. Zjemnění abstrakce tak, aby byl z výpočtu falešný protipříklad vyloučen:

   Definice 3.4 *Abstrakční funkci α' nazveme* zjemnění *abstrakční funkce α právě když pro ni platí, že $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Dále nazveme α'* skutečným zjemněním*, právě když dává menší nadaproximaci než α, formálně $\exists M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subset L(\alpha(M))$.*

   Falešný protipříklad reprezentovaný *Spurious* může být eliminován tak, že k α vytvoříme takové skutečné zjemnění α' že pro každý automat M bude platit, že $L(\alpha'(M))$ bude mít s *Spurious* prázdný průnik. Tím se *Spurious* spolehlivě vyloučí z výpočtu.

   Méně restriktivní zjemnění α'' může může pouze zaručit, že $L((\tau_\alpha{}''(\tau_\alpha^{k-1}(M))) \cap Spurious = \emptyset$. Výhodou zde je, že získaná abstrakce α'' je hrubší než α', a pevný bod výpočtu tedy může být dosažen dříve. Nevýhodou je, že hrubší abstrakce může snadněji vést k nalezení dalších sporných protipříkladů.

   Nyní spíše stručně proberu některé konkrétní metody abstrakce. Jejich základním principem je výpočet určité ekvivalenční relace na stavech automatu popisujícího regulární množinu a následné sloučení ekvivalentních stavů. Tokovým postupem je zřejmě zaručeno, že jazyk výsledného automatu bude vždy zahrnovat i jazyk automatu abstrahovaného. Je třeba ještě zdůraznit, že se jedná v podstatě o heuristiky. Úspěšnost nebo vhodnost konkrétní abstrakce lze měřit pouze testováním na konkrétních problémech. Jedním ze směrů, kterém se může ubírat další vývoj může být právě hledání abstrakcí specializovaných na určité typy problémů.

Nyní ke konkrétním metodám abstrakce [BHV04]:

- První podskupinou jsou abstrakce založené na takzvaných predikátových jazycích . Jde o skupinu (množinu) predikátových automatů P. Stavy automatu zde prohlásíme za ekvivalentní, pokud jejich dopředné, případně zpětné jazyky mají průniky se stejnými stavy automaty množiny P.

- Druhou podskupinou jsou abstrakce založené na pojmech jazyk a jazyk stop stavu do určité délky. Dva stavy automatu prohlásíme za ekvivalentní, pokud jejich dopředné/zpětné jazyky/jazyky stop jsou stejné.

3.2 Praktické výsledky

Abstraktní regulární model checking byl implementován a otestován na řadě prakticky užitečných instancích. Jmenujme některé:

- Systémy s frontami
  Testováno na protokolu alternujícího bitu (ABP), kde bylo úspěšně verifikováno doručení zaslané zprávy.

- Systémy se zásobníky
  Byl úspěšně verifiková příklad jednoduchého systému se zásobníkem s rekurzivními procedurami.

- Dynamické seznamy
  Zde byla úspěšně verifikována například procedura pro reverzi seznamu.

- Dále byly verifikovány například některé vlastnosti Petriho sítí a systémů s neohraničenými zásobníky.

Jde bezesporu o kvalitní metodu, problém však je v tom, že vyjadřovací síla konečných automatů není dostatečná a abstraktní regulární model checking tak není dost obecnou metodou.

3.3 Živost

Jak bylo řečeno, regulární a abstraktní regulární model checking jsou metody primárně vyvinuté za účelem verifikování vlastností typu dosažitelnost. To ale neznamená, že jsou vlastnosti typu živost naprosto mimo jejich dosah.

Verifikace vlastnosti typu dosažitelnost vyžadovala vypočítat množinu všech dosažitelných konfigurací a tu otestovat na průnik s množinou nežádoucích konfigurací. Pro verifikaci vlastnosti typu živost něco takového zřejmě nestačí. Ve světě konečně stavových systémů je známo, jak se s živostí

úspěšně vypořádat zejména s pomocí Büchiho automatů - automat akceptuje nekonečné slovo, pokud se v něm nějaký koncový stav opakuje nekonečně krát. Tento princip je převoditelný do prostředí regulárního model checkingu [SB05] – je třeba testovat, jestli nějaká konfigurace se musí opakovat nekonečně krát. Tedy jestli z požadované konfigurace specifikované verifikovanou životní vlastností je vždy dosažitelná identická konfigurace. Již tedy nejde o výpočet množiny dosažitelných konfigurací, ale o výpočet relace dosažitelnosti na této množině (ze které konfigurace je dosažitelná která). Samozřejmě je to problém výpočetně mnohem náročnější než výpočet množiny dosažitelných konfigurací, ale stále je možný.

3.4 Akcelerační schémata pro regulární model checking

Zabýval jsem se zde abstrakcí jako primární metodou, která umožňuje regulárnímu model checkingovému algoritmu terminovat. Pro abstrakci je terminace dokonce jistá a dokazatelná. Není ale jedinou metodou, je možné akcelerovat výpočet i jinak. Uvedu nyní několik přístupů:

- Nedříve schémata uvedená v [PS00], založená na výpočtu meta-přechodu, který pokrývá efekt libovolného množství přechodů základních. Tento přístup je možno aplikovat ve třech variantách:

  1. Lokální akcelerace umožňující libovolné množství přechodů procesu zahrnout do jednoho,
  2. Globální akcelerace umožňující přechody libovolného množství procesů zahrnout do jednoho,
  3. Globální akcelerace binárního přechodu. Zde je uvažován přechod synchronizující dva „sousední“ procesy (např. předání tokenu). Meta přechod umožňuje libovolné množství takových následných přechodů zahrnout do jednoho kroku (příkladrm je cesta tokenu přes libovolné množství procesů).

- Quotienting [BJNT00] – v podstatě lokální abstrakce aplikovaná na konečný transducer. Je hledán transducer odpovídající efektu libovolného množství aplikace transduceru původního.

- Extrapolace [BJNT00] – Při výpočtu pevného bodu se, kdy narůstá množina dosažitelných konfigurací (v případě regulárního model checkingu, kdy transduer obsahuje identitu, je jazyk automatů dosažitelných konfigurací z každým krokem obecnější) jsou porovnávány některé

prvky posloupnosti mezivýpočtu, jsou hledány opakující se vzory v přírůstku a na základě toho je pak odhadován výsledek dopředu.

Tyto techniky jsou vesměs přesné nebo přesnější než abstrakce v tom smyslu, že vypočítaná množina dosažitelných konfigurací je skutečnou množinou dosažitelných konfigurací a ne „pouhou“ aproximací. Tato výhoda je však vykoupena drastickým nárůstem prostředků potřebných k dokončení výpočtu, jehož terminace v mnoha případech ani není zaručena.

3.5 Abstraktní regulární stromový model checking

Abstraktní regulární model checking dovoluje verifikovat vlastnosti lineárních dynamických datových struktur typu seznam. Odpovídá to typu objektů – slov, jejichž množiny jsou kódovatelné konečným automatem. Pokud ale hledáme nástroj, kterém by bylo možno úspěšně verifikovat vlastnosti programů manipulujících dynamické paměťové struktury co nejobecněji, konečné automaty trpí příliš omezenou vyjadřovací schopností. Množinu v praxi běžných struktur typu strom mechanismus konečných automatů není schopen přímočaře kódovat.

Právě strom a stromová kostra je spojujícím atributem velké škály v praxi používaných struktur. Kořen stromu je analogií proměnné odkazující na celou strukturu v paměti. Pokud navíc umožníme specifikovat dodatečné pointery na základě stromové kostry, získáme mechanismus kódování grafů velmi vhodný pro kódování konfigurací paměti vznikajících při běhu pointerových programů.

V tomto kontextu se nabízí poměrně přímočaré zobecnění konečného automatu – konečný stromový automat. Jedná se o automat, které místo regulárních množin slov kóduje regulární množiny termů, respektive stromů těmto termům příslušných. Podobně i zobecnění konečného transduceru na konečný stromový transducer je na snadě. Podobné úvahy zřejmě vyústili ve vznik Abstraktního regulárního model checkingu, který je obdobou regulárního model checkingu ale místo konečných automatů využívá konečné automaty stromové a místo konečných transducerů využívá konečné stromové transducery.

Budu se věnovat přímo regulárnímu stromovému model checkingu abstraktnímu, přesto že existují i jeho neabstraktní větve. Pro ty ale ještě více, než v případě regulárního model checkingu a abstraktního regulárního model checkingu platí, že přesnost zjištěných množin dosažitelných konfigurací je příliš draze vykoupena výpočetní náročností a nejistoty terminování výpočtu. Jmenuji ale alespoň něco málo prací, které se neabstraktní větví zabývají [**?**].

3.5.1 Stromové konečné automaty a transducery

Začnu opět základními definicemi konečných stromových automatů a transducerů. Detailní přehled definic a základních teoretických výsledků ohledně tohoto tématu může být nalezen v [CDG+05].

Definice 3.5 Abeceda Σ *je konečnou množinou symbolů.* Σ *se nazývá* ohodnocená, *pokud existuje* ohodnocovací *funkce* $\# : \Sigma \to \mathbb{N}$. *Pro každé* $k \in \mathbb{N}$, $\Sigma_k \subseteq \Sigma$ *je množina symbolů s hodnotou* k. *Symboly ze* Σ_0 *se nazývají* konstanty. *Nechť* χ *je spočetná množina symbolů nazývaných* proměnné. $T_\Sigma[\chi]$ *značí množinu* termů *nad* Σ *a* χ. *Množina* $T_\Sigma[\emptyset]$ *je značena jako* T_Σ *a její elementy jsou nazývány* základní termy. *Term* t *ze* $T_\Sigma[\chi]$ *je nazýván* lineární *pokud se v něm každá proměnná vyskytuje maximálně jednou. Termy z* $T_\Sigma[\chi]$ *mohou být viděny jako stromy – listy jsou označeny konstantami a proměnnými, každý uzel s* k *syny je označen symbolem z* Σ_k.

A Stromový automat *nad ohodnocenou abecedou* Σ *je čtveřice* $A = (Q, \Sigma, F, \delta)$, *kde:*

- Q *je konečná množina stavů*
- $F \subseteq Q$ *je množina koncových stavů*
- δ *množina přechodů následujících typů: (i)* $f(q_1, \ldots, q_n) \to_\delta q$, *(ii)* $a \to_\delta q$, *a (iii)* $q \to_\delta q'$ *kde* $a \in \Sigma_0$, $f \in \Sigma_n$, *a* $q, q', q_1, \ldots, q_n \in Q$.

Nechť je t *základní term. Běh stromového automatu* A *na* t *je definován následovně. Zaprvé, listy budou označeny stavy. Pokud je list označen symbolem* $a \in \Sigma_0$ *a existuje pravidlo* $a \to_\delta q \in \delta$, *list bude označen stavem* q. *Vnitřní uzel* $f \in \Sigma_k$ *bude označen stavem* q *pokud existuje pravidlo* $f(q_1, q_2, \ldots, q_k) \to_\delta q \in \delta$ *a první syn uzlu je označen stavem* q_1, *druhý* q_2, ..., *a poslední* q_k. *Pravidla typu* $q \to_\delta q'$ *se nazývají* ε-kroky *a umožňují zjemnit návěští uzlu z* q *na* q'. *Pokud je kořenový symbol označen koncovým stavem z* F, *je term* t *akceptován automatem* A.

Množina termů akceptovaných stromovým automatem A *se nazývá* regulární stromový jazyk *a je značena* $L(A)$.

Definice 3.6 *A* Stromový transducer *je pětice* $\tau = (Q, \Sigma, \Sigma', F, \delta)$ *kde:*

- Q *je konečná množina stavů*
- $F \subseteq Q$ *je množina koncových stavů*
- Σ *je vstupní ohodnocená abeceda*

- Σ' *je výstupní ohodnocená abeceda*
- δ *je množina přechodových pravidel následujících typů: (i)* $f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow_\delta q(u)$, $u \in T_{\Sigma'}[\{x_1, \ldots, x_n\}]$, *(ii)* $q(x) \rightarrow_\delta q'(u)$, $u \in T_{\Sigma'}[\{x\}]$, *a (iii)* $a \rightarrow_\delta q(u)$, $u \in T_{\Sigma'}$ *kde* $a \in \Sigma_0$, $f \in \Sigma_n$, $x, x_1, \ldots, x_n \in \chi$ *a* $q, q', q_1, \ldots, q_n \in Q$.

Běh stromového transduceru τ *na základním termu* t *je podobný, jako v případě stromového automatu. Nejdříve jsou použita pravidla typu* (iii). *Pokud je list označen symbolem* a *a existuje pravidlo* $a \rightarrow_\delta q(u) \in \delta$, *je list nahrazen termem* u *a označen stavem* q. *Pokud je uzel označen symbolem* f *a existuje pravidlo* $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow_\delta q(u) \in \delta$, *první podstrom uzlu má stav* q_1, *druhý* q_2, *...*, *a poslední* q_n, *potom symbol* f *a všechny podstromy uzlu jsou nahrazeny podle pravé strany pravidla proměnnými* $x_1, \ldots, x_n$ *substituovanými za podstromy na levé straně. Označení stavem* q *je přiřazeno novému stromu. Pravidla typu* (ii) *se nazývají* ε-kroky. *Umožňují nahradit* q*-označený strom podle pravé strany pravidla a přiřadit označení* q'. *Běh transduceru je úspěšný, pokud je kořen označen koncovým stavem.*

Stromový transducer je lineární, *pokud jsou všechny pravé strany jeho pravidel lineární (každá proměnná se vyskytuje maximálně jednou).Transducer je* strukturu zachovávající, *pokud nemodifikuje tvar vstupního stromu a pouze mění návěští uzlů. Identifikujeme transducer* τ *s relací* $\{(t, t') \in T_\Sigma \times T_\Sigma \mid t \rightarrow_\delta^* q(t')$ *pro nějaký* $q \in F\}$. *Pro množinu* $L \subseteq T_\Sigma$ *a relaci* $\varrho \subseteq T_\Sigma \times T_\Sigma$, *značíme* $\varrho(L)$ *množinu* $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in \varrho\}$ *a* $\varrho^{-1}(L)$ *množinu* $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in \varrho\}$.

Nechť je $\iota \subseteq T_\Sigma \times T_\Sigma$ *identická relace a* $\circ$ *složení relací. Definujeme rekurzivně relace* $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$ *and* $\tau^* = \cup_{i=0}^{\infty} \tau^i$.

Platí že třída lineárních transducerů je uzavřena na sjednocení a že pokud je τ lineární transducer a L je regulární stromový jazyk, potom množiny $\tau(L)$ a $\tau^{-1}(L)$ jsou efektivně konstruovatelné [CDG+05].

3.5.2 Vlastní ARTMC

Potřebné pojmy jsou zavedeny naprosto analogickým způsobem, jako u klasického abstraktního regulárního model checkingu. Liší se jen struktury za nimi. Základní schéma ARTMC je naprostou analogií ARMC, abstrakce, abstrakční schéma a zjemění abstrakce se definují také stejně.

Používaná abstrkční schémata [BHRV05, BHRV06a] jsou také zobecněním abstrakčních schémat z ARMC. Opět jsou založena na výpočtu vhodné relace ekvivalence na stavech automatu a následném ztotožnění ekvivalentních stavů. Konkrétně se jedná o:

- zobecněnou predikátovou abstrakci, kde je ale množina P tvořena stromovými automaty,
- abstrkci postavenou na relaci ekvivalence, která ztotožňuje dva stavy, právě když jejich jazyka do určité délky jsou stejné

Ostatní akcelerační metody jsou také analogií ARMC metod. Konkrétně:

- Globální akcelerace unárních přechodů [SP02]
- Extrapolace [BT02]
- Quotienting [AJMd02]

3.5.3 Komplexní datové struktury a ARTMC

Za zmínku stojí metoda kódování komplexních datových struktur [BHRV06b]. Grafy paměti jsou zde totiž kódovány tak, že na stromové kostře grafu se zavedou takzvané směrovací výrazy, které kódují umístění dodatečných nekosterních pointerů formou specifikace cesty kostrou. Jedná se o přístup v základu v podstatě stejný, jaký je použit v nástroji PALE (zmíněném v přehledu alternativních přístupů), ale na rozdíl od PALE je zde metoda plně automatická a nevyžaduje od uživatele nic jako zadavání invariantů cyklů v PALE.

3.5.4 Experimantální výsledky

ARTMC byl vyzkoušen z úspěchem na řadě praktických příkladů. Jmenujme alespoň:

- Protokol dvoucestného předávání tokenu – token je v síti stromového tvaru předáván od listů ke kořenu a naopak.
- Percolate protokol – síť procesů ve tvaru stromu počítá distribuovaně hodnotu booleovské funkce.
- Stromový soudce – jedná se o implementaci vzájemné výlučnosti procesů pomocí stromové sítě
- Volba šéfa – klasický protokol volby šéfa v distribuovaném prostředí stromové sítě

3.5.5 Nedeterminismus a ARTMC

Doposud získané výsledky z v podstatě všemi formami stromového model checkingu byly postaveny na deterministických verzích tohoto přístupu. Algoritmy implementující základní operace pro nedeterministické automaty mají totiž nezřídka exponenciální složitost. Nedeterminismus má však mnohem silnější vyjadřovací schopnosti ve smyslu cena/výkon, nebo lépe velikost automatu/informace. V poslední době se objevila možnost použít velice kvalitní heuristiky, které do značné míry řeší problémy s exponenciální časovou složitostí nedeterministických automatových operací. Jedná se o zobecnění heuristických algoritmů pro klasické nedeterministické konečné automaty:

- Algoritmus rozhodující univerzalitu jazyka automatu, publikovaný v [WDHR06] a z něj odvoditelný algoritmus rozhodující inkluzi a ekvivalenci dvou automatů. Algoritmus je postavený na iterativním výpočtu protiřetězce množin dosažitelných stavů. Pomocí něj je buď nalezen protipříklad k univerzalitě automatu, nebo je dosažen pevný bod a automat je prohlášen za univerzální.

- Algoritmus částečné minimalizace nedeterministického automatu výpočtem bisimulační ekvivalence na jeho stavech [AKH06]. Bisimulační ekvivalence je totiž jemnější než relace jazykové ekvivalence, počítaná klasickým minimalizačním algoritmem a mnohem lehčeji vypočítatelná – s časovou složitostí, která se snadno vejde do kvadratické. Zde se ovšem objevuje problém s tím, že ten článek je špatně.

Pokud by se povedlo (právě se snažíme) všechny tyto techniky implementovat a začlenit do ARTMC frameworku, mohlo by to vést ke značnému zrychlení abstraktního stromového regulárního model checkingu a k rozšíření jeho možností.

Abstraktní regulární model checking je úspěšnou metodou. Několik týdnů se možná dal označit za nejúspěšnější. Nyní je pravděpodobně jedním z mála přístupů, jehož výsledky snesou srovnání s výsledky verifikace pomocí separační logiky, které se stručně budu věnovat následující sekci.

4 Separační logika

Nedávno se na poli softwarové verifikace zvedl povyk, výskalo se nadšením ale mnohé oči také zaplakaly, protože publikováním [NDQC06] se stalo, že tým, zabývající se aplikací separační logiky na model checking pointerových programů, kompletně všem vypálil rybník. Tokový je alespoň první dojem z

tohoto článku, druhý pohled odhaluje, že metoda přece jen není plně automatická. Přesto ale tato práce znamená v oboru značný posun. Tvrdí se zde, že s pomocí separační logiky lze verifikovat neuvěřitelně obecnou třídu paměťových struktur spolu s množstvím kvantitativních vlastností nad těmito strukturami, a to ve velice krátkých časech. Mé osobní studium této oblasti je zatím v začátcích, ale mělo by urychleně pokračovat. Zatím jsem schopen zde zhruba nastínit základní myšlenky.

4.1 Základní myšlenka separační logiky a její aplikace na verifikaci

Vstupní branou do světa verifikace pomocí separační logiky je práce [Rey02]. Separační logika je v podstatě rozšířenou logikou výrokovou. Hlavní změnou je zde zavedení nové logické spojky $*$ – separační konjunkce. Modelem separační logiky je halda – zjednodušeně soubor paměťových „skladů“ spolu s relací bezprostřední dosažitelnosti přes určitý pointer. Pracuje se zde s tranzitivním uzávěrem této relace a s množstvím dalších pomocných pojmů.

Jaký je tedy význam a přínos separační konjunkce? Mějme tedy haldu a pro ni platnou formuli $f = a * b$. Formule zhruba říká, že halda je sjednocením dvou *disjunktních* podhald, kde v jedné platí formule a v druhé formule b. Také jinak, že platnost a v první podhaldě nijak nesouvisí s druhou podhaldou a podobně platnost b v druhé podhaldě nijak nesouvisí s podhaldou první. Tato vlastnost určtým způsobem odpovídá charakteru operací pointerového programu. Změna určité části paměti je často plně popsatelná z pohledu jen malé části haldy. Laicky bychom prostě řekli, že jisté místo se směnilo a zbytek jednoduše zůstal stejný. Ve světě verifikace to znamená, že zanesení efektu programové akce měnící paměťovou strukturu do kódování této struktury nevyžaduje znovuzpracování celého kódování struktury, ale jen malé omezené části. Tato vlastnost je zřejmě oním klíčem k úspěchu separační logiky, neboť právě rozsáhlost struktur a vyplývající náročnost jejich změn reflektujících chování programu je základním úskalím verifikace pointerových programů.

4.2 Některé experimentální výsledky

Jedná se o výsledky uveřejněné v [NDQC06]. Byly zde verifikovány programy pracující se seznamy, obousměrně zřetězenými seznamy, cyklickými seznamy, stromy, vyhledávacími stromy, AVL stromy, černobýlími stromy. Verifikovány byly operace jako přidání, smazání prvku, spojení seznamů, třídící algoritmy, reverze seznamu, a to jednak na zachování invariantu struktury, tak na korektnost operace. Časy se pohybují v řádu sekund, maximálně minut.

5 Závěr

V této práci jsem se pokusil shrnout mé studium regulárního model chekingu a model checkingu pomocí separační logiky, spolu se základním souvisejícím okolím. Regulárnímu model checkingu jsem se věnoval podrobně, jelikož mé znalosti jsou v důsledku zaměření mé nynější práce největší v této oblasti. Kapitola o separační logice příliš rozpracována není, a to ze dvou důvodú. Jednak mé znalosti z této problematiky nejsou ještě kompletní a jednak rozsah práce okamžiku psaní nadpisu Separační logika již odpovídal požadavkům, takže jsem velké rozvádění této sekce nepovažoval za nutné.

Reference

[AJ96] P. Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. *Information and Computation*, 127(2):91–101, 1996.

[AJMd02] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[AKH06] Parosh Aziz Abdulla, Lisa Kaati, and Johanna Högberg. Bisimulation minimization of tree automata. In *CIAA*, pages 173–185, 2006.

[BFL06] S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proc. of AVIS'06*, 2006.

[BHRV05] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. In *Proc. of Infinity'05*, 2005.

[BHRV06a] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.

[BHRV06b] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[BIL03] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[BP96] B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[BPZ05] I. Balaban, A. Pnueli, and L.D. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.

[BT02] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[CDG+05] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005.
URL: `http://www.grappa.univ-lille3.fr/tata`.

[DN03] D. Dams and K.S. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*. Springer, 2003.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[Kle87] S. Kleene. *Introduction to Mathematics*. North-Holland, 1987.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[KS93] N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.

[LQ06] S.K. Lahiri and S. Qadeer. Verifying Properties of Well-Founded Linked Lists. In *Proc. of POPL'06*. ACM Press, 2006.

[MS01] A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. Also in SIGPLAN Notices 36(5), 2001.

[NDQC06] H.H. Nguyen, C. David, S. Qin, and W. Chin. Automated verification of shape, size and bag properties via separation logic, 2006.

[PS00] A. Pnueli and E. Shahar. Liveness and Acceleration in Parameterized Verification. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[SB05] V. Schuppan and A. Biere. Liveness Checking as Safety Checking for Infinite State Spaces. In *Proc. of Infinity'05*, 2005.

[SP02] E. Shahar and A. Pnueli. Acceleration in Verification of Parameterized Tree Networks. Technical Report MCS02-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002.

[SRW02] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.

[Ven04] A. Venet. A Scalable Nonuniform Pointer Analysis for Embedded Programs. In *Proc. of SAS'04*, volume 3148 of *LNCS*. Springer, 2004.

[WDHR06] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, pages 17–30, 2006.