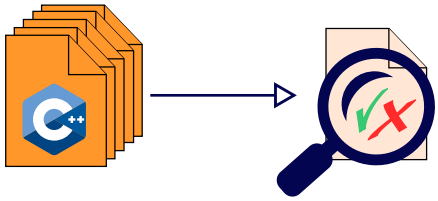
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Dynamická analýza parametrických kontraktů pro paralelismus

Monika Mužikovská*



Abstrakt

V paralelních programech a zejména ve velkých projektech složených z několika modulů může docházet k porušení atomicity. Takové chyby je často těžké odhalit, protože neexistuje přesný popis, kdy k nim dochází. Jednou z možností, jak požadavky na atomicitu specifikovat, jsou kontrakty pro paralelismus. Díky tomuto protokolu je možná tvorba automatizovaných nástrojů, které porušení atomicity odhalují. Jedna z metod byla implementována jako dynamický analyzátor v nástroji ANaConDA. Protože analýza může odhalit řadu problémů, u kterých vývojář díky kontextu ví, že se nejedná o chyby, je možné specifikaci kontraktů rozšířit o parametry. Cílem této práce bylo navrhnout a implementovat algoritmus pro analýzu parametrických kontraktů jako rozšíření nástroje ANaConDA. Výsledkem je nový analyzátor Param-contract-validator, jehož použití a výsledky budou demonstrovány na jednoduchých příkladech i programech obsahujících porušení atomicity.

Klíčová slova: ANaConDA — Dynamická analýza — Kontrakty pro paralelismus — Parametrická analýza kontraktů

Přiložené materiály: *N/A*

*xmuzik05@stud.fit.vutbr.cz, *Fakulta informačních technologií, Vysoké učení technické v Brně*

1. Úvod

Paralelní programy s sebou kromě výhod v podobě vyššího výkonu přinesly také specifické paralelní chyby. Jejich odhalování může být velmi složité, zvlášť když pro některé z nich ani neexistuje popis situací, za kterých vznikají. Jedním z takových obecných problémů je porušení atomicity. Článek Verifying Concurrent Programs Using Contracts [1] specifikuje kontrakt pro paralelismus, což je protokol, pomocí kterého lze popsat, které metody veřejného rozhraní nad stejným objektem musí být vykonány atomicky. Zároveň nabízí metody pro statickou i dynamickou analýzu kontraktů, z nichž druhá jmenovaná byla implementována jako analyzátor Contract-validator pro framework ANaConDA.

Výsledkem dynamické analýzy může být velké množství hlášení o porušení kontraktu, přestože se ve skutečnosti při zvážení kontextu o chybu nejedná. Informace o konkrétních datech, se kterými metody operují, lze do specifikace kontraktů zanést pomocí parametrů. Výsledky analýzy potom budou přesnější.

Tento článek popisuje základní principy dynamické analýzy kontraktů a navrhuje metodu pro analýzu s podporou parametrů. Navržený algoritmus byl implementován jako nový analyzátor pro nástroj ANaConDA. Experimenty na sadě programů ukázaly, že za cenu mírného zvýšení časové a paměťové náročnosti je možné zpřesnit výsledky analýzy až o desítky chybových hlášení, což vývojářům umožní soustředit se na opravu reálných a závažných chyb.

2. Kontrakty pro paralelismus

Kontrakt pro paralelismus (dále zjednodušeně kontrakt) je protokol, který vyjadřuje, které metody z veřejného rozhraní modulu je nutné vykonat atomicky, pokud operují nad stejným objektem.

Formálně, je-li $\Sigma_{\mathbb{M}}$ množina názvů všech veřejných metod modulu, pak kontrakt je definován jako množina klauzulí $\mathbb{R}$, přičemž každá klauzule $\rho \in \mathbb{R}$ je regulárním výrazem nad $\Sigma_{\mathbb{M}}$. K porušení kontraktu dojde tehdy, kdy je libovolná sekvence v programu, pro níž existuje reprezentace klauzulí $\rho \in \mathbb{R}$, proložena voláním libovolné metody z množiny $\Sigma_{\mathbb{M}}$ nad stejným objektem.

Pro demonstraci použití kontraktů můžeme uvažovat implementaci kontejneru, jehož API je tvořeno metodami:

- `get(index)` – vrací data na daném indexu,
- `set(index, data)` – uloží data na daný index,
- `rmv(index)` – odstraní element na daném indexu a zároveň všechny elementy na vyšších indexech posune,
- `idx(data)` – vrací index, na kterém se nachází hledaná data,
- `size()` – vrací velikost kontejneru (počet aktuálně uložených elementů) a
- `contains(data)` – vrací informaci o existenci elementu v kontejneru.

Kontrakt, který by zaručoval korektní chování kontejneru v paralelním programu, by mohl vypadat takto:

$$
\begin{array}{ll}
(\rho_1) & \texttt{idx (get|set|rmv)} \\
(\rho_2) & \texttt{size (get|set|rmv)} \\
(\rho_3) & \texttt{contains idx}
\end{array}
$$

Dle klauzule ρ_1 je nutné, aby sekvence metod pro obdržení indexu pomocí `idx` a pro modifikaci dat pomocí `get, set` nebo `rmv` proběhla atomicky. Pokud by jiné vlákno volalo metodu z množiny $\Sigma_{\mathbb{M}}$ nad stejným objektem, mohlo by dojít ke změně v kontejneru a získaný index by už nemusel být platný. Podobné chování vynucuje i klauzule ρ_2, zde by při vymazání elementu jiným vláknem mohlo dojít k přístupu za hranici platných dat, přestože metodě `get, set` nebo `rmv` předcházelo ověření velikosti kontejneru. Klauzule ρ_3 vynucuje atomicitu operací pro ověření existence dat a následné zjištění jejich indexu. Pokud by došlo ke změně nebo vymazání dat po tom, co byla potvrzena jejich existence, metoda `idx` by hledala neexistující data.

2.1 Cíl a spoiler

Z příkladu výše je patrné, že ne všechny metody z abecedy $\Sigma_{\mathbb{M}}$ mohou způsobit změny v kontejneru a vést k chybě. Například volání metody `size` by na základě uvedených informací způsobilo porušení kontraktu, přestože se nejedná o operaci, která by mohla ovlivnit konzistenci dat. Proto je možné specifikaci kontraktu rozšířit o tzv. cíl (angl. *target*) a spoiler (angl. originál zde nepřekládáme).

Klauzule kontraktu, tak jak byly definovány doteď, se nazývají cíle a budou uváděny ve dvojicích spolu se spoilery. Formálně, $\mathbb{R}$ je množina cílů a $\mathbb{S}$ je množina spoilerů, přičemž každý cíl $\rho \in \mathbb{R}$ a spoiler $\sigma \in \mathbb{S}$ je regulárním výrazem nad množinou názvů všech veřejných metod modulu $\Sigma_{\mathbb{M}}$. Kontrakt je definován jako relace $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$.

Spoiler reprezentuje sekvenci metod, které mohou cíl porušit. Není tedy nutné, aby sekvence cíle proběhla atomicky vzhledem ke všem metodám veřejného rozhraní modulu, ale její atomicita nesmí být porušena sekvencí spoileru (opět platí, že všechny metody musí být vykonány nad stejným objektem). K porušení také dojde pouze tehdy, když exekuce spoileru plně prokládá exekuci cíle – tedy začne později a skončí dříve. Bude-li potřeba detekovat i částečné proložení, je na místě rozdělit sekvenci do několika spoilerů.

S využitím rozšíření definice kontraktů o spoilery by mohly jednotlivé klauzule z předchozího příkladu vypadat takto:

$$
\begin{array}{ll}
(\rho'_1) & \texttt{idx (get|set|rmv)} \leftarrow \texttt{set|rmv} \\
(\rho'_2) & \texttt{size (get|set|rmv)} \leftarrow \texttt{rmv} \\
(\rho'_3) & \texttt{contains idx} \leftarrow \texttt{set|rmv}
\end{array}
$$

2.2 Parametry

Definice kontraktů lze zpřesnit také pomocí parametrů, které dodají informaci o toku dat. Například klauzule `idx (get|set|rmv)` vyžaduje atomicitu pro dané metody vždy, když pracují nad stejným objektem. Přesto má smysl ji vyžadovat pouze v případě, kdy metody `get`, `set` a `rmv` budou mít jako argument index získaný metodou `idx`.

Tyto požadavky lze do kontraktů zavést pomocí proměnných. Lze jimi označit návratové hodnoty či hodnoty parametrů jednotlivých metod a následně vyžadovat, aby se rovnaly všechny hodnoty označené stejnou proměnnou. Také je možné využít podtržítko ve smyslu anonymní proměnné, pro označení parametru jehož hodnota není pro kontrakt důležitá.

Specifikaci kontraktů z předchozího příkladu lze parametry doplnit takto:

(ρ_1'') `X=idx(_)  (get(X)|set(X,_)|rmv(X)) ← set(X,_)|rmv(_)`
(ρ_2'') `X=size()  (get(X)|set(X,_)|rmv(X)) ← rmv(_)`
(ρ_3'') `contains(X)  idx(X) ← set(_,_)|rmv(_)`

Pro klauzuli ρ_1'' je spoiler omezen pouze na metody `set` a `rmv`, jelikož ty mohou způsobit, že index dat získaný metodou `idx` nemusí být po jejich vykonání nadále platný. Chyba nastane pouze v případě, kdy metoda `set` upraví data na indexu získaném metodou `idx`, což je specifikováno pomocí parametru X. Metoda `rmv` posune veškerá data v kontejneru, a proto k chybě dojde i při jejím vykonání nad jiným indexem. V klauzuli ρ_2'' je vyžadováno, aby po metodě `size` nebyla v jiném vlákně nad stejným objektem volána metoda `rmv`, jelikož by došlo k přístupu za hranici platných dat. Zde se použití parametru řídí stejnou logikou jako v předchozí klauzuli. Klazule ρ_3'' vyžaduje, aby mezi potvrzením existence dat a následným hledáním jejich indexu nebyla nad kontejnerem provedena metoda `set` nebo `rmv`, jelikož by metoda `idx` mohla hledat data, která už v kontejneru neexistují. Pomocí parametru X je specifikováno, že k chybě dojde pouze pokud budou metody `contains` a `idx` pracovat nad stejnými daty.

2.3 Princip dynamické analýzy kontraktů bez parametrů

Pro dynamickou analýzu kontraktů byla v [1] navržena metoda, která podporuje specifikaci kontraktů rozšířených o cíl a spoiler, ale nezahrnuje analýzu parametrů. Algoritmus pro analýzu kontraktů s parametry z něj ovšem vychází a rozšiřuje ho, a proto zde bude alespoň zjednodušeně popsán základní princip.

Analýza vícevláknového programu probíhá za jeho běhu, tudíž je nutné si v paměti udržovat potřebné informace, které mohou být později důležité pro rozhodnutí, zda došlo k porušení kontraktu či nikoli. Z tohoto důvodu jsou pro analýzu klíčové instance cíle a spoileru a tzv. *happens-before* relace (dále v textu značena jako *hb*-relace). Události, které v monitorovaném programu nastaly, se uchovávají v tzv. okně stopy (z angl. *trace window*).

2.3.1 Instance cíle a spoileru

Instance je posloupnost událostí ve stopě programu. Jednotlivé události představují volání metod a jejich názvy odpovídají regulárnímu výrazu cíle (resp. spoileru). Všechny události musí být vykonány stejným vláknem a nesmí být proloženy voláním jiné metody z abecedy cíle (resp. spoileru). Instance cíle ρ se značí r a instance spoileru σ se značí s.

Uvažujme cíl $\rho = abc$, spoiler $\sigma = def$ a stopu $\tau = aabdcdeef$. Události na indexech 1, 3 a 5 nemohou tvořit instanci cíle ρ, protože jsou proloženy událostí 2, která je z abecedy Σ_ρ. Na indexech 2, 3 a 5 se nachází instance r, protože událost 4 do zmíněné abecedy nepatří. Ve stopě ale nelze nalézt žádnou posloupnost událostí def, která by nebyla narušena voláním metody z abecedy Σ_σ, a proto se v ní nenachází žádná instance spoileru σ. Události 4, 7 (resp. 8) a 9 jsou narušeny událostí 6, která je z abecedy spoileru. Události 6, 7 (resp. 8) a 9 jsou zase v každém případě proloženy událostí e, taktéž z abecedy Σ_σ.

Množina všech instancí cíle ve stopě τ se značí jako $[\rho]^\tau$ a množina všech instancí spoileru ve stopě τ jako $[\sigma]^\tau$.

2.3.2 *hb*-relace a vektorové hodiny

Dynamická analýza dokáže vyhodnotit pouze jeden běh programu, takže se pro získání co největšího množství informací využívá extrapolace. Jedná se o techniku, kdy se na základě dostupných dat vyvozují závěry o situacích, které v aktuálním plánu přepínání kontextu nezpůsobily chybu, ale v jiném podobném by mohly. U analýzy kontraktů se tedy sleduje, zda by mohlo dojít k přepnutí kontextu tak, aby došlo k porušení kontraktu. Za tímto účelem se využívá *hb*-relace a její implementace v podobě vektorových hodin. Nejedná se o metodu specifickou pro analýzu kontraktů, ale využívá se i v jiných dynamických analyzátorech, například v algoritmu FastTrack pro detekci souběhu nad proměnnou [2], a její definice vychází z [3] již z roku 1978.

hb-relace $\prec_{hb}$ určuje uspořádání dvou událostí s ohledem na teoreticky dosažitelné plánování přepnutí kontextu na základě vybraných synchronizačních primitiv. Formálně je *hb* relace $\prec_{hb}$ nad stopou programu τ definována jako nejmenší tranzitivně uzavřená relace nad množinou událostí $\{e_1, \cdots, e_n\}$ stopy τ. Dvě události e_j a e_k jsou v *hb* relaci $e_j \prec_{hb} e_k$ vždy, když platí, že $j < k$ a zároveň je splněna alespoň jedna z následujících podmínek:

- Obě události e_j a e_k byly vykonány stejným vláknem (pořadí v programu).
- Obě události e_j a e_k získávají nebo uvolňují stejný zámek.
- Jedna z událostí e_j a e_k je operace *fork* nebo operace *join* vlákna u ve vláknu t a druhá událost je provedena ve vláknu u.

Pokud mezi dvěma událostmi neexistuje *hb* relace, potom jsou považovány za souběžné a mohou být zdrojem paralelních chyb.

hb relaci je možné implementovat pomocí tzv. vektorových hodin (z angl. *vector clock*), které udržují informace o logickém čase pro každé vlákno v programu. Při synchronizačních operacích dochází k aktualizaci logického času podle pravidel konkrétní implementace. Pro potřeby kontraktů se v paměti udržují informace o začátku a konci jednotlivých instancí cíle a spoileru a při jejich dokončení se kontrolují vektorové hodiny a rozhoduje se, zda došlo k porušení kontraktu či nikoli.

K porušení kontraktu dojde právě tehdy, když ve stopě τ existuje instance cíle r a spoileru s, pro které platí $start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$.

2.3.3 Algoritmus analýzy bez parametrů

Výsledný algoritmus se skládá ze dvou hlavních kroků. První se vykoná při vstupu do metody a na základě jejího názvu se provádí kroky v konečných automatech popisujících jednotlivé instance cílů a spoilerů. Při výstupu z metody se kontrolují dokončené instance na porušení kontraktu. Při kontrole se využívají vektorové hodiny prvních a posledních událostí těchto instancí.

Součástí algoritmu jsou také pravidla pro odstraňování dokončených instancí z okna stopy, ale pro zjednodušení článku zde nebudou uvedeny. Detaily jsou popsány v [1].

3. Dynamická analýza parametrických kontraktů v nástroji ANaConDA

Algoritmus navržený v této sekci podporuje kontrakty včetně cílů, spoilerů i parametrů. Nejdříve bude zjednodušeně popsán princip analýzy a rozdíly, které s sebou parametry přináší, a poté framework ANaConDA, pro který byla navržená metoda implementována jako nový analyzátor.

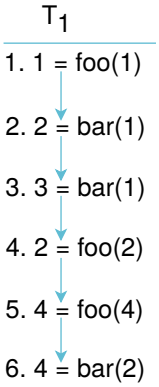
3.1 Omezení na použití parametrů

Hlavním rozdílem a problémem parametrické analýzy je chápání instancí. V předchozím případě byly instance cíle a spoileru tvořeny pouze na základě názvů metod, zde ale zahrnují i hodnoty jednotlivých parametrů. Teoreticky je tedy možné, aby pro jeden cíl či spoiler existovalo nekonečně mnoho probíhajících instancí, což velmi zvyšuje paměťové nároky na analýzu. Také při invalidaci běžících instancí je potřeba zvážit nejen názvy metod, ale i hodnoty parametrů. Na příkladu budou ukázány problémy, které je potřeba při návrhu algoritmu zohlednit.

Původní definice kontraktů nikterak neomezuje použití parametrů a teoreticky je tedy možné analyzovat kontrakt:

```
X=foo(_) Y=bar(X) foo(Y) <- spoiler(X)
```

Na obrázku 1 je zobrazena stopa jednoho vlákna programu a událostí, které vykonalo.



Obrázek 1. Ukázka stopy programu pro demonstraci problematiky instancí při parametrické analýze kontraktů.

První tři události v programu reprezentují tři různé běžící instance cíle. Události 1 a 2 tvoří instanci r_1 s hodnotami parametrů $X = 1$ a $Y = 2$, události 1 a 3 instanci r_2 s hodnotami $X = 1$ a $Y = 3$ a také událost 1 tvoří běžící instanci r_3 pro další hodnoty parametru Y, které by teprve mohly přijít. Událost 4 vytvoří novou instanci cíle r_4 s hodnotou $X = 2$ a zároveň dokončí instanci r_1. Událost 5 je velmi problematická. Jednak dojde k založení instance r_5 s hodnotou $X = 4$, ale potenciálně by také mohla invalidovat instanci vzniklou z r_4, což v tomto okamžiku ještě není rozhodnutelné. Jakmile dojde k události 6, došlo by k vytvoření další instance, ale ta kvůli události 5 není validní.

Z příkladu výše jsou patrné dvě věci – jednak je potřeba držet v paměti velké množství běžících instancí pro případ, že by přišla událost s novou hodnotou parametru, pro který ještě instance neexistuje. Dalším problémem je nutnost pamatovat si i minulá volání funkcí a jejich argumentů, protože v okamžiku vzniku události ještě nemusí být možné rozhodnout, jaký bude mít vliv na běžící instance. Jelikož se jedná o značnou komplikaci algoritmu pro analýzu parametrických kontraktů, který by kvůli hlídání všech teoretických možností mohl být nepoužitelný pro analýzu reálných programů, bylo zavedeno omezení, kdy hodnoty všech parametrů musí být jasně stanoveny při započetí exekuce instance. Hodnoty lze definovat i pomocí výrazů. Nemělo by se jednat o omezení v použití, protože kontrakty stejného typu jako výše uvedený je možné rozdělit na více klauzulí, čímž se vyřeší oba

výše popsané problémy:

```
(ρ1)   X=foo(_) bar(X) <- spoiler(X)
(ρ2)   Y=bar(X) foo(Y) <- spoiler(X)
```

Pro ještě přesnější výsledky analýzy byla do algoritmu i analyzátoru přidána možnost specifikovat omezení na hodnoty parametrů pomocí výrazů, jejichž výsledkem je pravdivostní hodnota. Instance je tedy započata pouze pokud je možné přijmout další událost v dané instanci a zároveň dodržet specifikovaná omezení hodnot proměnných.

3.2 Algoritmus analýzy s parametry

Pro výsledný algoritmus je nutné ještě definovat několik pojmů: nechť $\mathbb{P}_r$ označuje množinu parametrů a jejich hodnot instance r, $next(r)$ označuje název další metody, se kterou je možné provést přechod v konečném automatu instance r, $first(\rho)$ název první metody cíle ρ a výsledkem funkce $constraints(A)$ je pravdivostní hodnota udávající, zda parametry v množině A splňují daná omezení.

Algoritmus 1 Část algoritmu pro parametrickou analýzu kontraktů.

Data: okno v, metoda m s názvem n_m a argumenty A vygenerována vláknem t

1: **for** $\rho \in \mathbb{R}$ **do**
2: **for** $r \in [\rho]_t^\tau : start(r) \in v$ **do**
3: **if** $n_m \in \Sigma_\rho$ **then**
4: **if** $A \iff \mathbb{P}_r$ **then**
5: **if** $n_m = next(r)$ **then**
6: proveď přechod v instanci r
7: **else**
8: invaliduj instanci r
9: **if** $n_m = first(\rho)$ **then**
10: **if** $\nexists r \in [\rho]_t^v : A \iff \mathbb{P}_r$ **then**
11: **if** $constraints(A)$ **then**
12: vytvoř instanci r' s argumenty $\mathbb{P}_{r'}$

Všechny kroky algoritmu je nyní nutné vykonat až po dokončení exekuce metody, protože může být potřebné znát i návratovou hodnotu. První krok, ve kterém se vykonávají přechody v konečných automatech a vytváří nové instance, je popsán pomocí algoritmu 1. V ukázce jsou pro zjednodušení uvedeny pouze kroky pro cíle, celý algoritmus se ještě jednou zopakuje pro spoilery.

Po dokončení všech přechodů se provedou kontroly, zda nedošlo k porušení kontraktu. Zde je algoritmus téměř totožný jako u neparametrické analýzy, pouze je potřeba kontrolovat i omezení a rovnost hodnot pro parametry označené stejnou proměnnou.

3.3 Framework ANaConDA

ANaConDA[1] (Adaptable Native-code Concurrency-focused Dynamic Analysis) je open-source nástroj pro dynamickou analýzu vícevláknových C/C++ programů na binární úrovni [4]. Slouží jako podpora pro vytváření nových analyzátorů, protože obstarává sledování běhu programu a poskytuje analyzátorům informace o důležitých událostech, které v programu nastaly.

ANaConDA je tvořena ze tří hlavních částí – Pin, framework, analyzátory. Pin[2] slouží k dynamické instrumentaci binárního kódu a monitoruje běh programu. Při důležitých událostech informuje jádro prostředí ANaConDA, které získané informace dále poskytuje analyzátorům. Analyzátory pracují na vyšší úrovni a většinou se jedná o implementace algoritmů pro detekci paralelních chyb. Mají k dispozici API funkce, které by vývojářům měly ulehčit nové implementace, a proto se neustále pracuje na jejich rozšiřování.

Komunikace mezi analyzátory a jádrem nástroje ANaConDA probíhá pomocí tzv. zpětných volání (*callback*). V analyzátoru se pomocí speciálních funkcí zaregistrují obslužné rutiny pro události jako například získání či uvolnění zámku, volání funkce, operace *fork* a *join* nebo přístup do paměti, které se spustí vždy, když daná událost nastane v monitorovaném programu. O programu je možné získat také další dodatečné informace, např. hodnoty argumentů funkce. Na základě získaných informací se poté provádí samotný algoritmus analýzy a uživatel je informován o výsledcích a případných chybách.

Algoritmus pro analýzu parametrických kontraktů byl implementován jako nový analyzátor Param-contract-validator, který pro své fungování vyžaduje informace o synchronizačních operacích (ovlivňují posuny logického času a aktualizaci vektorových hodin) a o volání funkcí včetně jejich argumentů a návratových hodnot. V těchto zpětných voláních se poté vykonávají jednotlivé kroky algoritmu. Za účelem analýzy parametrů musela být ANaConDA rozšířena o API funkce, které by ulehčovaly práci s nimi (například konverzní funkce pro převody hodnoty mezi různými datovými typy nebo komparační funkce pro porovnání hodnot libovolného datového typu), a také o zpětná volání, která mohou získávat hodnoty více argumentů monitorovaných funkcí.

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/

[2] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

4. Použití analyzátoru Param-contract-validator

Pro využití analyzátoru Param-contract-validator je nezbytné sestavit konfigurační soubor. Při analýze se bude zpracovávat po řádcích a na každém očekává jednu z následujících možností:

- klauzule kontraktu ve formátu {<cíl> <- <spoiler>},
- omezení typu **datový typ**,
- omezení typu **podmínka**,
- omezení typu **přiřazení hodnoty** nebo
- komentář, což je řádek uvozený znakem #.

Cíl a spoiler se zapisují jako mezerou oddělená posloupnost funkcí s parametry a návratovými hodnotami, např.:

```
{A=fnc_bool() foo(A) <- bar(_,X)}
```

Pro každý parametr, který se objeví v klauzuli kontraktu, musí být specifikován datový typ. Syntaxe tohoto omezení je `parametr : datový typ`. Pro každý datový typ musí být v nástroji ANaConDA implementována podpora v podobě funkcí pro převod hodnot, porovnávání a výpočty výrazů a v současné době jsou podporovány základní datové typy[3].

Podmínky jsou výrazy, jejichž výsledkem je hodnota typu `bool`. Lze pomocí nich omezit hodnoty parametrů, pro které budou vytvářeny instance a pro které nikoli. Mezi operátory, které je pro zápis podmínek i dalších výrazů možno využít, patří `or`, `and`, `not` a binární aritmetické a relační operátory jazyků C/C++. Podporovány jsou také literály typu celé číslo, desetinné číslo, řetězec a pravdivostní hodnota. Ve výrazech lze samozřejmě využívat i názvy parametrů. Specifikace podmínky pro parametr *A* typu `float` potom může vypadat například takto:

```
not ( A == 5.0 and A <= 0.0 )
```

Přiřazením hodnoty lze pomocí výrazu jednoznačně určit hodnotu parametru a zápis může vypadat například takto:

```
B = ( ( ( A + 2 ) - 10 ) * 3 ) / 1
```

Je nutné, aby veškeré parametry a literály použité ve výrazu měly stejný datový typ, konkrétně zde `int`.

[3] `int`, `float`, `double`, `bool`, `const char*`, `string`, `void*`

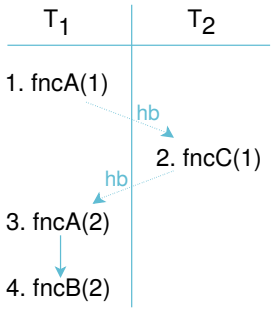
4.1 Příklad analýzy

Výsledky analýzy s parametry či bez jsou z pohledu odhalených chyb neporovnatelné. Využití parametrů může omezit množství chybových hlášení, což je důvod, proč byla tato podpora implementována, ale zároveň může dojít k odhalení více chyb, jelikož bude v paměti drženo více různých instancí, které by bez parametrů splynuly v jednu. Rozdíly mezi jednotlivými analyzátory budou prezentovány na dvou příkladech.

Uvažujme konfigurační soubor

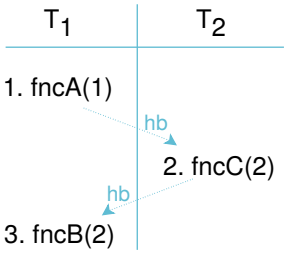
```
{fncA(X) fncB(Y) ← fncC(X)}
X : int
Y : int
Y = X + 1
```

a stopu programu zobrazenou na obrázku 2.



Obrázek 2. Ukázka stopy programu, ve které dojde k porušení kontraktu při parametrické analýze.

Param-contract-validator detekuje porušení kontraktu pro instanci r_1 tvořenou událostmi 1 a 4 instancí s_1 tvořenou událostí 2. Contract-validator by porušení kontraktu nedetekoval, protože instance cíle je pro něj tvořena událostmi 3 a 4, jejichž exekuci spoiler neprokládá.



Obrázek 3. Ukázka stopy programu, ve které dojde k porušení kontraktu při neparametrické analýze.

Pokud by stopa programu vypadala jako na obrázku 3, Param-contract-validator by nedetekoval porušení kontraktu, protože hodnota parametru *X* má u instance cíle a spoileru jinou hodnotu, Contract-validator by však chybu zahlásil.

4.2 Experimenty s analyzátorem

Pro ověření implementace i algoritmu metody bylo vytvořeno cca 100 testovacích případů, které testují vyhodnocení výrazů pro všechny smysluplné kombinace

Tabulka 1. Výsledky analýzy kontraktů pomocí nástrojů Contract-validator a Param-contract-validator.

Test	Čas (s)	Paměť (kB)	T/S	PK
Account (PCV)	1.33	59 106	1	318
Account (CV)	1.14	58 524	1	318
Coord03 (PCV)	2.66	62 760	8	336
Coord03 (CV)	2.36	64 868	8	344
Coord04 (PCV)	1.51	61 200	4	21
Coord04 (CV)	1.31	60 116	4	20
Local (PCV)	1.37	60 924	4	2
Local (CV)	1.23	59 580	4	2
NASA (PCV)	1.67	61 508	1	99
NASA (CV)	1.44	59 096	1	100

Tabulka 2. Výsledek analýzy kontraktů v programu NASA s využitím parametrů.

Test	Čas (s)	Paměť (kB)	T/S	PK
NASA s parametry	1.68	62128	1	1

datových typů a operátorů, přechody v končených automatech, vytváření a rušení instancí a samozřejmě také detekci porušení kontraktů v závislosti na hodnotách parametrů.

Kromě toho byly provedeny experimenty pro srovnání výsledků, časové a paměťové náročnosti obou analyzátorů na stejné sadě příkladů, která byla použita pro původní experimenty s analyzátorem Contract-validator [1]. Výsledky bezparametrické analýzy se nachází v tabulce 1. Zkratka PVC označuje Param-contract-validator a CV Contract-validator. Kromě času a paměti je uveden celkový počet klauzulí kontraktu, které byly analyzovány (sloupec T/S) a počet nalezených porušení kontraktu (sloupec PK). Podle očekávání jsou výsledky obou implementací při analýze bez parametrů téměř totožné.

Param-contract-validator byl také použit pro parametrickou analýzu příkladu NASA [5]. U ostatních příkladů bohužel nebylo možné parametry definovat. Z výsledků zobrazených v tabulce 2 je patrné, že se časová ani paměťová náročnost příliš nezvýšila (jedná se ovšem o poměrně jednoduchý příklad), ale porušení kontraktu bylo detekováno pouze 1. Výsledek tohoto experimentu dokazuje užitečnost parametrické analýzy, která velmi usnadní opravování nalezených chyb.

Analyzátor byl také podroben experimentu na poměrně rozsáhlém projektu. Stejně jako Contract-validator byl použit pro analýzu programu Chromium-1, což je starší verze prohlížeče Chrome obsahující porušení atomicity [6]. Výsledky analýzy jsou uvedeny v tabulce 3. I zde došlo k omezení chybových hlášení ze 14 na 2, ačkoli se nejedná o tak markantní rozdíl jako u předchozího příkladu. Experiment ovšem dokázal, že Param-contract-validator je možné použít i pro analýzu rozsáhlých programů[4] za rozumnou cenu.

Tabulka 3. Srovnání analýzy programu Chromium-1.

Analyzátor	Čas (s)	Paměť (kB)	T/S	PK
CVC	3:19.61	1873260	1	14
PVC bez parametrů	3:29.53	1911608	1	14
PVC s parametry	4:28.41	1912916	1	2

5. Závěr

V článku byla představena problematika kontraktů a jejich dynamické analýzy, která ulehčuje hledání chyb v paralelních programech. Dále byl navržen algoritmus pro analýzu parametrických kontraktů, který bude z monitorovaných událostí vyvozovat ještě přesnější závěry. Ten byl implementován jako nový analyzátor pro nástroj ANaConDA a byl podrobně otestován a použit pro experimenty nad sadou programů obsahujících porušení atomicity. Experiment nad programem Chromium-1 dokázal, že je možné parametrickou analýzu využít i pro velké projekty.

V budoucnu se zaměříme především na využití analyzátoru a odhalování chyb v reálných programech, s čímž souvisí hledání kontraktů, které by pro dané projekty dávaly smysl. Podle jejich požadavků budeme dále analyzátor rozšiřovat o podporu speciálních situací, které by mohly vyvstat.

Poděkování

Chtěla bych poděkovat Ing. Alešovi Smrčkovi, Ph.D. a Ing. Janu Fiedorovi, Ph.D. za jejich vedení a rady, které mi při práci velmi pomáhaly.

Literatura

[1] Ricardo J. Dias, Carla Ferreira, Jan Fiedor, João M. Lourenço, Aleš Smrčka, Diogo G. Sousa, and Tomáš Vojnar. Verifying concurrent programs using contracts. In *2017 IEEE International Conference on Software Testing, Verification and Validation*, pages 196–206, 2017.

[2] Cormac Flanagan and Stephen Freund. Fasttrack: efficient and precise dynamic race detection. In *Communications of the ACM*, pages 93–101, 01 November 2010.

[4] Chromium-1 se skládá ze 7.5 milionu řádků kódu.

[3] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, pages 558–565, 01 July 1978.

[4] Jan Fiedor and Tomáš Vojnar. Anaconda: A framework for analysing multi-threaded C/C++ programs on the binary level. In *Proc. of 3rd International Conference on Runtime Verification—RV'12*, volume 7687 of LNCS, pages 35–41, Istanbul, Turkey, 2012.

[5] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Journal on Software Testing, Verification and Reliability (STVR)*, 2003.

[6] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: A concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.