Programowanie Współbieżne i Rozproszone

# Zadanie 1

Tomasz Zakrzewski, tz336079

## Wstęp

W trakcie pracy nad rozwiązaniem, Spin okazał się być często dużą przeszkodą. Pojawiły się głównie dwa problemy, które musiałem kreatywnie obchodzić:

- Zbyt długie LTLe powodują buffer overflow - powyżej pewnej długości mój system to wykrywa i zabija proces - poniżej zdarzały się subtelne błędy (de facto faktyczny buffer overflow :) ). Często z tego powodu muszę rozbijać LTLe na mniejsze grupy i testować je osobno, argumentując przy tym że pokrywam wszystkie przypadki. Robię tak dla LTLi, które na najwyższym poziomie zawierają same andy - w ten sposób mam pewność, że jeśli zawsze zachodzą, to mogę je rozbić i testować osobno - gdyby którykolwiek nie działał, to koniunkcja też byłaby fałszywa;
- 10GB wolnego RAMu nie zawsze wystarcza do policzenia czegoś. Zdarza mi się więc testować LTLe dla mniejszej liczby procesów, lub z jednym wysyłającym.

O każdym z powyższych uproszczeń wspominam przy odpowiednich LTLach niżej. Jeśli nie wspominam, to znaczy że testowałem w domyślny sposób - na 3 procesach, każdy pełni rolę zarówno wysyłającego jak i odbierającego.

## Blokowanie send dla mailboksów i rozmiary kolejek

Teoretycznie rozwiązanie w pełni poprawne powinno tworzyć nowy proces dla każdego włożenia wiadomości do kolejki - proces ten powinien się na kolejce zawiesić jesli nie może umieścić w niej wiadomości - w ten sam sposób spełniając założenie o nieblokowaniu kolejki. Warto jednak zauważyć, że pesymistycznie w kolejce może znaleźć się n^3 wiadomości, gdzie n jest ilością procesów. Z kolei już dla 3 procesów, nie zawsze na przeciętnej maszynie wystarcza RAMu, a wielokrotnie też cierpliwości, co wynika z wykładniczej natury weryfikacji za pomocą spina. Stwierdziłem więc, że nie ma sensu pisać rozwiązania, którego działania nie da

się zweryfikowac. Spin nauczył mnie już, że jest kiepskiej jakości oprogramowaniem i niespecjalnie wierzę, że dałoby się go odpalić na klastrze obliczeniowym, a to jest zapewne kaliber sprzętu na którym możnaby wysycić kolejki w moim rozwiązaniu.

Z powyższych powodów, wszystkie kolejki na wiadomości w moim rozwiązaniu mają stały rozmiar. Sam rozmiar wybrany przeze mnie to 255, co wynika znowu z kiepskiej jakości oprogramowania jakim jest Spin - gramatyka Promeli zakłada że tablice mają rozmiar będący stałymi liczbowymi, ale w dokumentacji nie doszukałem się nigdzie jaki jest zakres wielkości. Praktyka pokazała, że np. dla tablicy rozmiaru 10tys., wywołanie spina kończyło się losowymi błędami w inicie (trail ended after -1 steps…), więc założyłem jeden bajt jako rozsądne ogarniczenie. Do naszych potrzeb w zupełności wystarcza.

## Tabela spełnialności

| | **Basic broadcast** | **Reliable broadcast** |
|---|---|---|
| **no_duplication** | ✓ | ✓ |
| **no_creation** | ✓ | ✓ |
| **weak_validity** | ✓ | ✓ |
| **strong_validity** | ✓ | ✓ |
| **weak_agreement** | ✗ | ✓ |
| **strong_agreement** | ✗ | ✗ |

Wszystkie LTLe dla mailboksów są spełnione.

## Opisy formuł

### mb_guaranteed_delivery

Sprawdzam, czy zawsze jeśli a wysyła do b wiadomość o treści a oraz b zawsze kiedyś będzie w stanie `proc_receive`, to b kiedyś odbierze wiadomość a. Ze względu na buffer overflow, mam formułę podzieloną na dwie - jedną dla wysyłania wiadomości samemu sobie, drugą dla każdej pary procesów (z symetrią).

### mb_at_most_once_delivery

Dla każdej wiadomości sprawdzam, czy została do każdego procesu dostarczona dokładnie raz - zawsze jeśli proces odebrał taką wiadomość, to nie znajdzie się znowu w stanie proc_send, dopóki wartość rm (odebranej wiadomości) nie zmieni się na jakąś inną. Sprawdzam dla każdego procesu.

### mb_no_fabrication

Zawsze kiedyś, jeśli kiedyś proces był w stanie `proc_receive` z odebraną wiadomością o danej wartości, to któryś z innych procesów wcześniej wysłał tę wiadomość. Ze względu na buffer overflow, sprawdzam to osobno dla każdego procesu.

### no_duplication

Ze względu na buffer overflow dla zbyt długich formuł w spinie, rozbijam te LTLe na 3 części - każdą osobno z punktu widzenia innego procesu. BSO załóżmy, że patrzę z punktu widzenia procesu 1. Sprawdzam dla każdej możliwej wiadomości (1-3), czy jeśli proces ją otrzymał, to jeśli kiedyś jeszcze wejdzie w proc_await_message, to nie odbierze już tej samej wiadomości.

Używam flagi ONE_BROADCASTER dla reliable broadcastu.

### no_creation

Sprawdzam, czy jeśli kiedyś dana wiadomość trafia do danego procesu, to czy była wcześniej wysłana przez jakiś proces. Sprawdzam to dla każdej trójki `proces1`, `proces2`, `wiadomość`. Ze względu na rozmiar, dzielę to na 4 LTLe - jeden testujący wysyłanie swojej wiadomości samemu do siebie i 3 sprawdzające każdą możliwą wiadomość między daną parą procesów (symetrycznie).

Tego LTLa testuję z flagą ONE_BROADCASTER dla reliable broadcastu.

### weak_validity

Zawsze, jeśli proces jest w stanie poprawnym, to dostarczy kiedyś wiadomość do samego siebie. Powtarzam dla każdego procesu, wszystko w jednym LTLu. Tego LTLa testuję z flagą ONE_BROADCASTER.

### strong_validity

Zawsze, jeśli kiedyś proces jest w stanie poprawnym oraz wysłał swoje id jako wiadomość, każdy poprawny proces (włącznie z nim samym) odbierze tę wiadomość). Tego LTLa testuję z flagą ONE_BROADCASTER.

### weak_agreement

Dla każdego procesu: jeśli jest poprawny i kiedyś dostarcza wiadomość, to każdy proces, jeśli jest poprawny, dostarcza tę wiadomość. Przez ogarniczenia RAMu, testuję to z pojedynczym broadcasterem, więc zakładam że sprawdzam tylko wiadomość o treści 1. (dla basic da się testować z wieloma broadcasterami, ale chodzi o najszybsze znalezienie błędu, a on się znajduje już dla jednej wiadomości).

### strong_agreement

Jak `przy weak_agreement`, tylko bez sprawdzania poprawności procesu po lewej stronie implikacji. Testuję również w analogiczny sposób, więc czynię te same założenia o wiadomości.