Wykorzystanie systemów regułowych do implementacji mechanizmu obsługi zdarzeń.

Kajetan Rzepecki

EIS 2014
17 stycznia 2015

1 Wstęp

Celem projektu jest zbadanie możliwości oraz opłacalności implementacji mechanizmu obsługi zdarzeń w systemie programistycznym z wykorzystaniem systemów regułowych z **wnioskowaniem w przód**.

Mechanizm ów ma za zadanie ułatwić obsługę zdarzeń zachodzących w systemie poprzez umożliwienie definiowania reguł i faktów w sposób deklaratywny i zintegrowany ze składnią i semantyką języka programowania, w którym jest wykorzystywany:

```
(assert!
  (predicate subject object) ;; Dodanie faktu do bazy faktów.
  ...)

(whenever rule               ;; Dodanie reguły reprezentującej zdarzenie
  action                     ;; oraz instrukcji je obsługujących do
  ...)                       ;; bazy faktów.

(declare (foo x y)           ;; Deklaracja funkcji, zawierająca
  (_@ a function)            ;; automatycznie inferowane fakty
  (_@ arity 2)               ;; dotyczące funkcji,
  (@ big-oh 1)               ;; dodatkowe fakty dostarczone przez autora oraz
  (@ equal (foo 2 21) 23))   ;; informacje o kontraktach funkcji.

(define (foo x y)
  (+ x y))
```

2 Analiza problemu

Mechanizm obsługi zdarzeń będzie docelowo wykorzystywany w zastosowaniach Internet of Things - środowisku rozproszonym z wysoką redundancją, gdzie wiele węzłów tworzących klaster udostępnia zbliżone funkcjonalności o nieco różnych charakterystykach.

Na potrzeby projektu, węzłem określana będzie instancja maszyny wirtualnej języka programowania, na której dostępne są **moduły** - zbiory funkcji realizujących jakąś funkcjonalność. Dynamicznie łączące i rozłączające się węzły będą generowały zdarzenia (indukowane przez i składające się z elementarnych operacji modyfikacji bazy faktów) takie jak: połączenie nowego węzła, pojawienie się nowego modułu, czy dowolne zmiany zawarte w kodzie przez porgramistę. Zdarzenia te będą przesyłane do pozostałych połączonych węzłów.

Dzięki zastosowaniu systemu regułowego, moduły wchodzące w skład danego węzła będą mogły reagować na napływające zdarzenia odpowiednio modyfikując swoje zachowanie. W celu obsługi danego zdarzenia definiowana będzie reguła (o dowolnej złożoności), która w momencie spełnienia uruchamiała będzie szereg instrukcji obsługujących zdarzenie.

Wykorzystanie wnioskowania w przód umożliwi definiowanie reguł z wyprzedzeniem - powiązane z nimi instrukcje obsługujące zdarzenie zostaną wykonane dopiero w momencie spełnienia reguły, po dostatecznej modyfikacji bazy faktów.

2.1 Przykład zastosowania proponowanego mechanizmu

Posiadając następujący moduł pobierający dane GPS z czujnika:

```
(define-module gps-default
  (provide gps)

  (declare (get-location)
    (@ tolerance 0.01))

  (define (get-location)
    ;; Code that gets current location.
    ))
```

...oraz następującą applikację z niego korzystającą:

```
(define-module gps-app
  (import 'gps-default)

  (define (use-gps-data)
    (let ((curr-location (gps-default:get-location)))
      ;; Use gps function to do something.
      ))

  (define (update-state)
    ;; Update apps state using latest gps data.
    ))
```

...programista jest w stanie zadeklarować obsługę pojawienia się modułu pobierającego dane GPS z większą dokładnością:

```
(define-module gps-app
  (import 'gps-default)

  (define gps-location-function gps-default:get-location)

  (whenever (and (module-loaded ?node ?module)
                 (declares ?module ?function)
                 (name ?function 'get-location)
                 (tolerance ?function ?tol)
                 (< ?tol 0.01))
            (set! gps-location-function ?node:?module:?function)
            (update-state))

  (define (use-gps-data)
    (let ((curr-location (get-location-function)))
      ;; Use gps function to do something.
      ))

  (define (update-state)
    ;; Update apps state using latest gps data.
    ))
```

...dzięki czemu, po podłączeniu węzła udostępniającego następujący moduł:

```
  (define-module gps-vendor
    (provide gps)

    (declare (get-location)
      (@ tolerance 0.0001))

    (define (get-location)
      ;; Code that gets current location.
      ))
```

...system działający na dotychczasowym węźle automatycznie będzie wyświetlał dane z większą dokładnością.

2.2 Analiza możliwości implementacji i przydatności

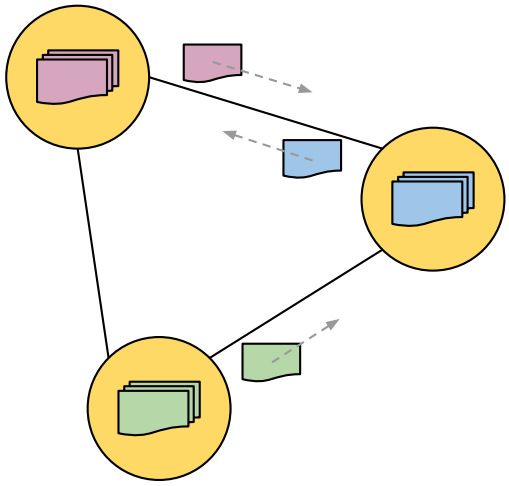
Przydatność proponowanego mechanizmu jest potencjalnie nieoceniona w domenie języków programowania ponieważ umożliwia ekspresję złożonego przepływu sterowania w deklaratywny sposób - za pomocą krótkich, dobrze zdefiniowanych reguł. Dzięki temu programista tworzący aplikacje wykorzystując system regułowy może skoncentrować się na rezultatach rozwiązania problemu, nie zaś na sposobie ich osiągnięcia - system regułowy zrobi to za niego.

Wykorzystanie systemów regułowych do implementacji systemu modułów języka programowania dodatkowo umożliwia automatyczne i skalowalne tworzenie rozproszonych, dynamicznych systmemów charakteryzujących się dużą redundancją - takich jak Internet of Things. Podejście regułowe zapewnia interfejs komunikacji i mechanizm rozwiązywania konfliktów między poszczególnymi modułami/jednostkami aplikacji, co ułatwia ich kompozycję i umożliwia redundancję.

Dodatkowym atutem zastosowania systemów regułowych jest potencjalna skalowalność systemów z nich korzystających - do systemu w każdym momencie można dodać więcej węzłów

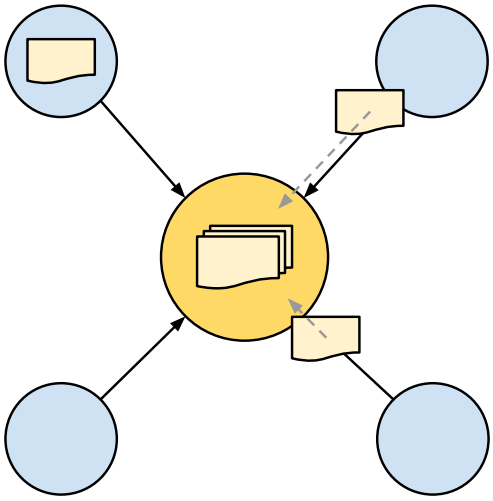
dostarczających pewne usługi, a system automatycznie będzie w stanie z nich korzystać, reagować na zmiany ich stanu i obsługiwać zdarzenia przez nie sygnalizowane. Zwiększenie ilości węzłów dostarczających te same usługi pozytywnie wpływa także na stabilność i bezpieczeństwo działania systemu dzięki zwiększeniu jego redundancji.

Potencjalne zalety wykorzystania systemów regułowych w systemach rozproszonych szczególnie dobrze widać w idealnym przypadku zaprezentowanym na poniższym diagramie, gdzie każdy węzeł zawiera pewne reguły i generuje zdarzenia przesyłane do pozostałych węzłów systemu:



Taka konfiguracja zapewnia wszystkie opisane powyżej zalety kosztem zasobów wymaganych do implementacji i działania systemów regułowych na każdym węźle.

W przypadku Internet of Things mały rozmiar i oganiczona wydajność pamięciowa/obliczeniowa urządzeń wchodzących w jego skład niestety uniemożliwia stosowanie istniejących, profesjonalnych systemów regułowych w celu implementcji powyższej, idealnej konfiguracji prowadząc do następującego układu:



W tej konfiguracji istnieją dwie klasy węzłów:

- węzły regułowe, których oprogramowanie korzysta z systemów regułowych do obsługi zdarzeń,
- węzły zdarzeniowe, które jedynie generują zdarzenia i przesyłają je do węzłów regułowych systemu.

Ponieważ węzły zdarzeniowe nie umożliwiają definicji reguł, są one zdane na alternatywne, często imperatywne i mało skalowalne sposoby obsługi zdarzeń, istotnym jest więc by proponowany w następujących sekcjach mechanizm obsługi zdarzeń charakteryzował się możliwie niskim narzutem wydajnościowym.

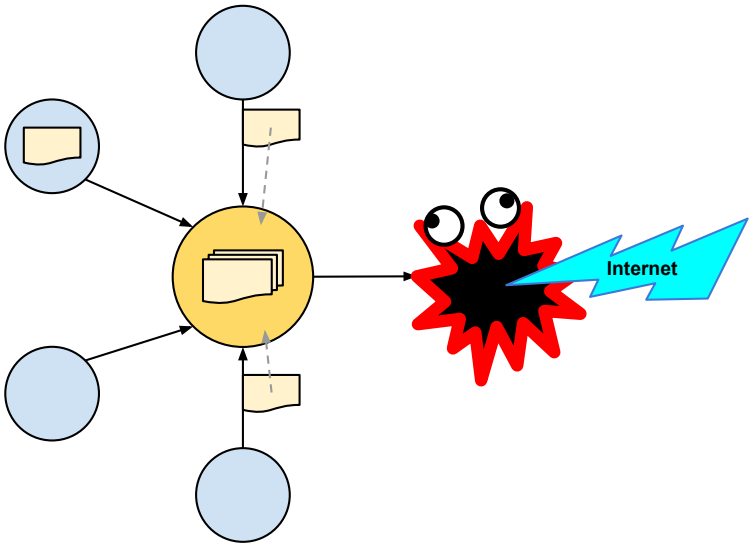
2.3 Podobne rozwiązania

Systemy regułowe wykorzystywane są w wielu różnych dziedzinach, przede wszystkim w systemach ekspertowych do przechowywania i manipulowania wiedzy.

Kluczowym przykładem jest system CLIPS, charakteryzujący się elastycznością - jest on narzędziem do budowy systemów ekspertowych umożliwiającym osadzenie go w gotowej aplikacji. Nie posiada on jednak wszystkich aspektów języka programowania ogólnego przeznaczenia, przez co jego wykorzystanie nie jest dogodne.

Następnym przykładem wykorzystania systemów regułowych do obsługi zdarzeń jest pakiet Drools Fusion, wykorzystujący reguły oraz arbitralny kod w języku Java w kontekście systemów reguł biznesowych (BRMS). Ponieważ jest to system *profesjonalny* i skierowany do dużych korporacji, jest on bardzo rozległy i posiada bardzo wiele, niekoniecznie pożądanych funkcjonalności, przez co jego wykorzystanie w projekcie również nie jest dogodne.

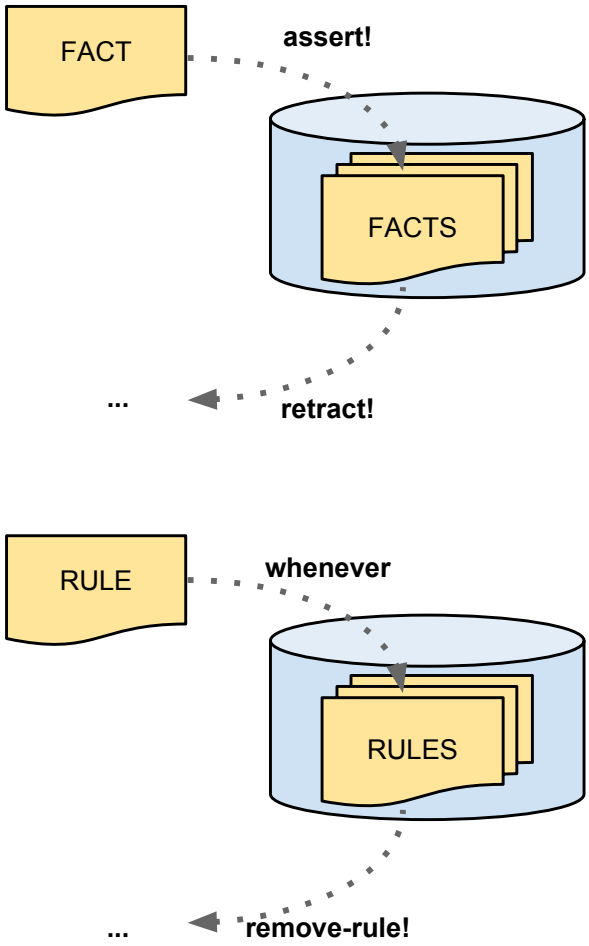
W domenie Internet of Things również powstają rozwiązania oparte o systemy regułowe. Przykładem jest system zaproponowany przez firmę Bosh, który umożliwia zarządzanie inteligentnymi, podłączonymi do IoT urządzeniami z wykorzystaniem reguł. Niestety, jest to system komercyjny i scentralizowany, przez co traci on wiele z zalet zapewnianych przez systemy regułowe:



Klienci mają jedynie możliwość pośredniej ingerencji w tzw. CCU, które są odpowiednikami węzłów regułowych opisanych powyżej. Reguły w CCU modyfikowane są przez zewnętrzny serwis operujący w *Chmurze*, od którego zależy cały budowany system.

3 Szkic rozwiązania

W celu realizacji projektu niezbędne będzie zaimplementowanie dwóch struktur danych, **fact store** oraz **rule store**, które będą odpowiedzialne za przechowywanie odpowiednio faktów i reguł w systemie. Na poniższym diagramie przedstawiono obie struktury danych oraz ich interfejsy:



Fact store umożliwia dodawanie nowych faktów przez `assert!` oraz usuwanie istniejących faktów przez `retract!`. Dodatkowo, możliwe jest sygnalizowanie zdarzeń poprzez `signal!`, które jest złożeniem `assert!` i `retract!`

Rule store umożliwia definiowanie nowych reguł poprzez konstrukcję `whenever` oraz ich usuwanie przez `remove-rule!`. Dodatkowo, możliwe jest czasowe ograniczenie reguł wykorzystując inne konstrukcje języka oraz podstawowe operacje, jak `whenever` i `remove-rule!`.

Powyższy protokół jest dostatecznie elastyczny, by umożliwić różne implementacje systemu regułowego i jednocześnie na tyle ekspresywny, by umożliwić w prosty sposób obsługę złożonych zdarzeń w systemie.

3.1 Porównanie różnych algorytmów

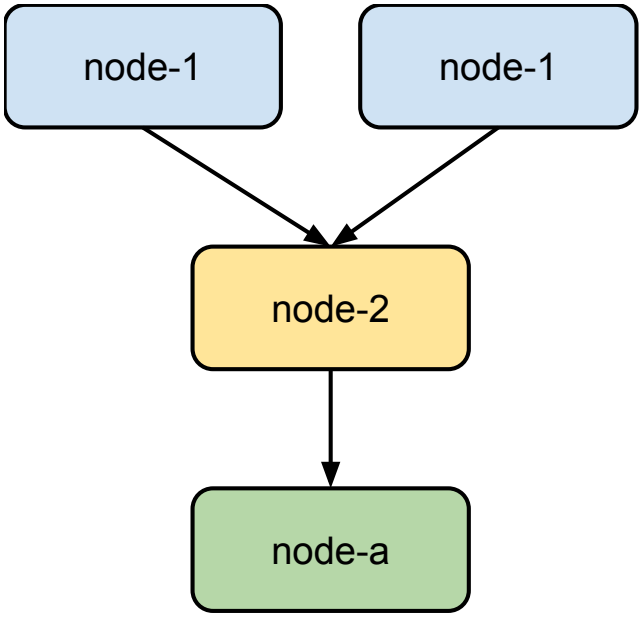
Najłatwiejszą implementacją powyższego protokołu jest naiwne podejście iteracyjne polegające na iteracyjnym sprawdzaniu każdego faktu z każdą regułą. Rozwiązanie to, mimo że jest proste, jest również bardzo niewydajne - złożoność obliczeniowa rzędu `O(RF^P)`, dla `R` reguł, `F` faktów i `P` średniej ilości wzorców przypadających na lewą stronę reguły.

Standardowym podejściem jest wykorzystanie algorytmu **Rete** - zaprojektowanego w roku 1974 przez dr Charls'a Forgy'iego, który charakteryzuje się znacznie lepszą złożonością obliczeniową - rzędu `O(RFP)` kosztem zwiększonego wykorzystania pamięci. Algorytm ten jest relatywnie nieskomplikowany i jednocześnie zadowalająco wydajny.

Istnieją także wersje zrównoleglone algorytmu Rete, opisane szeroko w *Parallel Algorithms and Architectures for Rule-Based Systems*, jednak owe algorytmy są zbyt skomplikowane na potrzeby projektu.

3.2 Rete

W związku z powyższym, zdecydowano się na implementacją podstawowej wersji algorytmu Rete. Algorytm Rete polega na budowie sieci reguł złożonej z węzłów należących do jednej z kilku kategorii przedstawionych na poniższym diagramie:



Węzły kategorii **node-1**, zwane także węzłami alfa, to proste węzły dopasowujące fakty do wzorców obecnych w regułach, np. `(provides ?x ?y)`. Posiadają one jedno wejście i potencjalnie wiele wyjść.

Węzły kategorii **node-2**, zwane także węzłami beta, lub *join-nodes* posiadają pamięć faktów i służą do unifikacji faktów pochodzących z dwóch wejść. Wynikiem ich działania jest szereg zunifikowanych faktów, pasujących do obu gałęzi sieci prowadzących do obecnego węzła.

Węzły kategorii **node-a** to węzły akcji, które przechowują prawe strony reguł i posiadają tylko jedno wejście - wynik działania sieci Rete prowadzący do uruchomienia danej produkcji.

Powyższe kategorie to tylko trzy podstawowe rodzaje węzłów, można wyróżnić jeszcze szereg innych przydatnych węzłów:

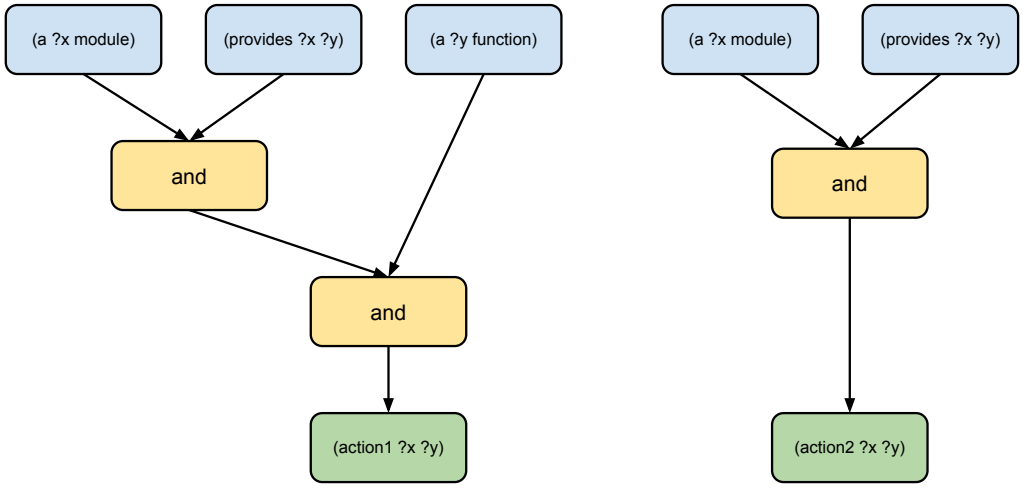
- **node-p** - węzły zawierające predykaty, które umożliwiają nietrywialną filtrację faktów,
- **node-r** - węzły redukcji, które przechowując stan wewnętrzny zezwalają na akumulację pewnego wyniku (np. minimalizację pewnej wartości obecnej w faktach).

Algorytm transformuje produkcje reguł zamieniając ich lewe strony na kombinacje `node-1` i `node-2`, natomiast prawe strony na węzły kategorii `node-a`. Dla przykładu, poniższe reguły:

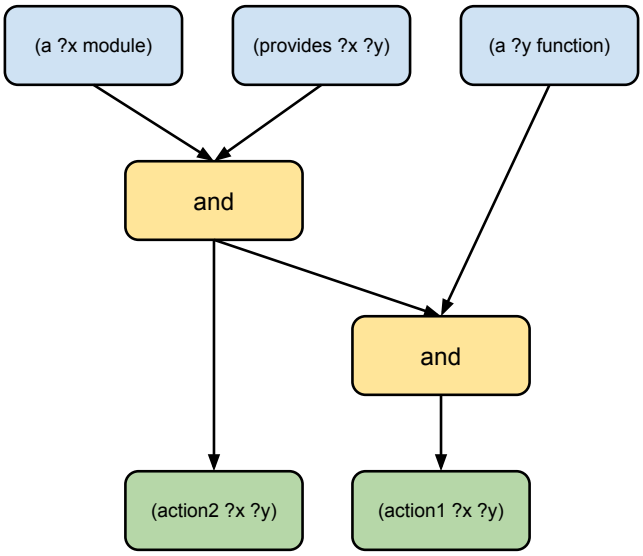
```
;; Rule 1
(whenever (and (a ?x module)
               (provides ?x ?y)
               (a ?y function))
          (action1 ?x ?y))

;; Rule 2
(whenever (and (a ?x module)
               (provides ?x ?y))
          (action2 ?x ?y))
```

...zostaną zamienione w wyniku działania algorytmu Rete na dwie rozłączne sieci zaprezentowane poniżej:



...które następnie zostaną zredukowane, w celu usunięcia węzłów redundantnych i optymalizacji ilości wykonywanych operacji, do następującej sieci Rete:



Ponieważ druga część algorytmu polegająca na redukcji sieci jest operacją relatywnie skomplikowaną, zdecydowano się w pierwszej kolejności zaimplementować *naiwną* wersję algorytmu Rete, która nie optymalizuje struktury sieci. Jeśli czas pozwoli, redukcja sieci zostanie dodana w terminie późniejszym.

4 Prototyp rozwiązania

Implementacja opisanych w poprzedniej sekcji algorytmów została udostępniona w internecie. Zawiera ona definicje następujących węzłów:

- `node-1` - węzły alfa,
- `node-2` - węzły beta,
- `node-2l` - lewostronne adaptery węzłów beta (ułatwiają one obsługę sieci),
- `node-a` - węzły akcji,
- `node-r` - węzły redukcji,
- `node-p` - węzły filtracji.

Implementacja umożliwia wykonywanie następujących operacji:

- `assert!` - asercja faktu do bazy faktów,
- `retract!` - retrakcja faktu z bazy faktów,
- `signal!` - sygnalizacja zdarzenia,
- `whenever` - definicja reguły,
- `remove-rule!` - logiczne usunięcie reguły (fragment sieci Rete odpowiadający lewej stronie reguły pozostaje w sieci nawet po jej usunięciu).

Dodatkowo, implementacja pozwala na parametryzowanie akcji w definicjach reguł, dzięki czemu możliwe jest wykorzystanie dopasowanych wartości w łatwy sposób:

```
(whenever pattern
          variables => actions ...)
```

Obecnie, implementacja nie optymalizuje reprezentacji sieci Rete, jednak w przyszłości będzie możliwe łatwe tego zrealizowanie.

Reguły uruchamiane są sekwencyjnie, co może doprowadzić do zaistnienia nieskończonych pętli w programach z nich korzystających, ponieważ implementacja nie buduje zbioru konfliktowego i agendy uruchamiania produkcji. Rozwiązaniem tego problemu może być dodanie obsługi agendy lub asynchroniczne przetwarzanie asercji i retrakcji faktów. W związku z przeznaczeniem projektu zdecydoano się na drugie rozwiązanie, którego implementacja wymaga integracji projektu z docelowym językiem programowania.

4.1 Przykłady zastosowania systemu regułowego

System regułowy powstały w wyniku projektu pozwala między innymi uruchamiać następujące fragmenty kodu:

```
;; This is an example usage of the Rete-based RBS.

(load "rete.scm")

(reset!)

(define new-foo
  (whenever (provides ?m foo)
            () => (display "New foo!\n")))

(whenever (and (module ?m)
               (provides ?m gps))
          () => (display "New GPS!\n"))

(whenever (reduce ?min-t
                  (min 0.1 ?t)
                  (and (module ?m)
                       (provides ?m gps)
                       (tolerance ?m gps ?t)))
          (?min-t) =>
          (display "Best GPS: ")
          (display ?min-t)
          (display "!\n"))

(whenever (filter (and (tolerance ?m1 ?gps ?t1)
                       (tolerance ?m2 ?gps ?t2))
                  (<= ?t1 ?t2)
                  (not-equal? ?m1 ?m2)
                  ((lambda (g) (equal? g 'gps)) ?gps))
          (?t1 ?t2) =>
          (display "Better GPS: ")
          (display ?t1)
          (display " vs. ")
          (display ?t2)
          (display "!\n"))

(assert! (module A))
(assert! (provides A foo)) ;; New foo!
(assert! (provides A bar))

(assert! (module B))
(assert! (provides B foo)) ;; New foo!
(assert! (provides B gps)) ;; New GPS!

(assert! (provides C gps))
(assert! (module C))       ;; New GPS!

;; Event signaling:
(signal! (provides A gps)) ;; New GPS!
(retract! (module B))
(assert! (module B))       ;; New GPS!

;; Reduction & filtration nodes:
(assert! (tolerance C gps 0.01))
(assert! (tolerance B gps 0.001))
(assert! (tolerance A gps 0.0001)) ;; A doesn't provide GPS.

;; Rule removal:
(remove-rule! new-foo)
(assert! (provides C foo))
```

5 Analiza proponowanego rozwiązania

Systemy regułowe stanowią dobry, relatywnie nieskomplikowany formalizm, na którym można oprzeć implementację mechanizmu obsługi zdarzeń. Zapewniają one wysoką ekspresywność i dobrze pasują do semantyki języków programowania do programowania rozproszonego bazujących na przetwarzaniu zdarzeń. Dodatkowo, znane są dostatecznie wydajne algorytmy, dzięki którym wykorzystanie takiego mechanizmu w środowisku *produkcyjnym* jest realistyczne.

5.1 Analiza wydajności proponowanego rozwiązania

Niniejsza sekcja zawiera analizę wydajności proponowanego mechanizmu obsługi zdarzeń bazującego na algorytmie Rete, którego teoretyczna złożoność obliczeniowa wynosi `O(RFP)` dla `F` faktów, `R` reguł oraz `P` wzorców przypadających średnio na regułę, z możliwością optymalizacji przez łączenie podobnych fragmentów sieci do `O(aRFP)`, gdzie `a` należy do przedziału (0, 1] i zależy od *podobieństwa* podsieci.

Niestety implementacja nie optymalizuje reprezentacji sieci, co uniemożliwiło osiągnięcie najlepszej złożoności obliczeniowej; w najgorszym przypadku, `R` identycznych reguł, z których każda definiuje `P` wzorców oraz `F` faktów, każda reguła musi unifikować przynajmniej `R` wzorców, co jest uruchamiane jednorazowo dla każdego z `F` faktów. Otrzymana złożoność obliczeniowa jest więc rzędu `O(RFP)`.

Dla porównania wykorzystano *naiwną* implementację, która dla każdej z `R` reguł kombinatorycznie unifikuje `F` faktów `P` razy, skutkując złożonością obliczeniową rzędu `O(RF^P)`.

Kod testujący wykorzystuje stałą liczbę **5** reguł o średniej liczbie **3.4** wzorców oraz zmienną liczbę faktów dodawanych/usuwanych/sygnalizowanych w bazie faktów inferowanych z implementacji poszczególnych modułów projektu, na których przeprowadzane są losowe operacje `signal!` (40%), `assert!` (35%) oraz `retract!` (25%):

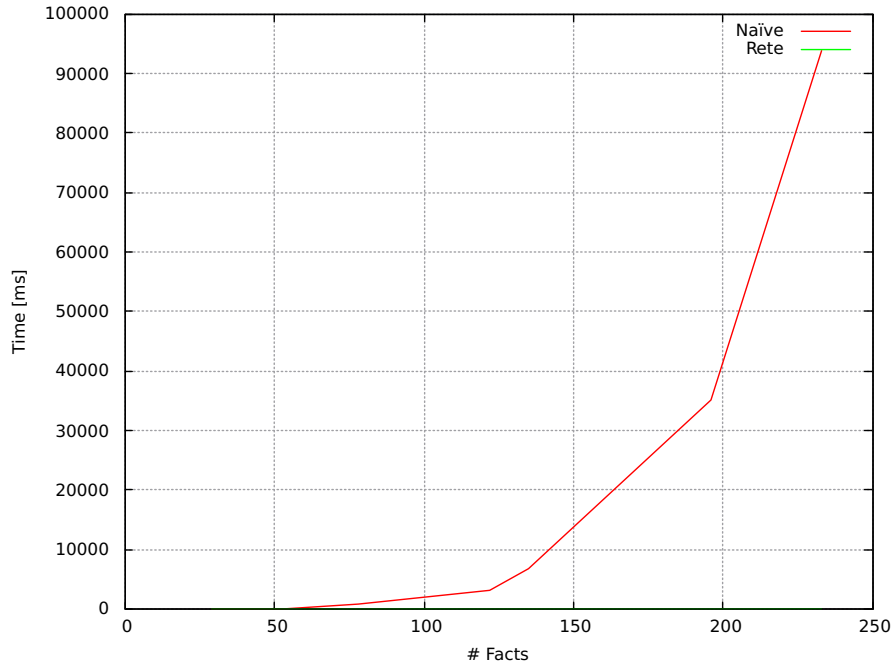
```
(module benchmark)
(provides benchmark benchmark)
(arity benchmark benchmark 1)
(argument benchmark benchmark seed)
(provides benchmark test-naive)
(arity benchmark test-naive 1)
(argument benchmark test-naive facts)
(provides benchmark test-rete)
(arity benchmark test-rete 1)
(argument benchmark test-rete facts)
(provides benchmark test)
(arity benchmark test 1)
(argument benchmark test bench-fun)
(provides benchmark infere-all)
(arity benchmark infere-all 1)
(argument benchmark infere-all modules)
(provides benchmark infere-module)
(arity benchmark infere-module 1)
(argument benchmark infere-module name)
(provides benchmark infere)
(arity benchmark infere 2)
(argument benchmark infere name)
(argument benchmark infere form)
(provides benchmark ->list)
(arity benchmark ->list 1)
(argument benchmark ->list improper-list)
(provides benchmark bench-function)
(arity benchmark bench-function 1)
(argument benchmark bench-function facts)
```

Kod testujący zbiera różne statystyki z działania wybranej implementacji, takie jak czasy działania, przyrost zużycia pamięci, czy liczby mutacji różnych struktur danych wykorzystanych w implementacji.

Ponieważ tylko czasy działania wykazywały istotne różnice, pozostałe statystyki zostały pominięte w niniejszej analizie. Otrzymane czasy działania w zależności od ilości inferowanych faktów zawarto w poniższej tabeli:

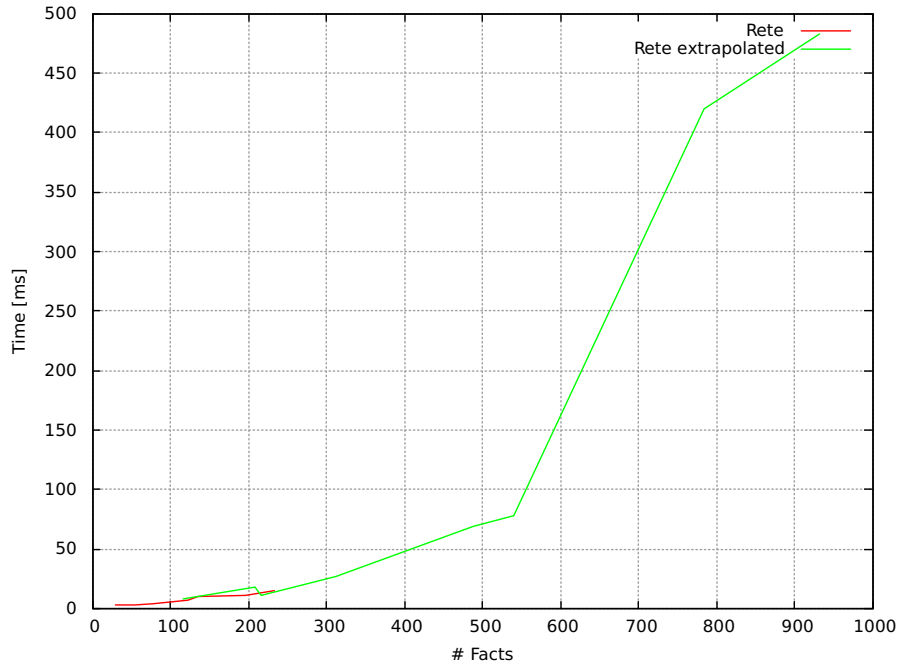
# Facts	Naïve [ms]	Rete [ms]
29	4	3
52	8	3
54	26	3
78	812	4
122	3117	7
135	6748	10
196	35111	11
233	93865	15

…oraz zaprezentowano na poniższym wykresie:



Otrzymane wyniki jednoznacznie potwierdzają, iż implementacja bazująca na algorytmie Rete jest znacznie bardziej wydajna czasowo od naiwnej, wykładniczej implementacji i, przynajmniej dla małych próbek faktów, zachowuje liniową charakterystykę złożoności czasowej.

Aby lepiej przeanalizować stworzoną implementację, ekstrapolowano pomiary wykorzystując większą próbkę danych otrzymaną przez zwielokrotnienie faktów inferowanych z kodu systemu regułowego. Niestety próbka ta zawiera wiele duplikatów, przez co nie oddaje wiernie rzeczywistych charakterystyk wykorzystania systemu. Wyniki zaprezentowano na poniższym wykresie:



Ekstrapolowane wyniki sugerują wykładniczy trend złożoności obliczeniowej implementacji w funkcji ilości faktów, co może wskazywać na pewne niedociągnięcia implementacji, lub zaistnienie pewnych czynników zewnętrznych, takich jak dodatkowe czasy zarządzania pamięcią.

Jednoznaczne określenie praktycznej wydajności powstałego mechanizmu obsługi zdarzeń będzie wymagało dokładniejszych testów przeprowadzonych na specjalnie przygotowanych, realistycznych scenariuszach wykorzystania mechanizmu w praktyce.

5.2 Wnioski

- Algorytm Rete jest relatywnie nietrudny do zaimplementowania. Całość implementacji zajmuje zaledwie dwa razy więcej kodu niż niewydajne, naiwne rozwiązanie.
- Wydajność algorytmu Rete jest zadowalająca nawet pomimo braku optymalizacji reprezentacji sieci. Dodatkowo, podczas testów wydajnościowych nie spotkano się z problemem nadmiernego zużycia pamięci podczas działania algorytmu.
- Najważniejszym wnioskiem, jest fakt, iż systemy regułowe stanowią dobrą bazę do implementacji mechanizmów obsługi zdarzeń.

6 Bibliografia

- Charles L. Forgy, *Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem*, Artificial Intelligence 19 (1982), 17-37, `http://dl.acm.org/citation.cfm?id=115736`
- Anoop Gupta, Charles Forgy, Allen Newell, Robert Wedig, *Parallel Algorithms and Architectures for Rule-Based Systems*, SIGARCH Comput. Archit. News, May 1986, 28-37, `http://dl.acm.org/citation.cfm?id=17360`
- Rule-based Event Management in the Internet of Things and Services