Wykorzystanie ontologii w językach programowania do programowania rozproszonego.

Kajetan Rzepecki

EIS 2014
27 stycznia 2015

1 Wstęp

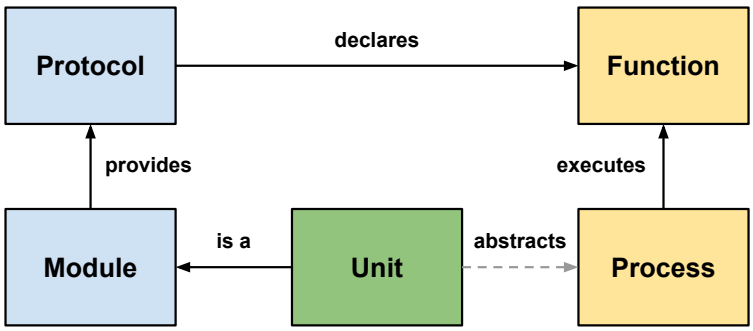
Celem projektu jest zbadanie możliwości oraz opłacalności wykorzystania ontologii w celu reprezentacji wiedzy w języku programowania przeznaczonego do programowania rozproszonego, zorientowanego zdarzeniowo.

Projektowany język ma umożliwiać łatwe rozproszenie aplikacji na wiele heterogenicznych maszyn cechujących się różnymi architekturami, dostępnymi peryferiami sprzętowymi oraz aplikacjami. Dodatkowo dużą wagę będzie przywiązywał do asynchronicznej obsługi zdarzeń zachodzących w systemie - wykorzystany zostanie mechanizm obsługi zdarzeń (Complex Event Processing, CEP) bazujący na Systemach Regułowych (RBS).

W celu ułatwienia pracy w tak zróżnicowanym środowisku niezbędna będzie jego semantyzacja i udostępnienie odkrytej wiedzy w przystępny sposób. Dodatkowym wymogiem jest łatwa integracja z obecnym w języku mechanizmem obsługi zdarzeń, uzupełnienie jego funkcjonalności i umożliwienie nie tylko reagowania na pojawienie się nowej wiedzy o systemie, ale również refleksji nad dotychczas odkrytą wiedzą.

2 Analiza problemu

Projektowany język zostanie wyposażony w system modułowy, którego zadaniem będzie abstrakcja pewnych funkcjonalności dostępnych na danej platformie sprzętowej bądź programowej, składający się z kilku kluczowych elementów przedstawionych na poniższym diagramie wraz z głównymi relacjami w jakich występują:



Elementy te pełnią następujące funkcje:

- **protokoły** - zapewniają interfejs oraz kontrakty (typy, testy, pre/post-conditions), są przechodnie (wszystko co definiuje odpowiedni interfejs jednocześnie implementuje protokół),
- **moduły** - zapewniają konkretną implementację protokołów (definicje funkcji, dodatkowe kontrakty i testy),
- **funkcje** - stanowią najmniejsze bloki budulcowe programów,
- **procesy** - stanowią kontekst uruchomieniowy jednego z wielu wątków programu (powiązane bloki pamięci, kolejkę asynchronicznie odebranych wiadomości, kontynuacje aktualnie uruchamianej funkcji oraz inne metadane),
- **jednostki** - wiążą statyczne moduły z dynamicznymi procesami zapewniając stan pomiędzy wywołaniami funkcji pewnego protokołu.

Głównymi zadaniami powyższego systemu modułowego jest zapewnienie dystrybucji i redundancji dostępnych funkcjonalności oraz ułatwienie tworzenia dynamicznych aplikacji:

- **dystrybucja** - odwołania do procesów wykorzystują dodatkowy poziom pośredni w postaci **identyfikatora procesu**, dzięki czemu możliwe jest odwoływanie się do procesów działających na innej maszynie,
- **redundancja** - w jednej chwili w systemie (tj. na różnych maszynach połączonych w klaster) może istnieć wiele modułów implementujących te same protokoły na różne sposoby,
- **dynamizm** - aplikacje zbudowane będą z jednostek, które w dowolnej chwili mogą być uruchamiane, zatrzymywane lub przenoszone na inne maszyny.

Dystrybucja, redundancja i dynamizm tworzą potrzebę rozróżniania dostępnych funkcjonalności nie tylko po ich nazwach lub interfejsach, jak ma to miejsce w większości obecnie stosowanych języków programowania, ale także po innych, arbitralnych i zależnych od konkretnej funkcjonalności cechach, takich jak złożoność obliczeniowa jej implementacji, dokładność odczytu danych sensorycznych, czy obciążenie sieci, w której działa jedna z maszyn podłączonych do klastra.

2.1 Przykład zastosowania systemu modułowego

Poniżej przedstawiono prosty przykład zastosowania opisanego powyżej systemu modułowego. Przykład definiuje prosty protokół `gps` pozwalający na pobieranie aktualnej lokacji (funkcja `get-location`) z maksymalną dozwoloną tolerancją (`tolerance`):

```
(define-protocol gps
  (declare (get-location pid)
    (@ tolerance 0.01)))
```

...oraz dwie jednostki `basic-gps` oraz `vendor-gps` go implementujące:

```
(define-module (basic-gps)
  (provide gps)
  (provide unit)

  ;; Unit specific
  (define (start _) nil)

  (define (stop reason) 'ok)

  (define (loop curr-location)
    (loop
     (receive
         (`(get-location ,from) (send! from curr-location))
         (after 1000 (update-location)))))

  ;; GPS specific
  (define (get-location pid)
    (send! `(get-location (self)))
    (receive))

  (define (update-location)
    ;; read & return GPS location
    ))
```

…które różnią się jedynie toleracją funkcji odpowiedzialnej za odczyt danych sensorycznych:

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

  ;; ... details same as basic-gps

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

  (define (update-location)
    ;; read & return better GPS location
    ))
```

Przykładowy scenariusz wykorzystania powyższych jednostek mógłby przebiegać następująco:

- Maszyna `N` posiada wiedzę o działających jednostkach w pozostałych maszynach podłączonych do klastra.
- Na maszynie `N` startuje nowa jednostka `U` wymagająca funkcji lokalizacyjnej.
- Jednostka `U` odkrywa wiedzę o istnieniu dwóch jednostek dostarczających funkcje lokalizacji: `basic-gps` oraz `vendor-gps`, z których `vendor-gps` działa na zdalnej maszynie.
- Jednostka `U` rozstrzyga o wyborze odpowiedniej jednostki na podstawie `tolerance` dostarczanych funkcji lokalizacji oraz fizycznych lokacji jednostek, wybierając `vendor-gps`.
- Jednostka `U` deklaruje regułę, iż w przypadku zatrzymania `vendor-gps` powinna się przełączyć na `basic-gps`.
- Maszyna `P`, na której działa `vendor-gps` ginie w pożarze domu starców.
- Maszyna `N` nie otrzymawszy odpowiedzi od maszyny `P` na pakiet *heartbeat* uznaje ją za niefunkcjonalną i inwaliduje całą wiedzę z nią związaną.
- Jednostka `U` otrzymawszy informację o dezintegracji `vendor-gps` automatycznie przełącza się na `basic-gps`.
- System działa długo i szczęśliwie.

2.2 Wykorzystanie wiedzy w języku

Projektowany język programowania będzie posiadał zintegrowany mechanizm obsługi zdarzeń, zaimplementowany jako system regułowy, korzystający z bazy faktów reprezentujących wiedzę o działającej aplikacji.

Faktem może być dowolna `n`-krotka przynajmniej dwóch wartości, która zostaje dodana do systemu przez procedurę `assert!` (stała asercja), lub `signal!` (tymczasowa sygnalizacja):

```
(predicate subject object-a object-b ...)
(is-a X unit)
(provides X unit-protocol)
(unit-started X)
```

Fakty te mogą reprezentować strukturę dostępnych modułów, jednostek lub protokołów, a także zachodzenie pewnych zdarzeń, takich jak wystartowanie jednostki, załadowanie modułu, lub podłączenie nowej maszyny do klastra.

Aby wykorzystać tak odkrywaną wiedzę o strukturze i działaniu aplikacji należy zdefiniować reguły za pomocą konstrukcji `whenever`:

```
(whenever pattern
          action)

(whenever (and (unit-started ?x)
               (provides ?x ?some-functionality))
          (start-using ?some-functionality))
```

Mechanizm taki pozwala korzystać ze zdobytej wiedzy w sposób **proaktywny** - zdefiniowane reguły zostają uruchomione (wykonane zostają związane z nimi akcje) w momencie zaistnienia odpowiedniej sytuacji - w momencie dodania do bazy wszystkich faktów wymaganych do spełnienia lewej strony reguły.
Nie jest jednak możliwe korzystanie ze zgromadzonej wiedzy w sposób **reaktywny** - nie można tworzyć dowolnych zapytań do bazy faktów - utrudniając implementację wielu pożądanych mechanizmów, takich jak odkrywanie usług (**service discovery**), czy dopasowywanie usług (**service matching**).

W związku z powyższym, niemożliwe jest zrealizowanie trzeciego punktu przytoczonego w poprzedniej sekcji przypadku użycia dotyczącego odkrywania wiedzy o strukturze systemu przez nowo-wystartowaną jednostkę - ponieważ fakty tworzone przy startowaniu jednostek `basic-gps` oraz `vendor-gps` zostały już przetworzone prez mechanizm, nowo-wystartowana jednostka nie ma możliwości ponownego ich odkrycia i wykorzystania do własnych celów.

Problemem do rozwiązania jest zatem **umożliwienie refleksji nad bazą powiązanych ze sobą faktów w sposób wydajny i wygodny w użytkowaniu** - na przykład wykorzystując język zapytań podobny do języków zapytań relacyjnych baz danych z rodziny SQL - przy jednoczesnym zachowaniu kompatybilności z mechanizmem obsługi zdarzeń i uzupełnieniu jego funkcjonalności o dodatkowe operacje na bazie faktów.

2.3 Podobne rozwiązania

- Przegląd obecnych metod SOA - statyczne formaty definicji usług, brak elastyczności w odkrywaniu usług,
- Call by Meaning - nieadekwatna wydajność.

3 Szkic rozwiązania

3.1 Porównanie różnych podejść

- Predefiniowane formaty opisu usług (np. WSDL)
- Systemy Regułowe - problem późnego startu/restartu usług
- Ontologie/bazy faktów

3.2 Podejście semantyczne

- Wstępny plan ontologii
- Lista pojęć i zależności między nimi

4 Prototyp rozwiązania

Link do repozytorium na GitHub.

4.1 Ontologia Systemu Modułowego

4.2 Przykłady wykorzystania ontologii

- Automatyczna inferencja dostępnej klasy obiektów
- Zapytania SPARQL

5 Analiza proponowanego rozwiązania

5.1 Wnioski

6 Bibliografia

- Hesam Samimi, Chris Deaton, Yoshiki Ohshima, Alessandro Warth, and Todd Millstein, *Call by Meaning*, In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014), ACM, New York, NY, USA, 11-28, `http://doi.acm.org/10.1145/2661136.266115`