Wykorzystanie ontologii w językach programowania do programowania rozproszonego.

Kajetan Rzepecki

EIS 2014
8 lutego 2015

1 Wstęp

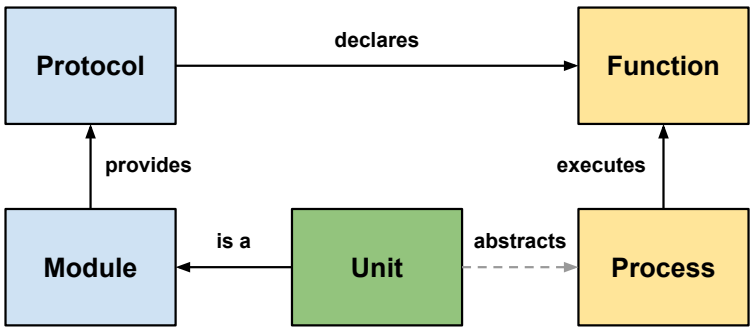
Celem projektu jest zbadanie możliwości oraz opłacalności wykorzystania ontologii w celu reprezentacji wiedzy w języku programowania przeznaczonego do programowania rozproszonego, zorientowanego zdarzeniowo.

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 Sposoby rozwiązania problemu

Ponieważ głównym przypadkiem użycia wykorzystującym bazę faktów w opisany w poprzedniej sekcji sposób jest odkrywanie istniejących usług (**service discovery**) oraz wybór najbardziej pasującej usługi (**service matching**), najłatwiejszym i zarazem najbardziej logicznym wyborem jest wykorzystanie istniejących i sprawdzonych protokołów oraz formatów umożliwiających odkrywanie usług.

Rozwiązanie takie jest efektywne, lecz mało elastyczne - poza odkrywaniem i dopasowywaniem usług nie umożliwia implementacji innych potencjalnie przydatnych mechanizmów. Dodatkowo, najczęściej stosowane protokoły SD przeważnie są bardzo rozbudowane i obejmują wiele aspektów aplikacji biznesowych, a co za tym idzie nie stanowią prostej, ortogonalnej funkcjonalności języka programowania.

Innym podejściem jest wykorzystanie bazy faktów w niezmienionej postaci oraz mechanizmu wnioskującego ją wykorzystującego. Mechanizm taki został wykorzystany w eksperymentalnym języku Novä opisanym szczegółowo w *Call by Meaning*, gdzie posłużono się bazą wiedzy użytkowej (**common sense knowledge**) Cyc. Programy w języku Novä wymagają dodania semantycznych adnotacji opisujących ich strukturę oraz działanie, natomiast Cyc wykorzystywany jest do poznawania wzajemnych zależności między poszczególnymi usługami oraz ich dopasowywania.

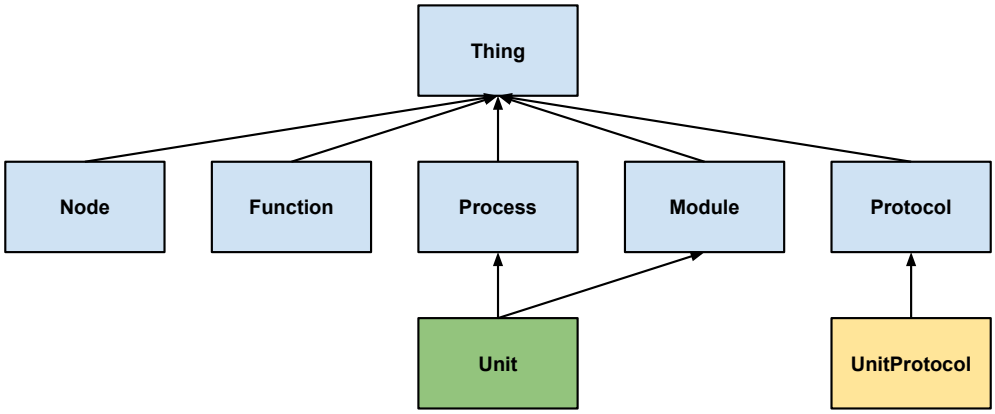
Rozwiązanie to jest bardzo elastyczne, ale wymaga długich opisów semantycznych poszczególnych elementów programu. Dodatkowo wykorzystywanie olbrzymiej bazy wiedzy użytkowej przy każdej próbie skorzystania z wiedzy o samym programie skutkuje znaczącą degradacją wydajności.

Podejściem pośrednim, pozwalającym osiągnąć zadowalającą wydajność przy dużej ekspresywności i elastyczności jest wykorzystanie wyspecjalizowanej ontologii obejmującej system modułowy języka oraz bazy faktów **triple store** pozwalającej konstruować zapytania w języku SPARQL.

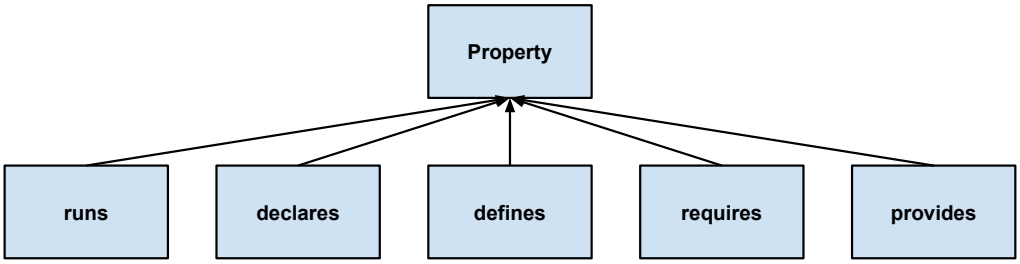
Rozwiązanie takie umożliwi implementację wielu pożądanych mechanizmów i jednocześnie pozostanie ortogonalne do pozostałych funkcjonalności projektowanego języka. Zostało ono opisane w następnej sekcji.

3 Szkic rozwiązania

Podejście ontologiczne wymaga uprzedniego zidentyfikowania klas obiektów w niej występujących oraz relacji w jakich się one znajdują. Ponieważ opisany w poprzednich sekcjach system modułowy jest w dużej mierze prostą taksonomią, wstępna identyfikacja klas została podsumowana na poniższym diagramie:



Natomiast wymagane pojęcia zaprezentowano poniżej:



Zidentyfikowanym klasom przypisano następujące semantyczne definicje i najważniejsze atrybuty (zapisane w nawiasie po nazwie klasy):

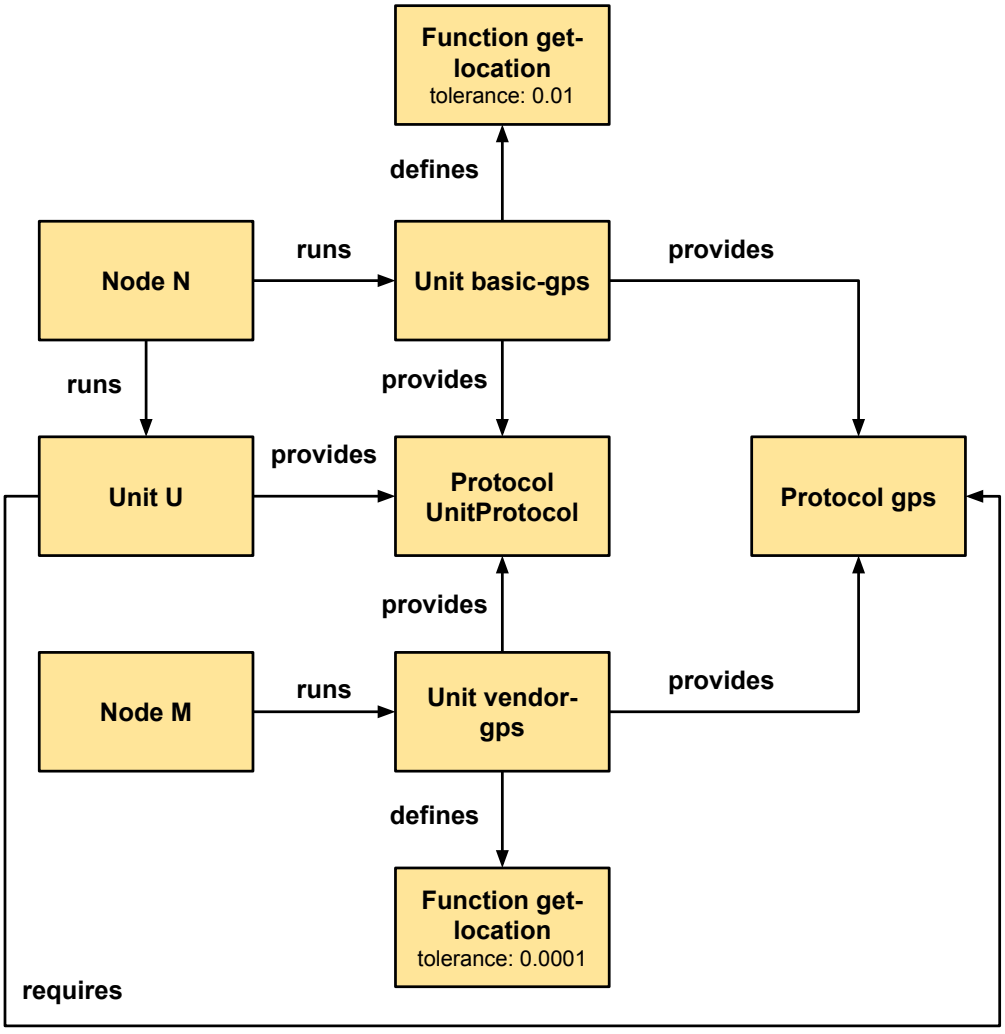
- **Node (id)** - węzeł, na którym działa (**runs**) `n` procesów (**Process**) i jednostek (**Unit**),
- **Function (name)** - definicja funkcji,
- **Process (id)** - process, który wykonuje można identyfikować na podstawie **process id**,

- **Unit** - specjalna klasa, która jednocześnie jest procesem (**Process**) oraz modułem (**Module**),
- **Module** - moduł, który może definiować (**defines**) pewne funkcje (**Function**), zależeć od (**requires**) protokołów (**Protocol**) lub innych modułów (**Module**) oraz dostarczać (**provides**) pewną funkcjonalność (**Protocol**),
- **Protocol** - protokół, który deklaruje (**declares**) `n` funkcji (**Function**),
- **UnitProtocol** - wyróżniony protokół (instancja klasy **Protocol**) dostarczająca wbudowanego w język interfejsu jednostek (**Unit**).

Natomiast relacjom:

- **Node runs Process** - działanie procesu na danym węźle,
- **Protocol declares Function** - przynależność funkcji do danego protokołu,
- **Module defines Function** - istnienie implementacji funkcji w pewnym module,
- **Module requires Protocol or Module** - wzajemna zależność pomiędzy modułami w systemie,
- **Module provides Protocol** - dostarczanie pewnej funkcjonalności przez moduł.

Na poniższym diagramie przedstawiono kilka przykładowych instancji powyższych klas wraz z ich wzajemnymi relacjami na podstawie przykładu z poprzedniej sekcji:



3.1 Scenariusze wykorzystania ontologii

W celu weryfikacji ontologii zaproponowano trzy scenariusze jej wykorzystania (wymienione poniżej). Każdy scenariusz zostanie wykonany na przykładowej bazie wiedzy zbudowanej z przykładowych instancji klas ontologii.

- **Lista jednostek na danych węzłach** - scenariusz polega na odkryciu przez nowowystartowaną jednostkę listy innych jednostek, które są dostępne na poszczególnych węzłach klastra,
- **Jednostka dostarczająca GPS o najniższej tolerancji błędu** - scenariusz polega na dopasowaniu dwóch usług bazując jedynie na ich wzajemnej zależności oraz atrybutach,
- **Dodanie klasy Server** - scenariusz polega na odkryciu wszystkich jednostek, które można dopasować do wygenerowanej podczas działania aplikacji (na przykład programatycznie na podstawie interfejsu) klasy reprezentującej serwer jedynie na podstawie funkcji, które definiują; jednostki nie powinny zostać zmodyfikowane (na przykład poprzez dodanie protokołu, który dostarczają).

4 Prototyp rozwiązania

Ponieważ opisana w poprzedniej sekcji i udostępniona w internecie ontologia jest bardzo prosta, została ona wzbogacona o dodatkowe klasy pojęć oraz ich instancje stworzone na podstawie przykładowego kodu aplikacji, napisanej w projektowanym języku programowania.

Celem aplikacji jest automatyczne zbieranie danych o temperaturze w różnych miejscach klastra oraz obliczanie statystyk na podstawie zdobytych danych. Aplikacja jest rozproszona i działa na trzech węzłach, z których każdy uruchamia kilka jednostek i dostarcza kilku modułów:

- **smartphone**, uruchamiający `basic-timer` oraz dostarczający `basic-stats`:

```
(define-protocol timer
  (declare (get-time)))

(define-protocol (stats)
  (declare (compute-stats time-series temp-vals)))

(define-module (basic-timer)
  (provide 'protocols.timer)
  (provide 'unit)

  (define (get-time)
    ;; Get unprecise time...
    )

  (define (start type)
    ;; ...
    )

  (define (loop state)
    ;; Dispatch requests, return current wall-clock time, repeat.
    )
```

```
  (define (stop reason)
    ;; ...
    ))

(define-module (basic-stats)
  (provide 'protocols.stats)

  (define (compute-stats time-series temp-vals)
    ;; Slowly compute stats...
    ))
```

- **raspberry pi**, uruchamiająca `RTC-timer` oraz `DS18B20-temp`:

```
(define-protocol temp
  (declare (get-temp)))

(define-module (DS18B20-temp)
  (provide 'temp)
  (provide 'unit)

  (define (get-temp)
    ;; Get wacky temp using DHT11 sensor...
    )

  (define (start type)
    ;; Initialize DS18B20...
    )

  (define (loop state)
    ;; Start temp conversion, read temp, repeat...
    )

  (define (stop reason)
    ;; ...
    ))

(define-module (RTC-timer)
  (provide 'protocols.timer)
  (provide 'unit)

  (define (get-time)
    ;; Get RTC time...
    )

  (define (start type)
    ;; Initialize the RTC...
    )

  (define (loop state)
    ;; Dispatch requests, return RTC time, repeat...
```

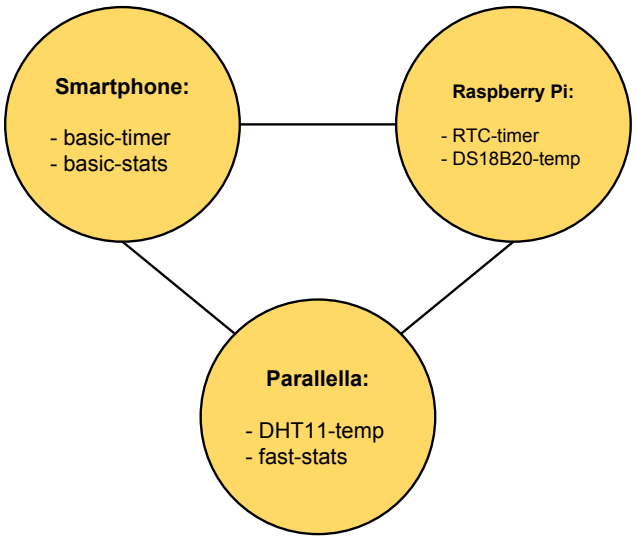
```
    )

  (define (stop reason)
    ;; ...
    ))
```

- **parallella**, uruchamiająca `DHT11-temp` oraz dostarczająca `fast-stats`:

```
(define-protocol other-temp
  (declare (get-temp)))

(define-module (DHT11-temp)
  (provide 'other-temp)
  (provide 'unit)

  (define (get-temp)
    ;; Get wacky temp using DHT11 sensor...
    )

  (define (start type)
    ;; Initialize DHT11...
    )

  (define (loop state)
    ;; Dispatch requests, read humidity & temp, return temp, repeat...
    )

  (define (stop reason)
    ;; ...
    ))

(define-module (fast-stats)
  (provide 'protocols.stats)

  (define (compute-stats time-series temp-vals)
    ;; Compute stats faster...
    ))
```

Należy zauważyć, iż `DHT11-temp` działające na węźle **parallella** oraz `DS18B20-temp` działające na węźle **raspberry pi** nie używają tego samego protokołu. Fakt ten zostanie wykorzystany w dalszej części analizy.

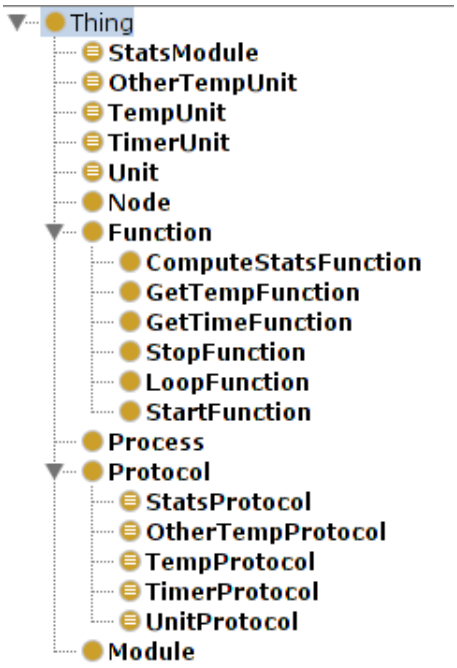
Schemat architektury przykładowej aplikacji został zawarty na poniższym diagramie:



4.1 Ontologia Systemu Modułowego

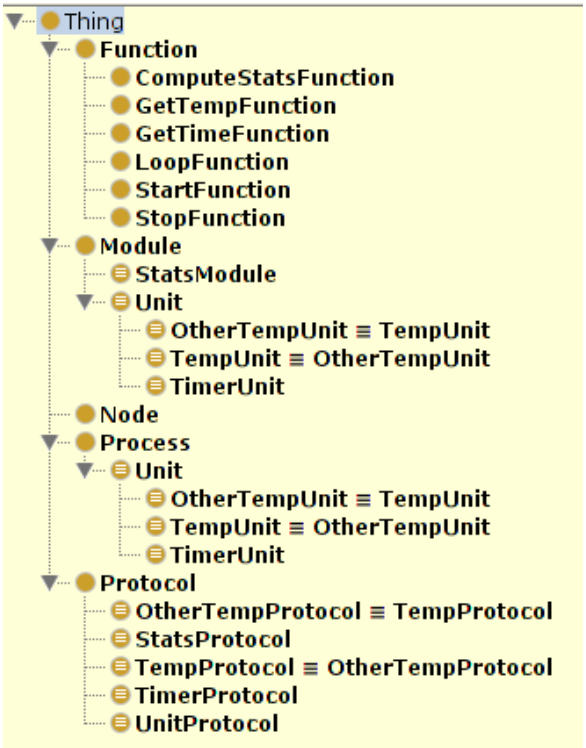
Ontologia została stworzona z wykorzystaniem programu Protégé. Poniższy diagram prezentuje strukturę klas ontologii po wnioskowaniu:

Ontologia udostępnia poniższe klasy. Poza klasami zidentyfikowanymi i opisanymi w poprzednich sekcjach, zawarto w niej także klasy niezbędne do reprezentacji modułów przykładowej aplikacji:



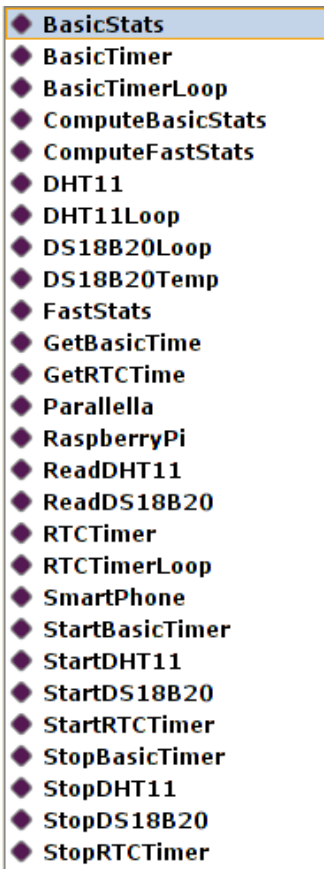
Należy zwrócić uwagę, iż reprezentacja modułów przez klasy w ontologii, nie zaś instancje klasy `Module`, jest uzasadniona możliwością istnienia wielu wersji tego samego moduły na wielu węzłach. W takim przypadku każda wersja reprezentowana jest przez osobną instancję klasy w ontologii.

Mechanizm wnioskujący dostępny w Protégé pozwala na automatyczną inferencję wzajemnych zależności klas na podstawie ich ograniczeń. Jest to bardzo wygodny i pomocny mechanizm, który ma duży potencjał zastosowania w systemie modułowym języka programowania:

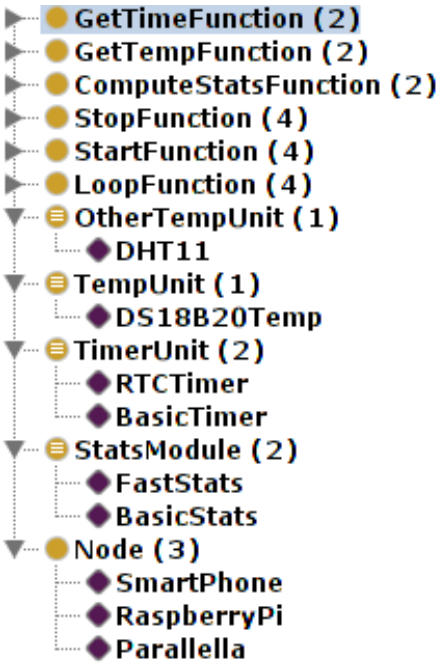


Pozwala on na ”unormowanie”klas - automatycznie wykrywana jest ich hierarchia oraz relacje `subClassOf`. Cieakwym jest także wykrywanie klas wzajemnie równoważnych (na przykład `TempProtocol` oraz `OtherTempProtocol`), co zostanie opisane szerzej w następnym punkcie.

Konkretną reprezentację wiedzy o przykładowej aplikacji stanowią instancje klas:



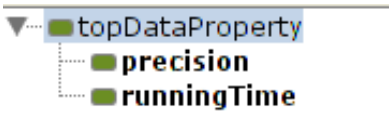
Reprezentują one poszczególne moduły oraz jednostki działające w danej chwili w aplikacji:



Klasy pojęć w ontologii powiązane są następującymi relacjami:

Relacje te w większości zostały opisane w poprzednich sekcjach, jednak koniecznym okazało się dodanie relacji `executes`, która wiąże procesy z funkcjami. Umożliwiło to powiązanie klass `Unit` oraz `Process` - są to klasy reprezuntujące ciągłość obliczeń w systemie - oraz ułatwiło tworzenie ontologii.

Poza relacjami między klasami, zdefiniowane także kilka atrybutów klas:



Atrybut `precision` określa precyzję czasu zwracanego przez funkcje z klasy `GetTimeFunction`, natomiast `runningTime` określa czas działania funkcji z klasy `ComputeStatsFunction`. Są to jedynie przykładowe atrybuty i istnieje możliwość dodania znacznie większej ich ilości.

4.2 Przykłady wykorzystania ontologii

- Najciekawszym przykładem wykorzystania ontologii jest **automatyczna inferencja ekwiwalentnych klas** na podstawie implementowanych interfejsów. Pozwala to na uwolnienie programistów oraz modułów języka programowania od statycznego określania dostarczanych protokołów i implementowanych interfejsów (na przykład `DHT11-temp` oraz `DS18B20-temp`). Zamiast tego, moduł definiujący pewien zbiór funkcji może zostać automatycznie dopasowany do odpowiednich protokołów i zostać wykorzystany przez inne moduły, wymagające implementacji owych protokołów. Programista zamiast definiować protokoły, które jego kod będzie udostępniał może definiować protokoły, których jego kod wymaga, a mechanizm wnioskowania automatycznie dopasuje odpowiednie moduły, które spełniają wymogi zdefiniowanych protokołów.

- **Lista jednostek działających na danych węzłach** - pozwalające na wykrywanie dostępnych usług:

Zapytanie SPARQL:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mso: <http://www.idorobots.org/mso#>
SELECT *
WHERE { ?node mso:runs ?unit }
ORDER BY ?node
```

Wynik:

?node	?unit
Parallella	DHT11
RaspberryPi	DS18B20Temp
RaspberryPi	RTCTimer
SmartPhone	BasicTimer

- **Jednostka dostarczająca czas o najwyższej precyzji** - pozwalające na dopasowywanie usług:

Zapytanie SPARQL:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX mso: <http://www.idorobots.org/mso#>
SELECT *
WHERE {
    ?unit a mso:TimerUnit.
    ?unit mso:defines ?function.
    ?function a mso:GetTimeFunction.
    ?function mso:precision ?precision
}
ORDER by ?precision
```

Wynik:

?unit	?function	?precision
RTCTimer	GetRTCTime	100
BasicTimer	GetBasicTime	25

5 Analiza proponowanego rozwiązania

5.1 Wnioski

- Do najważniejszych wniosków płynących z projektu należy fakt, iż ontologie stanowią ciekawe narzędzie inżynierii wiedzy, które można z powodzeniem wykorzystać w implementacji języków programowania jako ich integralną część.
- Wykorzystanie ontologii umożliwia implementację odkrywania i dopasowywania usług za pomocą zapytań do bazy inferowanej wiedzy - programista zostaje zwolniony z zawierania óczywistych"faktów w kodzie aplikacji.
- Mechanizm wnioskujący dostępny w narzędziach przetwarzających ontologie umożliwia łatwe rozwijanie i hierarchizowanie wiedzy o strukturze aplikacji, a język zapytań SPARQL dodatkowo pozwala tworzyć ustrukturyzowane zapytania, których wynik jest przystępny do dalszego przetwarzania.
- Wadą ontologii jest wykorzystywanie **trójek** lub **czwórek** do reprezentacji faktów. Wymusza to ograniczenie reprezentacji wiedzy istniejącego już w projektowanym języku programowania mechanizmu obsługi zdarzeń bazującego na systemach regułowych.
- Kolejną wadą wykorzystania ontologii jest znaczna ilość dodatkowych faktów, które istnieją w bazie a pochodzą z licznych innych ontologii, z których MSO korzysta.

5.2 Potencjalne kierunki rozwoju

- Dodanie większej ilości rozróżniających atrybutów do podklas klasy `Function`, co umożliwi lepsze dopasowywanie usług.
- Dodanie klas relacji czasowych oraz przestrzennych do węzłów w celu umożliwienia kontroli **Quality of Service**.
- Dodanie podklas klasy `Node` reprezentujących specjalne- i pseudo-węzły do różnych zastosowań.

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.
- Douglas Foxvog, *Cyc*, In Theory and Applications of Ontology: Computer Applications, Springer, 2010.
- Natalya F. Noy and Deborah L. McGuinness, *Ontology Development 101: A Guide to Creating Your First Ontology*, Stanford University, Stanford, CA, 94305.