Politechnika Warszawska

Wydział Matematyki i Nauk Informacyjnych

Reprezentacja wiedzy

Programy działań z efektami domyślnymi

Autorzy:

Dragan Łukasz
Flis Mateusz
Fusiara Marcin
Izert Piotr
Pielat Mateusz
Rząd Przemysław
Siry Roman
Waszkiewicz Piotr
Zawadzka Anna

18 kwietnia 2016

1 Opis zadania

Zadaniem projektu jest opracowanie i zaimplementowanie języka akcji dla specyfikacji podanej klasy systemów dynamicznych oraz odpowiadający mu język kwerend.

System dynamiczny spełnia podane założenia:

1. Prawo inercji
2. Niedeterminizm i sekwencyjność działań
3. Pełna informacja o wszystkich akcjach i wszystkich ich skutkach bezpośrednich
4. Z każdą akcją związany jest:
   (a) Warunek początkowy (ew. true)
   (b) Efekt akcji
   (c) Jej wykonawca
5. Skutki akcji:
   (a) Pewne (zawsze występują po zakończeniu akcji)
   (b) Domyślne (preferowane. Zachodzą po zakończeniu akcji, o ile nie jest wiadomym, że nie występują)
6. Efekty akcji zależą od jej stanu, w którym akcja się zaczyna i wykonawcy tej akcji
7. W pewnych stanach akcje mogą być niewykonalne przez pewnych (wszystkich) wykonawców

Programem działań nazywać będziemy ciąg $((A_1, W_1), (A_2, W_2), \ldots, (A_n, W_n))$, gdzie A_i jest akcją, zaś W_i jej listą wykonawców postaci $W_i = (w_1, w_2, \ldots, w_n), n = 0, 1, 2, \ldots$ gdzie dla $n = 0$ $W_i = \epsilon$ oznacza dowolnego wykonawcę.

Język kwerend zapewnia uzyskanie odpowiedzi na następujące pytania:

1. Czy podany program działań jest wykonywalny zawsze/kiedykolwiek?
2. Czy wykonanie podanego programu działań z dowolnego stanu spełniającego warunek π prowadzi zawsze/kiedykolwiek/na ogół do stanu spełniającego warunek celu γ ?
3. Czy z dowolnego stanu spełniającego warunek π cel γ jest osiągalny zawsze/kiedykolwiek/na ogół?
4. Czy wskazany wykonawca jest zaangażowany w realizację programu zawsze/kiedykolwiek?

2 Język akcji Ω

2.1 Definicja języka

Ω jest rodziną języków, w której każdy język $\mathcal{L}$ określony jest nad sygnaturą

$$\Upsilon = (F, A, V)$$

gdzie:

- F - niepusty zbiór zmiennych (fluenty)
- A - niepusty zbiór akcji
- V - niepusty zbiór wykonawców (aktorów), przy czym $\epsilon \in V$, gdzie ϵ oznacza kogokolwiek

2.2 Syntaktyka języka

Przed przystąpieniem do opisu typów zdań doprecyzowane zostanie oznaczenie listy wykonawców $W \in V; W = (w_1, w_2, \ldots, w_n)$ które reprezentuje listę dopuszczalnych wykonawców danej akcji. Dodatkowo możliwy jest zapis *negatywny* w postaci $W = (\neg w_1, \neg w_2, \ldots, \neg w_n)$ oznaczający wyłączenie wymienionych wykonawców z możliwości wykonywania danej akcji. Niedopuszczalny jest zapis łączący obie te formy, postaci $W = (w_1, w_2, \ldots, w_i, \neg w_{i+1}, \ldots, \neg w_n)$
Dodatkowo przez α oznaczana będzie dowolna kombinacja zmiennych (fluentów):

$$\alpha = f|\alpha|\neg\alpha|\alpha_1 \wedge \alpha_2|\alpha_1 \vee \alpha_2|\alpha_1 \rightarrow \alpha_2|\alpha_1 \leftrightarrow \alpha_2$$

W języku Ω występują następujące typy zdań:

- $\texttt{initially}\ \alpha$
  formuła α zachodzi w stanie początkowym
- $\alpha\ \texttt{after}\ (A_1, W_1), ..., (A_n, W_n)$
  formuła α zachodzi po wykonaniu sekwencji $(A_1, W_1), ..., (A_n, W_n)$, gdzie A_i jest akcją, zaś W_i jej niepustą listą wykonawców
- $(A, W)\ \texttt{causes}\ \alpha\ \texttt{if}\ \pi$
  skutkiem wykonania akcji A przez któregokolwiek z wykonawców W w stanie spełniającym warunek π jest stan, w którym spełniona jest formuła α. Przy czym:
  - Jeśli $\pi \Leftrightarrow \top$, to powyższe zdanie zapisujemy:
    $(A, W)\ \texttt{causes}\ \alpha$
    oznacza to, że skutkiem wykonania akcji A przez któregokolwiek z wykonawców W jest stan, w którym spełniona jest formuła α

- Jeśli $\alpha \Leftrightarrow \perp$, to powyższe zdanie zapisujemy:
  `impossible` (A, W) `if` π
  co oznacza, że niemożliwe jest wykonanie akcji A przez któregokolwiek z wykonawców W w stanie spełniającym warunek π

- `observable` α `after` $(A_1, W_1), ..., (A_n, W_n)$
  po wykonaniu sekwencji $(A_1, W_1), ..., (A_n, W_n)$, gdzie A_i jest akcją, zaś W_i jej potencjalnymi wykonawcami, w stanie początkowym może (ale nie musi) zachodzić formuła α

- (A, W) `releases` f `if` π
  wykonanie akcji A przez któregokolwiek z wykonawców W w stanie spełniającym warunek π może (ale nie musi) zmienić wartość zmiennej f

- (A, W) `typically causes` α `if` π
  skutkiem wykonania akcji A przez któregokolwiek z wykonawców W w stanie spełniającym warunek π na ogół jest stan, w którym spełniona jest formuła α

- `typically` α `after` $(A_1, W_1), ..., (A_n, W_n)$
  formuła α na ogół zachodzi po wykonaniu sekwencji $(A_1, W_1), ..., (A_n, W_n)$, gdzie A_i jest akcją, zaś W_i listą jej potencjalnych wykonawców

- `always` α
  formuła α jest spełniona w każdym stanie

- (A, W) `preserves` f `if` π
  wskutek wykonania akcji A przez któregokolwiek z wykonawców W w stanie spełniającym warunek π zmienna f nie może ulec zmianie

2.3 Przykłady

2.3.1 Gotujący John

John jest entuzjastą gotowania i bardzo lubi jeść. Zakładamy, że początkowo jest głodny, jego lodówka jest pusta, nie ma on żadnego gotowego posiłku i w jego kuchni panuje porządek. Jeśli John jest głodny, to jest też zły. Nie może jednak nic zjeść, jeśli nie ma co. Niemożliwym jest także ugotowanie posiłku, jeśli lodówka Johna jest pusta. Aby ją uzupełnić John idzie na zakupy. Zakupy również często powodują, że John ma lepszy humor. John może teraz coś ugotować, lecz jeśli jest zły to na ogół powoduje to ogromy chaos w kuchni. Zdolności kulinarne Johna są na tyle duże, że zawsze wyjdzie mu posiłek, który jest zdatny do jedzenia. John lubi duże porcje, więc możliwe, że po ugotowaniu posiłku znowu ma pustą lodówkę. Jest też takim łakomczuchem, że nie zostawia sobie nic na później i zjada cały posiłek. Po zjedzeniu głód mija, a humor Johna się poprawia.

```
initially hungry ∧ angry
initially emptyFridge ∧ ¬hasMeal ∧ ¬chaos
impossible (Eat, John) if ¬hasMeal
impossible (Cook, John) if emptyFridge
(Shop, John) causes ¬emptyFridge
(Shop, John) releases angry if angry
(Cook, John) typically causes chaos if angry
(Cook, John) causes hasMeal
(Cook, John) releases emptyFridge
(Eat, John) causes ¬hasMeal ∧ ¬hungry ∧ ¬angry
```

2.3.2 Programiści Fred i Bill

Farmer Bill i indyk Fred pracują razem nad pewnym projektem programistycznym. Zakładamy, że początkowo kod jest czytelny i kompilowalny. Bill to niedoświadczony programista, więc gdy dopisze on jakiś fragment cały kod przestaje być czytelny, a nierzadko przestaje się też kompilować. Indyk Fred jest z kolei weteranem branży IT, więc jego kod kompiluje się zawsze (gdy pracuje on z czytelnym kodem) lub prawie zawsze (gdy kod jest nieczytelny). W razie potrzeby Fred refaktoryzuje cały kod, dzięki czemu poprawia się jego czytelność. Bill i Fred zgodnie ustalili, że nie będą dopisywać nowych fragmentów kodu jeżeli dotychczasowy się nie kompiluje. W takim wypadku któryś z nich musi go najpierw zdebugować (co potrafi każdy programista mając odpowiednio dużo czasu).

```
initially compiles ∧ cleanCode
(Code, Bill) causes ¬cleanCode
(Code, Bill) releases compiles if compiles
(Code, Fred) causes compiles if cleanCode
(Code, Fred) typically causes compiles if ¬cleanCode
(Refactor, Fred) causes cleanCode
(Debug, ϵ) causes compiles
impossible (Code, ϵ) if ¬compiles
```

2.3.3 Alicja w krainie czarów

Alicja może być pomniejszona, lub być swojego normalnego, wysokiego wzrostu. Początkowo Alicja jest wysoka. Wypicie Eliksiru pomnniejsza Alicję, natomiast zjedzenie Ciastka ją powiększa.
Kot z Cheshire nigdy nie zje Ciastka, a wypicie przez niego Eliksiru zwykle go ujawnia, jeśli jest niewidzialny. Kot początkowo jest niewidzialny.
Kapelusznik początkowo nie jest szalony. Wypicie Eliksiru może doprowadzić go do szaleństwa, a zjedzenie Ciastka zwykle powoduje powrót do zdrowych zmysłów.
Ciastko jest tylko jedno więc znika po jego zjedzeniu i nie można próbować go

zjeść, natomiast duża, nieprzeźroczysta butelka Eliksiru nie musi być pusta po napiciu się z niej.
Biały Królik nigdy nie zje ciastka, a wypicie przez niego Eliksiru powoduje magiczne pojawienie się Ciastka z powrotem.
Początkowo Ciastko jest dostępne, a butelka pełna Eliksiru.

```
initially ¬aliceSmall ∧ ¬hatterMad ∧ ¬catVisible
initially cakeExists ∧ elixirExists
(DRINK, Alice) causes aliceSmall if elixirExists
(EAT, Alice) causes ¬aliceSmall
impossible (EAT, Cat)
(DRINK, Cat) typically causes catVisible if elixirExists
(DRINK, Hatter) releases hatterMad if ¬hatterMad ∧
    elixirExists
(EAT, Hatter) typically causes ¬hatterMad if hatterMad
(EAT, ϵ) causes ¬cakeExists
impossible (EAT, ϵ) if ¬cakeExists
(DRINK, ϵ) releases elixirExists if elixirExists
impossible (EAT, Rabbit)
(DRINK, Rabbit) causes cakeExists if elixirExists
```

2.3.4 Lampka Toma

Student Tom ma na biurku lampkę, której używa, jeżeli musi wieczorem przygotować się na zajęcia. Gdy Tom zaczyna naukę, lampka jest zapalona. Zadania są trudne i zdarza się, że Tom nie potrafi ich rozwiązać. Wtedy Tom denerwuje się i zrzuca wszystko razem z lampką z biurka. Zrzucenie na podłogę zazwyczaj powoduje, że w lampce zbije się żarówka i wtedy nie może ona świecić. Tom może poza tym włączać i wyłączać lampkę, ale żarówka się od tego nie naprawi.

```
initially lighted
always lighted → ¬broken
(THROW_DOWN,Tom) usually causes broken
(TURN_OFF,Tom) causes ¬lighted
(TURN_ON,Tom) causes lighted
(TURN_ON,Tom) preserves broken
```

2.4 Semantyka języka

2.4.1 Stan

Stanem będziemy nazywać dowolną funkcję $\sigma : F \to \{1, 0\}$, która przypisuje zmiennym wartości logiczne. Jeśli $\sigma(f) = 1$, to znaczy, że zmienna f zachodzi w stanie σ. Funkcję tę można rozszerzyć na zbiór wszystkich formuł nad zbiorem zmiennych F według zasad obowiązujących w klasycznej logice zdań.

2.4.2 Struktura

Strukturą nazywamy układ $S = (\Sigma, \sigma_0, ResAb, ResN)$, gdzie:

- Σ - zbiór stanów
- $\sigma_0 \in \Sigma$ - stan początkowy
- $ResAb, ResN : A \times V \times \Sigma \to 2^\Sigma$ są funkcjami przejść. $ResAb$ jest funkcją przejść nietypowych, $ResN$ jest funkcją przejść typowych oraz $ResAb \cap ResN = \emptyset$

2.4.3 Dziedzina

Niech $\mathcal{L}$ będzie językiem z rodziny Ω. Dziedziną akcji nazywamy niepusty zbiór D zdań języka $\mathcal{L}$.

2.4.4 Model dziedziny

W celu zdefiniowania pojęcia modelu dziedziny wprowadzone zostaną następujące funkcje pomocnicze:

1. $Res_0 : A \times W \times \Sigma \to 2^\Sigma$ konstruowane na podstawie zdań efektów akcji.

   $\forall_{a \in A, w \in W, \sigma \in \Sigma} Res_0(a, w, \sigma) = \{\sigma' \in \Sigma : ((a, w) \textbf{ causes } \alpha \textbf{ if } \pi) \in D \wedge (\sigma \models \pi) \Rightarrow (\sigma' \models \alpha)\}$

   Oznacza to, że Res_0 konstruuje się bez minimalizacji zmian.

2. Funkcję Res^- wyznacza się stosując minimalizację zmian.

3. Funkcję $Res_0^+ : A \times V \times \Sigma \to 2^\Sigma$ spełniającą warunek $\forall_{a \in A, w \in V, \sigma \in \Sigma}$:

   $$Res_0^+(a, w, \sigma) =$$

   $$\{\sigma' \in Res_0(a, w, \sigma) : ((a, w) \textbf{ typically causes } \beta \textbf{ if } \pi) \in D \wedge (\sigma \models \varphi) \Rightarrow (\sigma' \models \beta)\}$$

Niech D będzie dziedziną akcji języka Ω i niech $S = (\Sigma, \sigma_0, ResAb, ResN)$ będzie strukturą dla Ω. Mówimy, że S jest modelem D $\leftrightarrow$ spełnione są warunki:

- Σ jest zbiorem stanów z dziedziny D
- każde zdanie obserwacji i każde zdanie wartości z dziedziny D jest prawdziwe w S
- $\forall_{a \in A, w \in V, \sigma \in \Sigma} ResN(a, w, \sigma)$ jest zbiorem tych wszystkich stanów $\sigma' \in Res_0^+(a, w, \sigma)$, dla których zbiory $New(a, w, \sigma, \sigma')$ są minimalne
- $\forall_{a \in A, w \in V, \sigma \in \Sigma} ResAb(a, w, \sigma) = Res^-(a, w, \sigma) | ResN(a, w, \sigma)$

Warto zwrócić uwagę na to, że skutki *pewne* dla akcji traktowane są jak *typowe*.

2.4.5 Funkcja przejścia

Niech $S = (\Sigma, \sigma_0, ResAb, ResN)$ będzie strukturą dla języka. Konstrukcja funkcji $\Psi_S : (A \times V)^* \times \Sigma \to \Sigma$ wygląda następująco:

- $\Phi_S(a, \epsilon, \sigma) = \sigma$ gdzie ϵ oznacza ciąg pusty
- jeśli $\Phi_S(((a_1, w_1), \ldots, (a_n, w_n)), \sigma)$ jest określona to

$$\Phi_S(((a_1, w_1), \ldots, (a_n, w_n)), \sigma) \in ResAb((a_n, w_n), \Phi_S((a_1, w_1), \ldots, (a_{n-1}, w_{n-1})))$$
$$\cup ResN((a_n, w_n), \Phi_S((a_1, w_1), \ldots, (a_{n-1}, w_{n-1})))$$

2.5 Konstrukcja modeli

Przy wyznaczaniu funkcji przejścia $ResN$ i $ResAb$ stosujemy zasadę zaczerpniętą z języka *ARQ*. Stany niedopuszczalne są eliminowane przed minimalizacją zmian oraz wprowadzone zostaje zdanie *preserves*, które ma na celu ochronę wartości zmiennej przed zmianami przy wykonaniu danej akcji.

```
initially ¬charged ∧ ¬playing
always playing -> charged
(SWITCH_ON, radio) typically causes playing
(SWITCH_ON, radio) preserves charged
```

Zbiór stanów dopuszczalnych to: $\Sigma = \{\sigma_0, \sigma_1, \sigma_2\}$, gdzie:
$\sigma_0 = \{\neg charged, \neg playing\}$, $\sigma_1 = \{charged, \neg playing\}$, $\sigma_2 = \{charged, playing\}$.

stan $\sigma_3 = \{\neg charged, playing\}$ jest niezgodny z dziedziną, więc zostaje wyeliminowany.

$Res_0^-(\text{SWITCH\_ON}, radio, \sigma_0) = \{\sigma_2\}$
$New(\text{SWITCH\_ON}, radio, \sigma_0, \sigma_2) = \{charged, playing\}$
$Res^-(\text{SWITCH\_ON}, radio, \sigma_0) = \{\sigma_2\}$
$ResN(\text{SWITCH\_ON}, radio, \sigma_0) = \{\sigma_2\}$

Jako, że wartość *charged* pomiędzy stanami σ_0 i σ_2 ulega zmianie, przejście to nie może zostać zawarte w funkcji ResN.

$$ResN(SWITCH\_ON, \sigma_0) = \emptyset$$

Jest to logiczne zachowanie, gdyż rozładowane radio nie może zacząć grać po włączeniu.

2.6 Scenariusz

Scenariuszem nazywamy skończony ciąg par akcja-wykonawca:

$$SC = ((A_1, w_1), (A_2, w_2), ..., (A_n, w_n))$$

Przykład:

```
(Shop, John), (Cook, John), (Eat, John)
```

Powyższy scenariusz mówi, iż John wykonał kolejno akcje: `SHOP, COOK, EAT`.

2.7 Przykłady

2.7.1 Gotujący John

Rozważmy następującą dziedzinę:

```
initially hungry ∧ ¬hasMeal
impossible (Eat, John) if ¬hasMeal
(Cook, John) typically causes hasMeal
(Eat, John) causes ¬hasMeal
(Eat, John) releases hungry if hungry
```

Zbiór stanów dopuszczalnych to $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$, gdzie:
$\sigma_0 = \{h, \neg m\}$, $\sigma_1 = \{h, m\}$, $\sigma_2 = \{\neg h, m\}$, $\sigma_3 = \{\neg h, \neg m\}$, zaś h i m oznaczają odpowiednio *hungry* i *hasMeal*.

$Res_0(\text{Cook}, John, \sigma_0) = \Sigma$
$New(\text{Cook}, John, \sigma_0, \sigma_0) = \emptyset$
$New(\text{Cook}, John, \sigma_0, \sigma_1) = \{m\}$
$New(\text{Cook}, John, \sigma_0, \sigma_2) = \{\neg h, m\}$
$New(\text{Cook}, John, \sigma_0, \sigma_3) = \{\neg h\}$
$Res^-(\text{Cook}, John, \sigma_0) = \{\sigma_0\}$
$Res_0^+(\text{Cook}, John, \sigma_0) = \{\sigma_1, \sigma_2\}$

$ResN(\text{Cook}, John, \sigma_0) = \{\sigma_1\}$
$ResAb(\text{Cook}, John, \sigma_0) = \{\sigma_0\}$

$Res_0(\text{Eat}, John, \sigma_0) = \emptyset$,
ponieważ:
impossible (Eat, *John*) if ¬hasMeal

$ResN(\text{Eat}, John, \sigma_0) = \emptyset$,
$ResAb(\text{Eat}, John, \sigma_0) = \emptyset$,

$ResN(\text{Cook}, John, \sigma_1) = \{\sigma_1\}$
$ResAb(\text{Cook}, John, \sigma_1) = \emptyset$

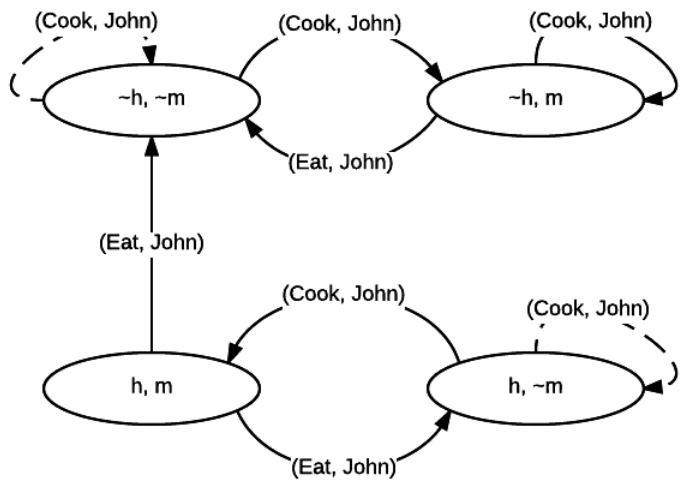
$ResN(\text{Eat}, John, \sigma_1) = \{\sigma_0, \sigma_3\}$
$ResAb(\text{Eat}, John, \sigma_1) = \emptyset$

$ResN(\text{Cook}, John, \sigma_2) = \{\sigma_2\}$
$ResAb(\text{Cook}, John, \sigma_2) = \emptyset$

$ResN(\text{Eat}, John, \sigma_2) = \{\sigma_3\}$
$ResAb(\text{Eat}, John, \sigma_2) = \emptyset$

$ResN(\text{Cook}, John, \sigma_3) = \{\sigma_2\}$
$ResAb(\text{Cook}, John, \sigma_3) = \{\sigma_3\}$

$ResN(\text{Eat}, John, \sigma_3) = \emptyset$
$ResAb(\text{Eat}, John, \sigma_3) = \emptyset$



Rysunek 1: Graf zależności - gotujący John

2.7.2 Programista Fred

Rozważmy następującą dziedzinę:

initially compiles ∧ cleanCode
(Code, Fred) **causes** ¬cleanCode
(Code, Fred) **causes** compiles **if** cleanCode
(Code, Fred) **typically causes** compiles **if** ¬cleanCode
(Refactor, Fred) **causes** cleanCode
(Debug, Fred) **causes** compiles
impossible (Code, Fred) **if** ¬compiles

Zbiór stanów dopuszczalnych to $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$, gdzie:
$\sigma_0 = \{cm, cc\}$, $\sigma_1 = \{cm, \neg cc\}$, $\sigma_2 = \{\neg cm, cc\}$, $\sigma_3 = \{\neg cm, \neg cc\}$, zaś cm i cc oznaczają odpowiednio *compiles* i *cleanCode*.

$Res_0(\text{Code}, Fred, \sigma_0) = \{\sigma_1\}$
$New(\text{Code}, Fred, \sigma_0, \sigma_1) = \{cc\}$
$Res^-(\text{Code}, Fred, \sigma_0) = \{\sigma_1\}$
$Res_0^+(\text{Code}, Fred, \sigma_0) = \{\sigma_1\}$

$ResN(\text{Code}, Fred, \sigma_0) = \{\sigma_1\}$
$ResAb(\text{Code}, Fred, \sigma_0) = \emptyset$

$Res_0(\text{Code}, Fred, \sigma_1) = \{\sigma_1, \sigma_3\}$
$New(\text{Code}, Fred, \sigma_1, \sigma_1) = \emptyset$
$New(\text{Code}, Fred, \sigma_1, \sigma_3) = \{cm\}$
$Res^-(\text{Code}, Fred, \sigma_1) = \{\sigma_1\}$
$Res_0^+(\text{Code}, Fred, \sigma_1) = \{\sigma_1\}$

$ResN(\text{Code}, Fred, \sigma_1) = \{\sigma_1\}$
$ResAb(\text{Code}, Fred, \sigma_1) = \emptyset$

$ResN(\text{Debug}, Fred, \sigma_0) = \{\sigma_0\}$
$ResAb(\text{Debug}, Fred, \sigma_0) = \emptyset$

$ResN(\text{Debug}, Fred, \sigma_1) = \{\sigma_1\}$
$ResAb(\text{Debug}, Fred, \sigma_1) = \emptyset$

$ResN(\text{Debug}, Fred, \sigma_2) = \{\sigma_0\}$
$ResAb(\text{Debug}, Fred, \sigma_2) = \emptyset$

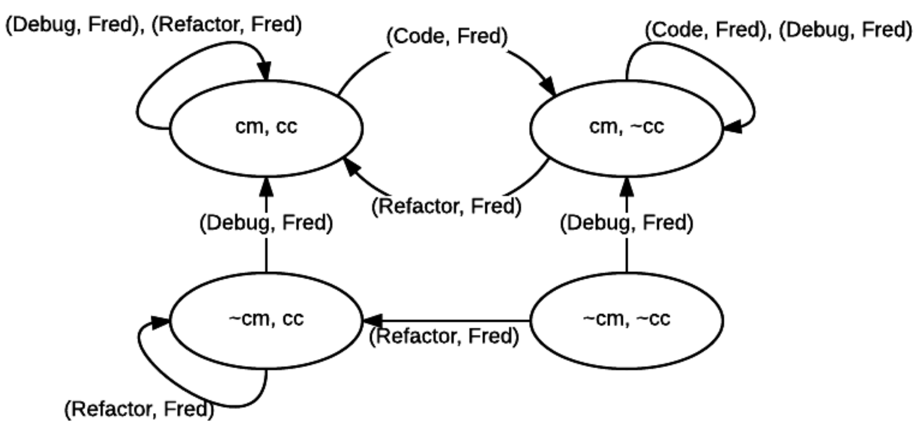
$ResN(\text{Debug}, Fred, \sigma_3) = \{\sigma_1\}$
$ResAb(\text{Debug}, Fred, \sigma_3) = \emptyset$

$ResN(\text{Refactor}, Fred, \sigma_0) = \{\sigma_0\}$
$ResAb(\text{Refactor}, Fred, \sigma_0) = \emptyset$

$ResN(\text{Refactor}, Fred, \sigma_1) = \{\sigma_0\}$
$ResAb(\text{Refactor}, Fred, \sigma_1) = \emptyset$

$ResN(\text{Refactor}, Fred, \sigma_2) = \{\sigma_2\}$
$ResAb(\text{Refactor}, Fred, \sigma_2) = \emptyset$

$ResN(\text{Refactor}, Fred, \sigma_3) = \{\sigma_2\}$
$ResAb(\text{Refactor}, Fred, \sigma_3) = \emptyset$



Rysunek 2: Graf zależności - Programista Fred

2.7.3 Kapelusznik w krainie czarów

Rozważmy następującą dziedzinę:

```
initially ¬hatterMad ∧ cakeExists ∧ elixirExists
(DRINK, Hatter) causes hatterMad if elixirExists
(EAT, Hatter) typically causes ¬hatterMad if hatterMad
impossible (EAT, Hatter) if ¬cakeExists
(DRINK, Hatter) releases elixirExists if elixirExists
(EAT, Hatter) causes ¬cakeExists
```

Zbiór stanów dopuszczalnych to $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6, \sigma_7\}$, gdzie: $\sigma_0 = \{\neg h, c, e\}$, $\sigma_1 = \{h, c, e\}$, $\sigma_2 = \{h, \neg c, e\}$, $\sigma_3 = \{\neg h, \neg c, e\}$, $\sigma_4 = \{h, \neg c, \neg e\}$, $\sigma_5 = \{h, c, \neg e\}$, $\sigma_6 = \{\neg h, c, \neg e\}$, $\sigma_7 = \{\neg h, \neg c, \neg e\}$, zaś h, c, e oznaczają odpowiednio *hatterMad*, *cakeExists*, *elixirExists*.

$Res_0(\text{DRINK}, Hatter, \sigma_0) = \{\sigma_1, \sigma_2, \sigma_4, \sigma_5\}$
$New(\text{DRINK}, Hatter, \sigma_0, \sigma_1) = \{h, e\}$
$New(\text{DRINK}, Hatter, \sigma_0, \sigma_2) = \{h, c, e\}$
$New(\text{DRINK}, Hatter, \sigma_0, \sigma_4) = \{h, c, e\}$
$New(\text{DRINK}, Hatter, \sigma_0, \sigma_5) = \{h, e\}$
$Res^-(\text{DRINK}, Hatter, \sigma_0) = \{\sigma_1, \sigma_5\}$
$Res_0^+(\text{DRINK}, Hatter, \sigma_0) = \{\sigma_1, \sigma_2, \sigma_4, \sigma_5\}$

$ResN(\text{DRINK}, Hatter, \sigma_0) = \{\sigma_1, \sigma_5\}$
$ResAb(\text{DRINK}, Hatter, \sigma_0) = \emptyset$

$Res_0(\text{EAT}, Hatter, \sigma_0) = \{\sigma_1, \sigma_2, \sigma_4, \sigma_5\}$
$New(\text{EAT}, Hatter, \sigma_0, \sigma_2) = \{h, c\}$
$New(\text{EAT}, Hatter, \sigma_0, \sigma_3) = \{c\}$
$New(\text{EAT}, Hatter, \sigma_0, \sigma_4) = \{h, c, e\}$
$New(\text{EAT}, Hatter, \sigma_0, \sigma_7) = \{c, e\}$
$Res^-(\text{EAT}, Hatter, \sigma_0) = \{\sigma_3\}$
$Res_0^+(\text{EAT}, Hatter, \sigma_0) = \{\sigma_1, \sigma_2, \sigma_4, \sigma_5\}$

$ResN(\text{EAT}, Hatter, \sigma_0) = \{\sigma_3\}$
$ResAb(\text{EAT}, Hatter, \sigma_0) = \emptyset$

$ResN(\text{DRINK}, Hatter, \sigma_1) = \{\sigma_1, \sigma_5\}$
$ResAb(\text{DRINK}, Hatter, \sigma_1) = \emptyset$

$ResN(\text{EAT}, Hatter, \sigma_1) = \{\sigma_3\}$
$ResAb(\text{EAT}, Hatter, \sigma_1) = \{\sigma_2\}$

$ResN(\text{DRINK}, Hatter, \sigma_2) = \{\sigma_2, \sigma_4\}$
$ResAb(\text{DRINK}, Hatter, \sigma_2) = \emptyset$

$ResN(\textsc{Eat}, Hatter, \sigma_2) = \emptyset$
$ResAb(\textsc{Eat}, Hatter, \sigma_2) = \emptyset$

$ResN(\textsc{Drink}, Hatter, \sigma_3) = \{\sigma_2, \sigma_7\}$
$ResAb(\textsc{Drink}, Hatter, \sigma_3) = \emptyset$

$ResN(\textsc{Eat}, Hatter, \sigma_3) = \emptyset$
$ResAb(\textsc{Eat}, Hatter, \sigma_3) = \emptyset$

$ResN(\textsc{Drink}, Hatter, \sigma_4) = \{\sigma_4\}$
$ResAb(\textsc{Drink}, Hatter, \sigma_4) = \emptyset$

$ResN(\textsc{Eat}, Hatter, \sigma_4) = \emptyset$
$ResAb(\textsc{Eat}, Hatter, \sigma_4) = \emptyset$

$ResN(\textsc{Drink}, Hatter, \sigma_5) = \{\sigma_5\}$
$ResAb(\textsc{Drink}, Hatter, \sigma_5) = \emptyset$

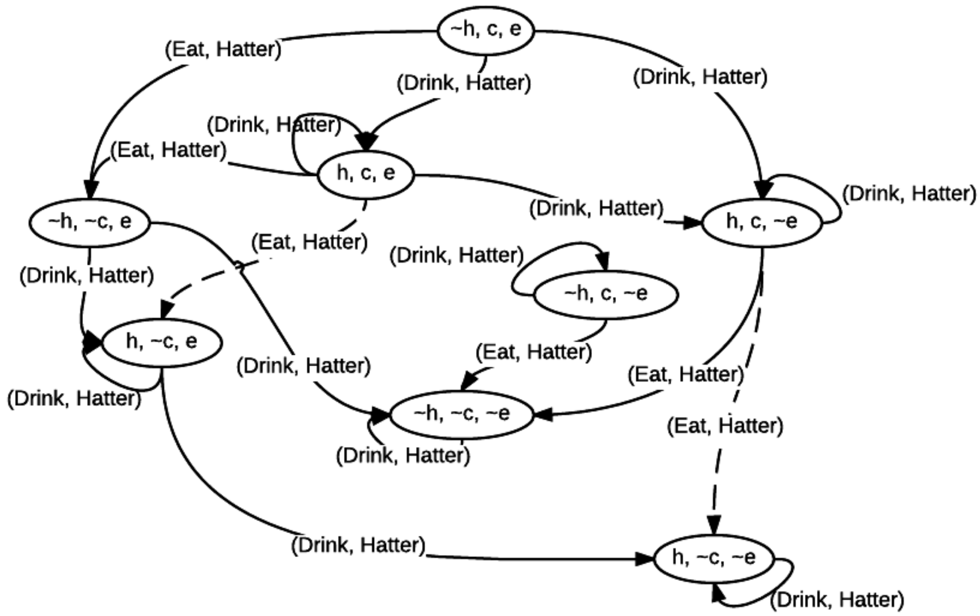
$ResN(\textsc{Eat}, Hatter, \sigma_5) = \{\sigma_7\}$
$ResAb(\textsc{Eat}, Hatter, \sigma_5) = \{\sigma_4\}$

$ResN(\textsc{Drink}, Hatter, \sigma_6) = \{\sigma_6\}$
$ResAb(\textsc{Drink}, Hatter, \sigma_6) = \emptyset$

$ResN(\textsc{Eat}, Hatter, \sigma_6) = \{\sigma_7\}$
$ResAb(\textsc{Eat}, Hatter, \sigma_6) = \emptyset$

$ResN(\textsc{Drink}, Hatter, \sigma_7) = \{\sigma_7\}$
$ResAb(\textsc{Drink}, Hatter, \sigma_7) = \emptyset$

$ResN(\textsc{Eat}, Hatter, \sigma_7) = \emptyset$
$ResAb(\textsc{Eat}, Hatter, \sigma_7) = \emptyset$



Rysunek 3: Graf zależności - Kapelusznik w krainie czarów

3 Język kwerend

W celu zadawania pytań dotyczących świata opisanego powyższym językiem został stworzony język kwerend. Składa się on z następujących pytań (kwerend) pozwalających uzyskać odpowiedzi typu PRAWDA/FAŁSZ:

3.1 Syntaktyka kwerend

- Q1 - Czy podany program działań jest wykonywalny zawsze/kiedykolwiek?
  `always/ever executable` SC
  Zdanie to mówi, że wykonując podany ciąg akcji SC z dowolnego, niesprzecznego stanu zawsze(`always`) osiągnięty zostanie niesprzeczny stan końcowy, lub istnieje przynajmniej jedna ścieżka z dowolnego z tych stanów do poprawnego stanu wynikowego(`ever`).

- Q2 - Czy wykonanie podanego programu działań z dowolnego stanu spełniającego warunek π prowadzi zawsze/kiedykolwiek/na ogół do stanu spełniającego warunek celu γ ?
  `always/ever/typically accessible` γ `if` π `when` SC
  Podobnie jak Q1 zdanie to mówi, że osiągnięcie legalnego stanu końcowego spełniającego warunek γ, startując ze stanu spełniającego warunek π, nastąpi niezależnie od wybranego punktu początkowego(`always`), przynajmniej raz(`ever`), lub zawsze pod warunkiem wyboru zawsze ścieżki akcji domyślnej(`typically`).

- Q3 - Czy z dowolnego stanu spełniającego warunek π cel γ jest osiągalny zawsze/kiedykolwiek/na ogół?
  `always/ever/typically accessible` γ `if` π

- Q4 - Czy wskazany wykonawca jest zaangażowany w realizację programu zawsze/kiedykolwiek?
  `always/ever partakes` w `when` SC

3.2 Semantyka kwerend

Kwerenda Q jest prawdziwa względem dziedziny D wtedy i tylko wtedy gdy

1. Jeśli Q jest zdaniem typu Q1:
   - z warunkiem `always`: $\forall_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D}\forall_{\Phi_S}\Phi_S(((A_1,w_1),\ldots,(A_n,w_n)),\sigma_0)$ jest określona
   - z warunkiem `ever`: $\forall_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D}\exists_{\Phi_S}\Phi_S(((A_1,w_1),\ldots,(A_n,w_n)),\sigma_0)$ jest określona
2. Jeśli Q jest zdaniem typu Q2:

- z warunkiem `always`: $\forall_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D,\sigma_0\models\pi,SC=((A_1,w_1),\ldots,(A_n,w_n))} \Phi_S(SC,\sigma_0)$ jest określona dla każdej funkcji Φ_S oraz $\Phi_S(SC,\sigma_0) \models \gamma$
- z warunkiem `ever`: $\exists_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D,\sigma_0\models\pi,SC=((A_1,w_1),\ldots,(A_n,w_n))} \Phi_S$ oraz $\Phi_S(SC,\sigma_0) \models \gamma$
- z warunkiem `typically`: $\forall_{S=(\Sigma,\sigma_0,Res_0^+)\in D,\sigma_0\models\pi,SC=((A_1,w_1),\ldots,(A_n,w_n))} \Phi_S(SC,\sigma_0)$ jest określona dla funkcji Φ_S oraz $\Phi_S(SC,\sigma_0) \models \gamma$

3. Jeśli Q jest zdaniem typu Q3:

- z warunkiem `always`: $\forall_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D,\sigma_0\models\pi} \forall_{SC=((A_1,w_1),\ldots,(A_n,w_n))}$ taki, że $\Phi_S(SC,\sigma_0)$ jest określona dla każdej funkcji Φ_S oraz $\Phi_S(SC,\sigma_0) \models \gamma$
- z warunkiem `ever`: $\forall_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D,\sigma_0\models\pi} \exists_{SC=((A_1,w_1),\ldots,(A_n,w_n))}$ taki, że $\Phi_S(SC,\sigma_0)$ jest określona dla funkcji Φ_S oraz $\Phi_S(SC,\sigma_0) \models \gamma$
- z warunkiem `typically`: $\forall_{S=(\Sigma,\sigma_0,Res_0^+)\in D,\sigma_0\models\pi} \exists_{SC=((A_1,w_1),\ldots,(A_n,w_n))}$ taki, że $\Phi_S(SC,\sigma_0)$ jest określona dla funkcji Φ_S oraz $\Phi_S(SC,\sigma_0) \models \gamma$

4. Jeśli Q jest zdaniem typu Q4:

- z warunkiem `always`: $\forall_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D,\sigma_0\models\pi} \exists_{SC=((A_1,w_1),\ldots,(A_n,w_n))}$ taki, że $\Phi_S(SC,\sigma_0)$ jest określona dla każdej funkcji Φ_S oraz $\forall_{(A_i,w_i)\in SC}$
- z warunkiem `ever`: $\forall_{S=(\Sigma,\sigma_0,ResAb,ResN)\in D,\sigma_0\models\pi} \exists_{SC=((A_1,w_1),\ldots,(A_n,w_n))}$ taki, że $\Phi_S(SC,\sigma_0)$ jest określona dla każdej funkcji Φ_S oraz $\exists_{(A_i,w_i)\in SC}$

3.3 Przykłady kwerend

3.3.1 Gotujący John

- Scenariusz: $SC = ((Shop, John), (Cook, John), (Eat, John))$
  Kwerenda: `always executable` SC
  Odpowiedź: `FALSE`
- Scenariusz: $SC = ((Shop, John), (Cook, John), (Eat, John))$
  Kwerenda: `ever executable` SC
  Odpowiedź: `TRUE`
- Scenariusz: $SC = ((Cook, John), (Eat, John))$
  Kwerenda: `always accessible hungry if angry when` SC
  Odpowiedź: `TRUE`
- Scenariusz: $SC = ((Cook, John), (Eat, John))$
  Kwerenda: `ever accessible hungry if ¬emptyFridge`
  Odpowiedź: `TRUE`

3.3.2 Programiści Fred i Bill

- Scenariusz: $SC = ((Code, Fred), (Code, Bill), (Code, Fred))$
  Kwerenda: `always executable` SC
  Odpowiedź: `FALSE`
- Scenariusz: $SC = ((Code, Fred), (Code, Bill), (Code, Fred))$
  Kwerenda: `ever executable` SC
  Odpowiedź: `TRUE`
- Scenariusz: $SC = ((Code, Fred), (Code, Bill))$
  Kwerenda: `typically accessible compiles if cleanCode when` SC
  Odpowiedź: `TRUE`
- Scenariusz: $SC = ((Code, Fred), (Code, Bill))$
  Kwerenda: `always accessible compiles if cleanCode when` SC
  Odpowiedź: `FALSE`
- Scenariusz: $SC = ((Code, Fred), (Code, Bill))$
  Kwerenda: `ever accessible compiles if ¬compiles`
  Odpowiedź: `TRUE`

3.3.3 Alicja w krainie czarów

- Scenariusz: $SC = ((Eat, Alice), (Drink, \epsilon), (Eat, Alice))$
  Kwerenda: `always executable` SC
  Odpowiedź: `FALSE`

- Scenariusz: $SC = ((Eat, Alice), (Drink, \epsilon), (Eat, Alice))$
  Kwerenda: `ever executable` SC
  Odpowiedź: `TRUE`
- Scenariusz: $SC = ((Eat, Alice), (Drink, \epsilon), (Eat, Alice))$
  Kwerenda: `ever partakes` $Rabbit$ `when` SC
  Odpowiedź: `TRUE`
- Scenariusz: $SC = ((Drink, Rabbit), (Eat, Hatter))$
  Kwerenda: `typically accessible ¬hatterMad if hatterMad ∧ ¬cakeExists ∧ elixirExists when` SC
  Odpowiedź: `TRUE`